# I/O-efficient Symbolic Model Checking

Steffan Christ Sølvsten

PhD Dissertation


Department of Computer Science
Aarhus University
Denmark

# I/O-efficient Symbolic Model Checking

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree *

by
Steffan Christ Sølvsten
August 11, 2025

*This version has been updated by Steffan Christ Sølvsten to fix small errors; the change are primarily typographic and grammatical in nature. Yet, none of the changes have not been presented to nor are endorsed by Aarhus University.

*In loving memory of Ove Sølvsten Jørgensen,*
*who taught me so much.*

*For the things, that shaped my current self:*
*Things, I now know, were once part of him:*
*Things, for which it is too late to say:*

*Thank you*

# Abstract

In this work, we develop I/O-efficient algorithms for the manipulation of binary decision diagrams (BDDs) within the context of symbolic model checking.

In our modern society, we depend every day on several critical systems. Model checking provides a way to prove that a digital system behaves according to its requirements. Having verified the system a priori, one can put it into the real world safe in the knowledge that it works as intended. This technique essentially amounts to checking by brute force whether all of the states that the system could ever find itself in are safe. Yet, systems are extremely complex, and so a naive exhaustive enumeration of all states can easily exceed the limits of any computer that has existed or ever will be.

A binary decision diagram (BDD) is a data structure with which one can reason about sets of states within much less space and time than is possible with enumerative methods. This data structure has been key for pushing the limits of what can be proven safe via model checking. Conventionally, BDDs have been – and still are – implemented by means of memoised recursive algorithms.

Arge showed in 1995 that these conventional BDD algorithms are inherently I/O-inefficient. That is, BDD computations are bottlenecked by how fast the computer can read from and write to its memory. Thirty years later his predictions have come true: BDD implementations have not been able to capitalise on the recent advances in CPU speed. Furthermore, BDD implementations are incapable of dealing with BDDs that are so large that they have to be stored on the machine's disk. To resolve this, Arge proposed to use a radically different algorithmic approach to compute on BDDs in a way which is I/O-efficient.

We follow up on Arge's theoretical contributions to develop I/O-efficient BDD algorithms for model checking. We do so with the additional goal of making these algorithms also usable in practice. To this end, we simplify and drastically improve upon Arge's original ideas. Furthermore, we extend the algorithmic technique to support all the BDD operations needed for model checking. The results of our efforts is the new BDD package, *Adiar*. Unlike other BDD implementations, our algorithms are capable of computing on BDDs that are much larger than the machine's internal memory (RAM). This is only at a slight but acceptable performance decrease for small scale BDD computations in comparison to conventional implementations.

# Resumé

I denne afhandling udvikler vi I/O-effektive algoritmer til at manipulere binære beslutningsdiagrammer (BDD'er), der kan bruges til symbolsk modelafprøvning.

I nutidens digitaliserede samfund afhænger vores hverdag af en masse kritiske systemer. Modelafprøvning tilbyder en måde, hvorpå vi kan sikre os, at et digitalt system fungerer, som ønsket. På den måde kan man finde alle fejl i et system, før det tages i brug. Dybest set fungerer dette ved at lade computeren tjekke hvorvidt der ikke findes nogen tilstande, som systemet ville kunne befinde sig i, der er uønskede. Eftersom de fleste systemer er rimelig komplekse, så er det dog ikke muligt i praksis for nogen computer at beregne en sådan udtømmende oplistning af alle systemets tilstande.

Et binært beslutningsdiagram (BDD) er en datastruktur, der har gjort det muligt at regne på mængder af tilstande på meget mindre plads og tid, end hvad der førhen var muligt ved at opliste alle tilstande særskilt. Dette har været et vigtigt skridt hen imod at kunne bruge modelafprøvere til at sikre korrektheden af store og komplekse systemer. BDD'er har – og er stadig – typisk implementeret ved hjælp af rekursion og memoisering.

Arge viste i 1995, at disse gængse BDD algoritmer er i sagens natur I/O-ineffektive. Det vil sige, hastigheden hvormed computeren kan læse fra og skrive til dets hukomelse er udslagsgivende for BDD algoritmernes hastighed. Tredive år senere kan dette nu også ses i praksis: implementationer af BDD'er er på det seneste ikke blevet hurtigere på trods af processorernes fortsatte udvikling. Samtidig er de ude af stand til at beregne på BDD'er, der er så store, at de er nød til at blive opbevaret i den eksterne hukommelse (disken). For at løse dette har Arge foreslået at lave beregninger på BDD'er med en radikalt anden algoritmemetode, som er I/O-effektiv.

Vi fortsætter Arges teoretiske arbejde med henblik på at udvikle I/O-effektive BDD algoritmer, der kan anvendes til modelafprøvning. Dette gør vi specifikt med det mål, at algoritmerne også er brugbare i praksis. Vi har derfor forenklet, forbedret og færdiggjort Arges oprindelige idéer således de understøtter alle operationer, der skal bruges til modelafprøvning. Som resultat af vores arbejde kan vi præsentere en ny implementation af BDD'er, der hedder *Adiar*. Modsat andre implementationer, så kan vores algoritmer arbejde ufotrødent på BDD'er, der er større end computerens interne hukommelse (RAM). I forhold til rekursive implementationer af BDD'er er dette gennembrud kun på bekostning af en lille men acceptabel nedsættelse af algoritmernes hastighed på små BDD'er.

# Acknowledgments

This PhD thesis was initially supposed to be about symbolic model checking of multi-agent systems. But, as fate would have it, my supervisor's early offhand comment that "*no one has ever looked into cache-oblivious binary decision diagrams*" spun me in a completely different direction. Luckily at that time, I was taking the course on I/O-efficient algorithms. So, I was able to find and understand an old and quite overlooked paper on this very topic. This paper by the late Lars Arge propelled me towards this fascinating intersection of formal methods and algorithmic engineering. Science is said to be built on the shoulders of giants. Lars Arge is certainly one of them; without his legacy, these past many years of my life would certainly have been very different.

Looking back, the path I took towards my PhD topic was a long sequence of seemingly random encounters. I want to thank my former colleagues at SCALGO for igniting my interest in I/O-efficient computation and Kira Kutscher for matching me with my supervisor. Jaco van de Pol has been an endless supply of kindness, wisdom, and thoughtfulness. Furthermore, I want to thank him for trusting me with following my own interests and judgements. I could not have hoped for a better supervisor to guide me through these past five years.

I also want to thank Randal E. Bryant and Marijn Heule for inviting me to Carnegie Mellon University and welcoming me into their research group and even their home. Getting to meet and work with both has truly been an honour. Furthermore, thanks to Fabian Pieroth, Joseph Reeves, Sean O'Connor, Jennifer Brana, Emin Berker, Colin McDonald, and the many lovely Lindy Hoppers of Pittsburgh for making my stay in the Steel City four wonderful months that I will remember fondly.

Some of the truest joys are found in the possibility to share and collaborate with others. I want to thank Anna Blume Jakobsen, Mathias W. B. Thomasen, Anders Benjamin Clausen, Kent Nielsen, Casper M. Rysgaard, and Erik F. Carstensen for taking interest and part in my PhD project. Furthermore, thanks to Mathias Rav, Jakob Truelsen, Gerth S. Brodal, Tom van Dijk, and Lasse L. Hansen for helping me find answers to multiple questions that I've had throughout these years.

Aarhus University has been such a wonderful place to have spent so many of my waking hours these past ten years. I want to thank my office mates, Mikael B. Dahlsen-Jensen, Simon Wimmer, Irfansha Shaik, and Andreas S. Larsen, that I've had the joy to talk to and laugh and boulder with. Furthermore, I especially want to thank the close-knit community at *Regnecentralen* for the many chats and laughs we have

had over audacious amounts of coffee and while eating, watching movies, or playing board games. Thanks to Andreas H. H. Hansen, Louise and Simon Dohn, Mikkel Ugilt, Magnus B. Wind, Anna Hallenberg, Anne Kirstine D. Overgaard, Mathias R. Tversted, Ulrik B. Djurtoft and many many others for their support and unending supply of hugs.

Finally, I want to thank my parents and two brothers for their never-ending support and love and their willingness to put up with my many idiosyncrasies. Sadly, my father never got to see the end of this PhD journey nor will he ever witness any of my future (mis)adventures. Even though I will continue in life not knowing what lies ahead, I will do so with an open mind and in the knowledge that I am forever in his debt.

*Steffan Christ Sølvsten,*
*Aarhus, 11th August 2025.*

# Contents

# Part I

# Overview

# Chapter 1

# Introduction[†]

As our modern society advances, more and more of our critical infrastructure depends on complex digitised systems. Doing so, it becomes vital to be able to guarantee that all hardware and software used to make up these systems behave as required. While a rigorous set of tests and decades of engineering creates a relatively high amount of confidence, it fundamentally cannot provide any guarantees on its correctness. To actually provide a proper guarantee, we need a (mathematical) proof that shows that each part of the system will behave according to its specification.

To verify such a system according to its specification, one can write down a proof with pen and paper. Yet, this does not much more than document the designer's careful ideas behind the system design; the proof will, especially if quite informal, most often subtly and unconsciously hide the error(s) in question. Instead, what is needed is a proof that is fully formalised and rigorous, so much so that it is machine checkable. This proof could be done in an interactive proof assistant, e.g. the Coq [59] proof assistant. Even though such proof assistants provide a lot of help, writing such rigorous proofs requires additional expertise and most often substantially more time than it took to develop the actual system in the first place. The properties that need to be proved are in some cases so complex that this is the only way to do so. But, quite often the property of interest is so "simple" that the machine can prove it by itself.

In many cases, such auto-generated proofs are created by means of *model checking* [57, 157]. This approach essentially boils down to checking all possible executions of the system or its individual components by brute force. To do so, the system is translated, either automatically or by hand, into a mathematical *model* where its execution is well-defined and unambiguous. In turn, this model can be expanded into a transition system (TS) with all the possible states that the model could find itself in. By checking that the TS never exhibits undesired behaviour, one proves that the model (and transitively, the system) is safe.

For example, Fig. 1.1 shows a four-state TS of a (Danish) traffic light. Of course,

[†]Since the published papers in Part II already include formal introductions to all relevant preliminary theory, we will keep this introduction light. What is lost in formal rigour is hopefully instead gained in an understanding of the bigger picture.

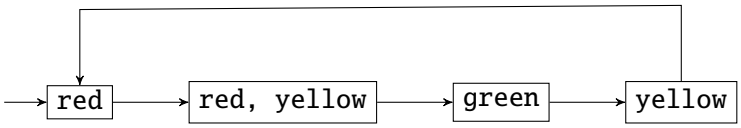



Figure 1.1: A simple transition system for a traffic light.

this particular TS ignores most of the internal logic inside of the traffic light. In reality, each of the four states in Fig. 1.1 consists of multiple states to keep track of time and/or coordinate with other signals. If care is not taken, then the number of states in such a TS can easily exceed the number of particles in the universe. Even though a lot of work has been spent on taming this *state-space explosion*, e.g. by accounting for interleaved execution in concurrent systems [84, 154, 194] or finding symmetries within the system [148], the resulting smaller TS is still often too large to be computable in practice.

Even though it may not be possible to further decrease the number of states without loosing vital information about the system's behaviour, the way the states are represented can still be made smaller. In most cases, whether the TS exhibits the desired behaviour does not require an explicit enumeration of each of its states. Instead, it suffices to merely reason about set of states. For example, a traffic light is either red and/or ($\vee$) yellow or it is exclusively ($\oplus$) green. This can be written as the following Boolean formula which represents the four unique states in Fig. 1.1,

$$(\texttt{red} \vee \texttt{yellow}) \oplus \texttt{green} \ . \tag{1.1}$$

Such *symbolic* representations have been vital for automated reasoning about vast (possibly even infinite) state spaces without running out of space or time.

To explore a TS symbolically, one has to design algorithms that reason about such sets of states. For example, Fig. 1.2 provides a simple symbolic algorithm that computes the set of reachable states in a TS. Starting with the initial set of states (`ts.initial` on line 3), it computes the next set of states (line 6) to then finally accumulate them together with the ones found previously (line 7). Key to doing so is that the TS's transitions have been translated into a Boolean relation (`ts.relation`

```
1  forward(ts):
2    prev := ∅
3    curr := ts.initial
4    do:
5      prev := curr
6      curr := relnext(curr, ts.relation)
7      curr := prev ∪ curr
8    while prev ≠ curr
9    return curr
```

Figure 1.2: A simple symbolic algorithm to compute all reachable states in a TS.

on line 6) between the variables that encode the current and next set of states. This relation is used to compute the next set of states by means of the *relational product* (`relnext`). This operation consists of an *and* ($\wedge$), an *existential quantification* ($\exists$), and a *variable substitution* ($[x \mapsto y]$) operation; we refer to Chapter 8 for the exact details on the relational product. This computation is repeated until a fixpoint is reached and no new states are found (line 8).

## 1.1 Binary Decision Diagrams

As shown in Table 1.1, one way to represent Eq. (1.1) is by means of a truth table. For each combination of *true* ($\top$) and *false* ($\bot$), this table encodes whether the formula is satisfied. Yet, the size of this table is $2^3 = 8$. In general, a truth table has an exponential size with respect to the number of variables, $n$. If such a truth table was used for symbolic model checking then nothing would be gained since it enumerates all states (and non-states) one by one.

This table can also be represented as the binary tree in Fig. 1.3. Here, each node in the tree contains one of the three colours. Each colour is treated as a Boolean *decision variable*. The nodes have two outgoing directed edges (arcs): one arc (*low*) represents said variable is set to $\bot$ whereas the other (*high*) corresponds to it being assigned to $\top$. In other words, each node represents an if-then-else decision on the decision variable.

Table 1.1: Truth Table for Eq. (1.1).

| `red` | `yellow` | `green` | Eq. (1.1) |
|---|---|---|---|
| $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| $\bot$ | $\bot$ | $\top$ | $\top$ |
| $\bot$ | $\top$ | $\bot$ | $\top$ |
| $\bot$ | $\top$ | $\top$ | $\bot$ |
| $\top$ | $\bot$ | $\bot$ | $\top$ |
| $\top$ | $\bot$ | $\top$ | $\bot$ |
| $\top$ | $\top$ | $\bot$ | $\top$ |
| $\top$ | $\top$ | $\top$ | $\bot$ |

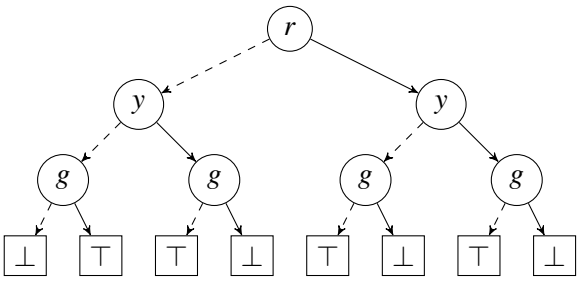



Figure 1.3: Decision tree for Eq. (1.1) with decision variables `red` ($r$), `yellow` ($y$), and `green` ($g$). High arcs are drawn solid whereas low arcs are drawn dashed.

Each path corresponds to a binary search for a specific row in Table 1.1. In particular, each path from the root to a $\top$ leaf corresponds to a unique *satisfying assignment* for the Boolean formula. In terms of size, this tree consists of $2 \cdot 2^3 = 16$ nodes and so it is larger than the truth table in Table 1.1. In general, such a decision tree requires $2^{n+1}$ nodes to store a Boolean formula with $n$ variables.

### Reduction Rules

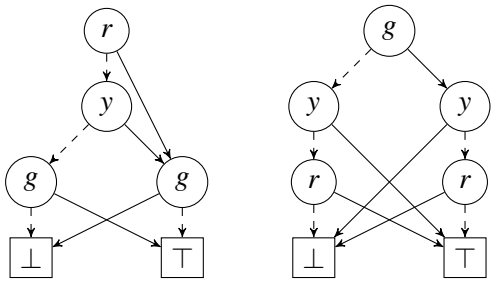
Looking at Fig. 1.3, one will notice the following two redundancies:

1. There is no need to have $2^n$ leaves (terminals) with values $\top$ and $\bot$; one can reuse the two unique terminals amongst its parents. Similarly, there are three nodes with a decision on `green` that are equivalent. In general, *dupliate* subtrees can be merged together. This turns the tree into a directed acyclic graph (DAG), i.e. a diagram.

2. The right-most decision on `yellow` in Fig. 1.3 has no effect on which terminal is reached at the end. Such *don't care* nodes may as well be skipped.

By exhaustively removing duplicates and don't cares, one obtains the *reduced* binary decision diagram [38] (reduced BDD) in Fig. 1.4a.

Whereas both the truth table and the decision tree have an exponential size, the size of the reduced BDD is in many cases only polynomial with respect to the number of variables. Of course, there could potentially be (almost) no redundancies in the worst case. If so, the reduced BDD is (almost) the same as its decision tree. As is evident from Table 1.1 and Fig. 1.3, this makes the BDD at most twice the size needed to enumerate the corresponding truth table. Particular for symbolic model checking, the BDDs are in the worst-case only as big as the explicit enumeration of all the model's states (up to a constant factor of two).




(a) $r, y, g$ (b) $g, y, r$

Figure 1.4: Binary Decision Diagrams for Eq. (1.1) for decision variables `red` ($r$), `yellow` ($y$), and `green` ($g$) and with two different variable orderings. High arcs are drawn solid whereas low arcs are drawn dashed.

### Variable Ordering

In Figs. 1.3 and 1.4a, we considered the three variables in order of `red`, `yellow`, and `green`. This makes Fig. 1.4a an *ordered* BDD for Eq. (1.1): each decision variable occurs on all paths at most once and always according to some ordering, $\pi$. It is so common to use BDDs with the aforementioned two reduction rules and a variable ordering that their usage is implied. That is, what are colloquially referred to as BDDs are in fact reduced and ordered BDDs.

The main advantage of making BDDs ordered is that the accompanying BDD algorithms can be implemented as quite simple and elegant recursive procedures [38]. At an intuitive level, by imposing an ordering, BDDs are made akin to (minimised) deterministic finite automata over binary words of fixed length $n$. Continuing this line of intuition, computing the disjunction, or any other binary operation, is done via a product construction of the two input BDDs. This is done by letting the BDD node that is "most behind" with respect to its decision variable "catch up" to the other node (see Chapter 3 for a more detailed and formal description).

As shown in Fig. 1.4b, the variable ordering can have an impact on the size of the resulting BDD. Figure 9.4 (Chapter 9) shows an example of a Boolean formula that has a polynomially or an exponentially sized BDD depending on the variable ordering used. Furthermore, there exists Boolean formulæ where the corresponding BDD has an exponential size no matter which variable ordering is applied [38].

### Canonicity

By making BDDs both ordered and reduced, the resulting BDDs become a *canonical* representation of Boolean formulæ. That is, assuming $\pi$ is fixed, two equivalent Boolean formulæ, i.e. two formulæ with equivalent truth tables, will also share the exact same ROBDD representation [38]. For example, Eq. (1.1) is equivalent to

$$(\texttt{red} \wedge \neg\texttt{green}) \vee (\texttt{yellow} \wedge \neg\texttt{green}) \vee (\neg\texttt{red} \wedge \neg\texttt{yellow} \wedge \texttt{green}) \ . \qquad (1.2)$$

Since Eq. (1.2) is equivalent to Eq. (1.1), its truth table is also the one in Table 1.1, its decision tree is the one in Fig. 1.3, and so Fig. 1.4a is also the resulting BDD.

## 1.2 The Unique and Computation Tables

Due to the key role BDDs have had in scaling symbolic model checking (see also Section 1.4) to verify much larger models, a lot of attention has been given to the design and performance of BDD implementations.

### Unique Node Table

At the heart of almost all BDD implementations is the *unique node table* [33, 141]. As shown in Fig. 1.5, this table allows one to manage the BDD node creation such that BDD nodes are always reduced [33, 141]. The first case (lines 3–4) ensures that

```
1  make_node(x, high, low):
2    // Case: Don't care
3    if low = high:
4      return low
5
6    // Case: Duplicate
7    if (unique_node_table[x, high, low] ≠ null:
8      return unique_table[x, high, low]
9
10   // Case: New node
11   return unique_table[x, high, low] = new node(x, high, low)
```

Figure 1.5: The `make_node` function from [33, 141]. This manages the creation of new reduced BDD nodes (`bdd`) with decision variable $x$ and `high` and `low` child by means of a hash table, `unique_table`.

no suppressible BDD nodes are created. The second case (lines 7–8) mitigates the construction of duplicate BDD nodes.

This decreases space usage since no redundant BDD nodes are constructed. Furthermore, BDD nodes are fully shared between the individual BDDs that are constructed via the same unique node table. Since BDDs are a canonical representation of Boolean formulæ and BDD nodes are shared, equivalence checking is a mere pointer-comparison [33, 141]. Hence, the unique node table provides an $\mathcal{O}(1)$ equivalence checking operation for BDDs [33, 141].

Due to it saving space and making equivalence checking for free, almost all implementations of BDDs during the past 35 years have relied on a unique node table to manage their BDD nodes [25, 50, 65, 93, 119, 182]. Furthermore, this hash table's design has continuously been improved upon to further increase the performance of BDD implementations [65, 109, 121, 153].

### Computation Cache

A second hash table, a so-called *computation cache* [38], is also used in almost all BDD implementations to memoise [138] the results of previous computations [25, 50, 65, 93, 119, 182]. Assuming this computation cache is large enough, this guarantees that the complexity of all BDD operations only depends on the size of its inputs [38]. In the case of symbolic model checking this means that the performance of the BDD operations does not depend on the number of states explored but merely the size of the BDD which encodes them.

For example, consider once more the disjunction of two Boolean formulæ $\phi$ and $\psi$ represented by BDDs. As mentioned in Section 1.1, this algorithm is a product construction of $\phi$'s and $\psi$'s BDDs and it is usually implemented recursively. The use of the computation cache ensures that such a recursive algorithm uses only $\mathcal{O}(N_\phi \cdot N_\psi)$ time to compute the desired result, where $N_\phi$ and $N_\psi$ are the respective sizes of

$\phi$ and $\psi$ [33, 38]. Without the computation cache, the time to compute the result would be exponential, since the recursion implicitly would unfold the BDD into the corresponding decision tree.

Making the computation cache work well in practice is slightly more complicated. The unique node table will inevitably contain a substantial number of dead BDD nodes. These will have to be cleaned up at some point by means of a garbage collection algorithm. Yet, freeing up these dead BDD nodes will invalidate the previous results stored in the cache. Hence, garbage collection can greatly increase the running time since the same subtrees end up being recomputed multiple times [33]. Furthermore, the cache is usually implemented as a lossy hash table, i.e. hash collisions result in the previously stored value being overwritten [33]. Hence, not only should the hash function and the size of the computation cache be chosen carefully [33] but it may also be beneficial to split the computation cache between separate groups of BDD operations [119]. The latter ensures that one operation cannot pollute the cache of another. Finally, only the computational results that are potentially reused later ought to be memoised [153]; otherwise, the computation cache is polluted with irrelevant information [153].

## 1.3 Beyond *Binary* Decision Diagrams

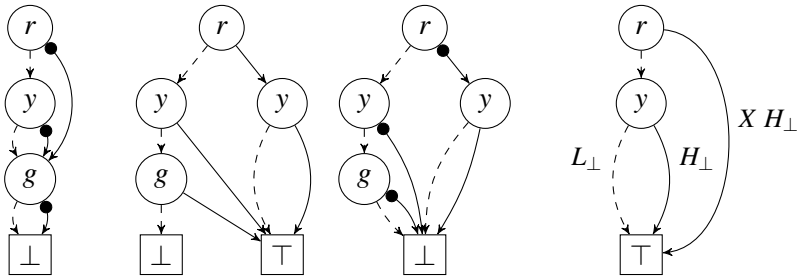
Reduced and Ordered BDDs were conceived by Bryant in 1986 [38, 39] based on the work on (unreduced and unordered) BDDs by Lee [117] and Akers [4]. As described in Section 1.2, the strength of BDD algorithms lies in their time complexity only being dependent on the size of the input and output BDDs. Hence, a lot of attention has been put towards making BDDs even smaller. Furthermore, their success (see also Section 1.4) has also spawned interest in applying them to non-Boolean domains. What follows is a non-exhaustive list of such endeavours.

### Complemented Edges

Madre and Billon [124], and Karplus [106] independently introduced in 1988 BDDs with *complemented edges* (BCDDs). As shown in Fig. 1.6a, high arcs are possibly marked ($\bullet$ on arcs in Fig. 1.6a) to denote the Boolean negation ($\neg$) of a subtree. This allows one to reuse the same subtree to both represent the formula, $\phi$, and its complement, $\neg\phi$. Furthermore, it makes Boolean negation on BDDs an $\mathcal{O}(1)$ time operation [124].

### Alternative Node Suppression

Rather than suppressing don't cares as done in BDDs, Minato proposed in 1993 to instead suppress nodes where the $\top$ terminal is only reachable along its low arc [140]. If the Boolean formula $\phi$ in question represents a sparse set, then such a *zero-suppressed* decision diagrams (ZDDs, Fig. 1.6b) can be orders of magnitude




(a) BCDD [106, 124] (b) ZDD [140] (c) ZCDD [140] (d) ESRBDD [15, 189, 190]

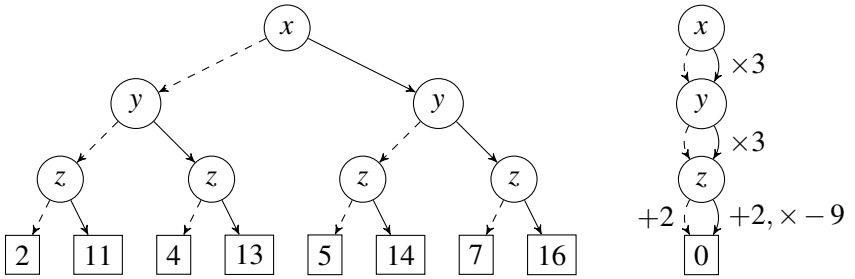
Figure 1.6: Variants of Binary Decision Diagrams for Eq. (1.1) for decision variables `red` ($r$), `yellow` ($y$), and `green` ($g$).

smaller than the corresponding BDDs. The idea of complemented edges can also be applied to ZDDs to further decrease their size (Fig. 1.6c) [140].

A lot of work has been put into merging the strengths of both BDDs and ZDDs. For example, both Bryant [40] and Van Dijk [68] have proposed to combine the strengths of both by marking consecutive blocks of variables affected by either suppression rule. Babar, Jiang, Ciardo, and Miner [15] and Thibault and Ghorbal [189, 190] take this idea one step further with *edge-specified* BDDs (ESRBDD, Fig. 1.6d) where an explicit list of suppression rules are stored on each arc, e.g. $X$ for BDD-like suppression, $H_\perp$ for ZDD-like suppression, and $L_\perp$ for the rule that is symmetric to the one in ZDDs.

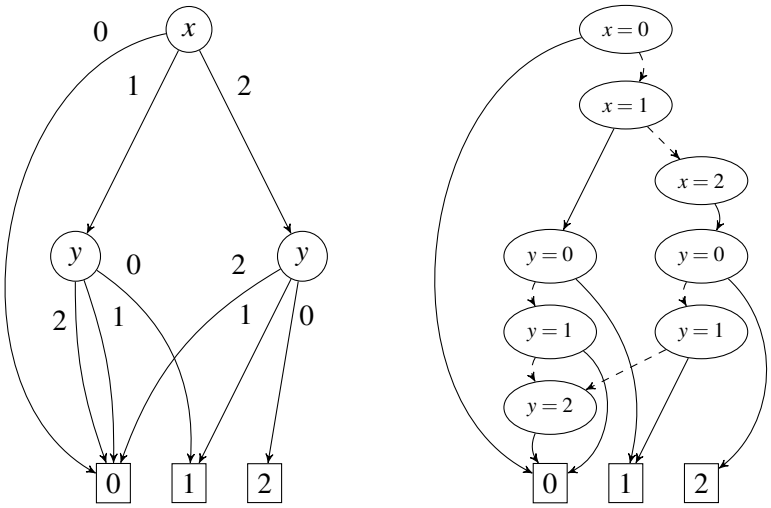
### Non-binary Terminals

BDDs themselves and all of the variations we have covered above are only concerned with the succinct representation of Boolean formulæ, i.e. $n$-ary functions $\mathbb{B}^n \to \mathbb{B}$ over the Boolean domain $\mathbb{B} = \{\perp, \top\}$. By allowing the terminal values to not only be $\top$ or $\perp$ but any value in the natural numbers, $\mathbb{N}$, or the real numbers, $\mathbb{R}$, one obtains




(a) MTBDD [79] (b) EVDD [140]

Figure 1.7: Decision Diagrams for non-binary co-domains. The MTBDD and EVDD both represent the function $f(x,y,z) = 3x + 2y - 9z + 2$ (example taken from [115]).




(a) MDD [103] (b) LDD [140]

Figure 1.8: Decision Diagrams for non-binary domains and co-domains. The MDD and LDD both represent the function $g(x,y) = \max(0, x-y)$ for $x \in \{0,1,2\}$.

the class of *multi-terminal* decision diagrams [79] (MTBDDs, Fig. 1.7a). Unlike BDDs, these can represent any function of the type $\mathbb{B}^n \to \mathbb{N}$ or $\mathbb{B}^n \to \mathbb{R}$. This allows MTBDDs to be used to succinctly represent matrices [79], certain types of transition systems [110], and probabilistic systems [78, 88, 114].

Subtrees that only differ by an additive [115] and/or a multiplicative [185] factor can also be merged together; the values by which these subtrees differ would then be stored in the diagram's arcs. Doing so turns an MTBDD into an *edge-valued* decision diagrams (EVDDs, Fig. 1.7b). These can be several orders of magnitude smaller than the corresponding MTBDD.

A special case of EVDDs are *quantum multi-valued* decision diagrams [139] (QMDDs). These have proven to be a succinct representation of large matrices over the complex domain [139, 188]. Hence, they are a vital tool for classical computers to be able to reason about programs on quantum computers [188].

## Non-binary Nodes

The class of MTBDDs can also be further extended into *multi-valued* decision diagrams [103] (MDDs, Fig. 1.8a) to deal with functions of the type $\mathbb{N}^n \to \mathbb{N}$. To do so, each BDD node has a non-binary number of outgoing arcs each of which is annotated with the respective value assigned to the node's variable.

List-decision diagrams [65] (LDDs, Fig. 1.8b) make the nodes of an MDD once again binary by moving the variable's value into the node itself. Doing so, eases implementation and allows for more node sharing (for example, the decision on $y = 2$ in Fig. 1.8b). In the worst case, this is at most at the cost of a linear increase in the diagram's size compared to MDDs.

## 1.4 The Rise and Fall (and Rise) of BDDs

### The BDD Boom

Bryant initially developed BDDs in 1986 for applications to hardware verification [38, 39]. His ideas were further developed by Madre and Billon [124] in 1988. Burch, Clarke, Long, McMillan, and Dill [46, 48] then truly showed in the beginning of the 1990s that BDDs could scale far beyond previous approaches for hardware verification.

In 1989, Bose and Fisher [32] applied BDDs to symbolic model checking of CTL [57] formulæ [46–48, 60]. In 1992, Burch, Clarke, McMillan, Dill, and Hwang [47] showed how to generalise this to the more general formulæ from the $\mu$-calculus [16, 34, 91, 112, 156]. Using BDDs, they were able to reason about models with more than $10^{20}$ states – models that still to this day are infeasible to enumerate explicitly. Coudert, Berthet, and Madre [60, 61] further proved BDDs to be useful for symbolic model checking by successfully applying them to Mealy machines [131].

These results made BDDs a staple for model checking throughout the 1990s.

### In the Shadow of SAT Solvers

BDDs quickly lost their popularity as Biere, Cimatti, Clarke, and Zhu [27] and McMillan [130] published their impressive results on symbolic model checking via Boolean Satisfiability (SAT) solvers at the turn of the millennium. At that point in time, SAT solvers improved drastically thanks to the invention of conflict-driven clause learning [18, 128, 146] (CDCL). This has made SAT-based symbolic model checking, with very good reason, supersede the BDD-based approach in many cases.

SAT solvers are, unlike BDDs, praised for being fully automatic. For example, one does not have to find a variable ordering that makes computation possible. But, saying that SAT solvers are "fully" automatic is an overstatement. SAT solvers are restricted to handling Boolean formulæ in conjunctive normal form (CNF). For example, Eq. (1.1) has to be rewritten into the following a conjunction of three clauses,

$$(\texttt{red} \vee \texttt{yellow} \vee \texttt{green}) \wedge (\neg\texttt{red} \vee \neg\texttt{green}) \wedge (\neg\texttt{yellow} \vee \neg\texttt{green}) \ . \qquad (1.3)$$

While it is possible to automatically convert any Boolean formula into its CNF representation [192], doing so obfuscates relevant structural information inside of a formula which has to be rediscovered by the CDCL SAT solver [150]. Furthermore, the inclusion (or exclusion) of redundant clauses, i.e. clauses that are implied by others, should be done with a lot of care. These clauses can guide or confuse the SAT solver's heuristics which in turn drastically changes its performance [169]. The order in which these clauses are provided can also substantially affect the time a SAT solver needs to process a formula [26]. In many ways, getting the most out of CDCL-based SAT solvers requires, quite similar to BDDs, a detailed intuition about its strengths, weaknesses, and peculiarities.

As stated by Cimatti et al. *"BDD-based and SAT-based model checking are often able to solve different classes of problems, and can therefore be seen as complementary*

*techniques*" [55]. That is, comparing BDDs to SAT solvers is like comparing apples to oranges. Bryant, Biere and Heule's recent work in [41–43] (based on [98, 172]) and especially in [45] shows that the combination of BDD-based and CDCL-based techniques has great potential to further push the current limits of SAT solvers. Another example is the Q3B [97] library which uses BDDs to reason about bitvectors on behalf of a SAT solver. In the case of symbolic model checking, the BDD-based approach is, to this day, one of many useful ways to approach the problem of model checking. The NuSMV 2 [55], LTSMIN [104], STORM [88], EPMC [78], and many other model checkers all can use BDD-based techniques when they are called for.

### Contemporary Applications of BDDs

Due the many unique properties and their strengths, BDDs are still used to this day. Lately, they have also found renewed interest.

**Hardware Verification:** Based on the ideas in [44, 136, 168], BDDs are used at Intel as a key part of their verification process for their CPU designs [99–101].

**Network Configuration:** At Microsoft, BDDs are used to verify network configurations, i.e. the rules put on routers and other small digital devices used to orchestrate complex multi-layered networks [5, 6, 37, 122]. In fact for this application, BDDs scale better than SAT [122]*.

**Feature Models:** Recently [1, 75], both BDDs and SAT solvers have been used to validate configurations of feature models, i.e. models that describe the dependencies and conflicts of different features of a system. A lot of work has been spent on deriving a good variable ordering [75, 89]. Based on [89], more work needs to be done to improve BDD-based reasoning on feature models.

**Model Checking:** As already mentioned, model checkers, such as NuSMV 2 [55] and LTSMIN [104], still actively include and use BDD-based algorithms for symbolic model checking. In the past two decades we have seen continuous efforts put towards improving BDD-based symbolic model checking. For example, the simple algorithm in Fig. 1.2 has been vastly improved in [35, 54, 66]. Furthermore, the first proper symbolic algorithm to compute the strongly connected components of a TS was only designed within the last few years [95, 116].

One can also reason about probabilistic systems symbolically. To this end, one does not do binary reasoning of whether a state is in a given set of states but the probability it is. This is possible by replacing BDDs with the MTBDDs or EVDDs mentioned in Section 1.3. For example, the probabilistic model checkers PRISM [114], STORM [88], and EPMC [78] all use MTBDDs as a possible backend.

*There exists alternatives to BDDs, e.g. disjoint difference Normal Form [28], that are even better. But, these suffer from the need to develop and maintain such niche data structures [37].

To reason about distributed and concurrent systems, one would model these as games with multiple players. BDDs are used to reason about such multi-agent systems in the symbolic model checkers MCMAS [120], MCK [82], and MCTK [184].

**Cryptography and Security:** The ZDDs shown in Section 1.3 have also been used as the underlying data structure to implement polynomials over the Boolean ring. The resulting tool, PolyBoRi [36], is not only useful for verification purposes but also in the area of cryptography [36].

Chadha, Mathur, and Schwooon used MTBDDs to analyse the leakage of private data within a program by means of symbolic model checking techniques [52].

**Program Analysis:** Abstract interpretation [102] is one of the primary procedures used within modern day compilers to do a static analysis of programs. Berndl et al. [22] showed in 2003 that BDDs can provide an efficient backend for such an analysis. In 2014, Beyer and Stahlbauer [24] added BDDs as a backend for the CPAchecker [23] tool with moderate success. The same year, Lovato, Macedonio, and Spoto [123] specially developed a thread-safe BDD package to support multi-threaded abstract interpretation algorithms in the Julia Static Analyser [183].

The Flix programming language uses a sophisticated Hindley-Milner type system to provide multiple guarantees and non-trivial optimisations. To do so, its type system uses Boolean effects [125]. Using a BDD-based backend inside of the compiler to reason about these effects has allowed them to decrease the compiler's memory usage and its running time [126]. Lately, they have switched to polynomials over the Boolean ring. But, as PolyBoRi [36] shows, decision diagrams can compress these polynomials and thereby further improve the compiler's performance.

**Quantum Computing:** As also mentioned in Section 1.3, a BDD-based approach, in the form of QMDDs [139], has proven to be vital to simulate and verify quantum circuits on classical computers [188].

### Contemporary Developments of BDDs

Despite of their age, the most popular BDD packages are still to this day the BuDDy [119] and the CUDD [182] libraries. In parts, this is because these libraries support a wide number of BDD operations. In parts, this may be due to the human labour involved in learning another BDD package is deemed not worth the benefits. But, this is also because these two libraries are, despite their age, still highly optimised and some of the fastest implementations [93, 178, 180].

There has been progress on improving the performance of BDDs in the last ten years. A lot of effort has been spent parallelising the unique node table and the computation cache (Section 1.2) [50, 65, 93] and to manage the many worker threads [50, 65, 93]. In 2016, Van Dijk and Van de Pol [65] were successful in creating a fully multi-threaded implementation of BDDs capable of parallel BDD computations without considerable thread congestions [65]. To this day, the resulting BDD package,

Sylvan, still seems to be the best at utilising the many cores that are available on modern CPU [93]. Not counting the BDD package presented in this thesis (see Chapter 2), at least four new BDD packages have been published during the past five years: Lib-BDD [19], PJBDD [25], HermesBDD [50], and OxiDD [93]. All four support thread-safety whereas the last three also all focus on modularity and multi-threading.

In 2023, Pastva and Henzinger [153] showed that the performance of BDD packages has, with respect to the CPU's speed, slowed down considerably during the past decade. In other words, even though the CPUs have gotten faster, the performance of BDD packages has not followed suit [153]. As was foreshadowed by Long [121], Klarlund [109] and especially Arge [10, 12] in the late 90s, this performance degradation is due to the fact that the unique node table and computation cache is not properly designed with respect to the machine's memory hierarchy [153]. That is, BDD computations are on modern hardware bottlenecked by the speed by which arbitrary accesses to data can be transferred from the RAM to the CPU [10, 12, 109, 153]. This problem is only exacerbated when BDD computation is multi-threaded.

# Chapter 2

# Contributions

Chapter 1 provides an introduction to the Binary Decision Diagram [38] (BDD) data structure and its applications and challenges within the context of software verification. Building on top of this, this chapter presents the research results and other contributions to the research community that are the result of our work in the past five years.

Even if BDDs (or any of its variants in Section 1.3) are the right choice for a certain model checking task, the transition system may turn out to be so large or complex that the BDDs will grow beyond the machine's RAM. At that point, the BDDs will have to be stored on the disk. Yet compared to the machine's RAM, reading from and writing to the disk is extremely slow. This setting only exacerbates the issue that was highlighted by Pastva and Henzinger [153]: conventional BDD implementations are plagued by the slowdown they experience due to cache misses.

The I/O-model [2] by Aggarwal and Vitter provides a theoretical framework with which one can analyse the data transfers (I/Os), e.g. cache misses, induced by an algorithm and its data structures; we refer to Section 2.1 in Chapter 3 for more details on the I/O-model. Using this, Arge showed in 1995 that the issue highlighted by Pastva and Henzinger [153] is fundamentally due to the reliance on recursion, a unique node table, and computation caches [10, 12]. To mitigate this, he proposed to instead design BDD algorithms based on *time-forward processing* [11, 53]. This algorithmic technique uses I/O-efficient priority queues to drastically decrease the number of I/Os. This is only at the cost of an additional log-factor in the running time. We refer to Sections 2.2.1 to 3.2 in Chapter 3 for more details on Arge's work.

This thesis follows up on Arge's theoretical contributions in [10, 12]. At its simplest, the work presented here is an investigation into how his ideas can be fully realised to create algorithms for symbolic model checking where one or more BDDs have to be stored on the disk. Unlike [10, 12], our work is not only concerned about the algorithms in theory but also in practice. This places our contributions at the intersection of *formal methods*, which provides the motivation and the benchmarks, *algorithmics*, which provides the theoretical tools to design and analyse I/O-efficient BDD algorithms, and *algorithmic engineering*, which is also concerned with the

performance of the algorithms in practice.

The contributions of our work is threefold.

1. Building on top of Arge's previous work, we improve upon and design new time-forward processing algorithms for I/O-efficient BDD manipulation.

2. These I/O-efficient algorithms have been implemented within a new, modern, and production-grade BDD library.

3. We implement a reusable and extensible benchmarking suite for BDDs with which one can evaluate the performance of different BDD implementations.

We describe these contributions in more details in the following three sections.

## 2.1 I/O-efficient Decision Diagram Manipulation

Part II contains all of our research contributions which have been published. Chapters 4 to 6, i.e. publications [175, 176, 181], are reproduced as-is with permission from Springer Nature. For consistency, all publications in Part II are reproduced exactly as they have been published. In the case of extended papers, we provide the version that was published on arXiv.

In Part III, on the other hand, we provide new theoretical contributions which address the two most pressing open problems that still remain.

### Part II Publications

As mentioned at the very beginning of Chapter 1, symbolic model checking with BDDs depends on five Boolean operations. In Chapters 3, 7 and 8, we design I/O-efficient time-forward processing algorithms for each of these operations. In particular, we address the *and* ($\wedge$), *or* ($\vee$) and *equality* ($=$) operators in Chapter 3, *existential quantification* ($\exists$) in Chapter 7, and *variable substitution* ($[x \mapsto y]$) in Chapter 8. As a result, these chapters provide a complete set of algorithms for I/O-efficient BDD-based symbolic model checking.

In practice, our algorithms can compute efficiently on BDDs larger than the machine's RAM. Furthermore, when computing on moderately sized BDDs our algorithms are only slightly slower than conventional BDD algorithms. This performance is in many ways due to the efforts to improve the algorithms which have been documented in Chapters 3 and 5 to 8. These optimisations either pertain to changing the algorithm based on some meta information about the BDD graph or to incorporate one set of computations within another.

As Fig. 2.1 shows, we initially focused quite extensively on improving the performance of the basic BDD algorithms in Chapter 3. We only proceeded onto the more complex BDD operations in Chapters 7 and 8 as performance was sufficiently improved. This is due to the fact that the experimental results in Chapters 3 and 4 indicated that our implementation had an abysmal performance on smaller BDDs. If

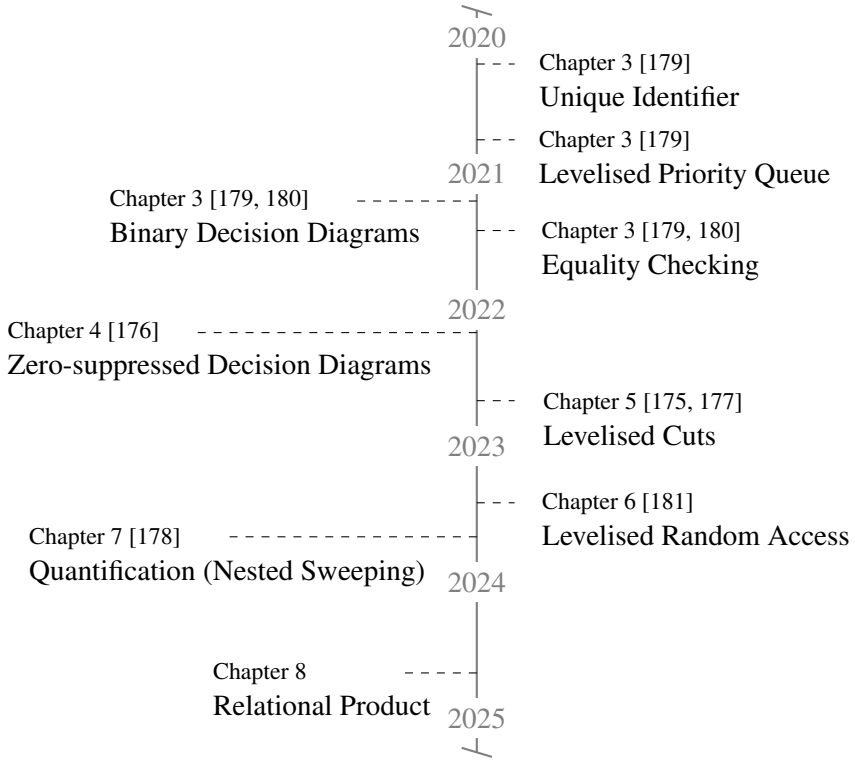



Figure 2.1: Overview of all published contributions. Contributions on the left pertain to the theoretical design of I/O-efficient Decision Diagram algorithms whereas the ones on the right pertain to improving practical performance.

left untreated, we expect these issues would have multiplied and tainted the performance of the more complex BDD operations to a such a degree that the work would have been unpublishable. In fact, as also mentioned in Chapter 3, Šmérek spend half a year back in 2009 to also attempt to implement Arge's algorithms [174]. But, the performance of the resulting implementation was disappointing [174]. We strongly believe that our prolonged focus on improving the algorithmic foundation has been vital to making the resulting algorithms useful in practice.

While we provide the chapters in the chronological order of publication, Fig. 2.1 suggests an alternative non-chronological reading order. For the I/O-efficient BDD algorithms, first read Chapter 3 (skipping Sections 3.4, 3.5, and 5.3.1 to 5.3.3), Chapter 7 (skipping Sections 3.2 and 3.3) and finally Chapter 8 (skipping Section 3 after having read Propositions 1). For the many optimisations, read the remainder of Chapter 3, then Chapters 5 and 6 and finally the remainders of Chapters 7 and 8. Chapter 4, which extends our algorithms to ZDDs, can be treated as optional material.

**Chapter 3 [179, 180]:** In this work, we investigate the foundations for how to apply time-forward processing to BDD manipulation. One key contribution is the following.

*I/O-efficient Apply–Reduce*: We simplify, improve upon, and implement Arge's I/O-efficient BDD algorithms. In particular, this covers an I/O-efficient *and* ($\wedge$) and *or* ($\vee$) which is needed for symbolic model checking.

This may only be tantamount to an incremental step on Arge's theoretical contributions. But, this chapter further provides the following improvements upon this foundation.

*Unique Identifier*: The time-forward processing technique is based on exploiting a certain ordering on the BDD nodes. We show how this ordering can be reduced to a mere integer comparison which is computationally cheap.

*Levelised Priority Queue*: In practice, a sorting algorithm is much faster than a priority queue [144]. We show how to (partially) replace the primary priority queue within the time-forward processing algorithms with much faster sorting steps. As a result, we improve the running time considerably.

*Negation*: Even though we keep things simple by not including complement edges (see Section 1.3), we show how to make *negation* ($\neg$) a constant-time operation within this I/O-efficient setting.

*Equality Checking*: The I/O-efficiency is at the cost of not using a unique node table. As a result, *equality checking* ($=$), which is key for the fixpoint computations in symbolic model checking, cannot be a constant-time operation. A naive algorithm would be insufficient since it has a quadratic running time.

- Based on the canonicity of BDDs [38] (see also Section 1.1), we are able to decrease the time and I/O complexity to only be linearithmic. In practice, this improves the performance of equality checking by an order of magnitude. Coincidentally, this algorithm has also been proposed by Hellings, Fletcher, and Haverkort [87]. But, it has been independently rediscovered by us.
- Furthermore, we show that the time-forward processing algorithms produce BDDs that adhere to a much stricter definition of canonicity than the one in [38]. As a result, a mere linear bit-wise comparison is sufficient. In the setting without a unique table, this is optimal – not only asymptotically but also with respect to the constant. In practice, this algorithm is one additional order of magnitude faster than a naive solution. In fact, it is so fast it computes the answer virtually instantaneously.

As a result of all these contributions, we present the *Adiar* BDD package which can, unlike conventional BDD packages, efficiently compute on huge BDDs beyond the limits of the machine's main memory. If the BDDs are large but still fit inside of the RAM, Adiar is only a small constant of four slower than the most popular conventional implementations.

This work was published at the International Conference on *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS '22). We provide the full paper

that was published on arXiv. This includes the proofs and details left out of [180] due to limited space. In particular this is the proof of the equality checking algorithms in Section 3.3.3, the levelised priority queue in Section 3.4, the unique identifier in Section 3.5, and additional experiments on the improvements made on Arge's original algorithm in Section 5.

**Chapter 4 [176]:** We show how the ideas presented in Chapter 3 can be generalized to also cover Zero-suppressed Decision Diagrams [140] (ZDDs, Section 1.3).

> *I/O-efficient Apply–Reduce*: We show how the algorithms from Chapter 3 can be modularised in two parts. The first part is completely oblivious to the use of time-forward processing. Instead, it is merely concerned with the diagram's semantics and it dictates whereto recurse. These *recursion requests* are managed by the second part to make the algorithm I/O-efficient.

This work was published at the *NASA Formal Methods* Symposium (NFM '23).

**Chapter 5 [175, 177]:** The experiments for Chapters 3 and 4 reveal that the performance of our algorithms are in practice affected by the amount of RAM available. This is due to the fact that the I/O-efficient data structures need time to allocate internal memory. If it it was possible to compute a priori whether these data structures will not exceed main memory then they can safely be replaced with cheaper alternatives that do not have such a high initialisation cost.

To address this, we provide the following contribution.

> *Levelised Cuts*: We rephrase the question of how large the priority queue may be into a graph-theoretic problem. In particular, we show it corresponds to graph cuts with a particular shape in the BDD DAG.
>
> - Given (upper bounds of) these cuts in the input BDDs, we prove upper bounds on the corresponding cut in the output.
> - We show how to compute (over-approximations of) these cuts for free as part of the other algorithms.

Doing so, we improve the performance of our algorithms across the board. Performance improves in particular by up to several orders of magnitude for moderate BDD sizes. As a result, the BDD size needed for our algorithms to be competitive with conventional BDD libraries is now much smaller.

This was published at the International Symposium on *Automated Technology for Verification and Analysis* (ATVA '23). We provide here the full version published on arXiv which includes theorems and proofs left out of [175] due to space constraints.

**Chapter 6 [181]:** In Chapter 5 we show how to safely change the underlying data structures based on meta information about the BDD. In this chapter, we instead show how to change the algorithm itself. In particular, we provide the following contribution.

> *Levelised Random Access*: We show that if the BDD is "narrow" relative to the size of main memory then the time-forward processing algorithms can omit their secondary priority queue(s). Instead, it can do random access into a confined window of the BDD(s).

This further improves our algorithm's performance. In particular, it considerably improves the running time for the large and huge instances.

These results were published at the International Symposium on *Model Checking Software* (SPIN '24).

**Chapter 7 [178]:** As such, the algorithms in Chapter 3 also includes support for *existential quantification* ($\exists$) which is key for symbolic model checking applications. But, that algorithm can only deal with the quantification of a single variable at a time.

To support quantification of multiple variables at once and other complex BDD operations, this work provides the following two key contributions.

> *Nested Sweeping*: We design a framework with which multiple time-forward processing algorithms can work in tandem by passing information between each other.
>
> *I/O-efficient Multi-variable Quantification*: Using the nested sweeping framework, we design a multi-variable quantification algorithm. To this end, we also identify multiple optimisations particular to this operation.

In practice, this extends our previous performance results in Chapters 3 to 6 to also apply to the quantification operations. In particular, this allows us to use Adiar to solve quantified Boolean formulæ (QBFs).

At the current time of writing, this work has only been self-published on arXiv in 2024. A slightly shorter version is submitted to an international conference.

**Chapter 8:** With Chapters 3 and 7 we have covered all BDD operations needed for symbolic model checking but variable substitution ($[x \mapsto y]$). In this chapter, we provide the following two contributions as a final step towards I/O-efficient symbolic model checking.

> *I/O-efficient Variable Substitution*: In the context of symbolic model checking, we notice that all variable substitutions are monotone with respect to the variable ordering. Using this, we design an optimal standalone algorithm for variable substitution. Furthermore, we show how to include the substitution operation inside of the surrounding operations "for free".

*A combined* `and-exists`*:* We investigate different ways to merge the *and* ($\wedge$) operation with *existential quantification* ($\exists$) to further save on computation time and I/Os.

We provide as a result of this work an I/O-efficient relational product. Hence, this chapter finally provides an evaluation of our algorithms on symbolic model checking applications. In comparison to conventional BDD libraries, our implementation is about one order of magnitude slower in practice. This is worse than the results in Chapters 3 and 7. As also mentioned in Chapter 8, based on the results of Van Dijk et al. [195], this gives reason to believe that *existential quantification* ($\exists$) can be further integrated into the *and* ($\wedge$) operation.

This work has only been self-published on arXiv in 2025. Based on feedback from peer reviews when submitting it to an international conference, we need to address performance issues on smaller instances (Chapter 11) and maybe also on the larger ones to make this publishable.

## Part III Future Work

These chapters provide unpublished work which addresses the most important research directions that have still been left to be resolved. What mainly sets these chapters apart from the ones in Part II is the lack of an implementation and hence an experimental evaluation. Doing so is left as future work and would result in making Adiar into a BDD package that is on-par with others both in terms of features and performance.

**Chapter 9:** In this chapter, we design algorithms to I/O-efficiently replace a BDD's variable order with another.

*I/O-efficient Variable Substitution*: We provide an algorithm that, unlike the one in Chapter 8, is capable of dealing with arbitrary variable substitutions. This also provides a way to switch to any arbitrary variable ordering.

*Jumps and Exchanges*: We design an asymptotically much faster algorithms for the case where the variable substitution only includes (disjoint) variable jumps and/or exchanges.

*Swaps*: We provide similar improvements if the variable substitution only consists of adjacent variable swaps.

This proves that our I/O-efficient approach is capable of dealing with BDD variable orderings at least on-par with conventional BDD operations.

**Chapter 10:** In this chapter, we repurpose the algorithms from Chapter 9 to implement the search for an optimal variable ordering of a BDD. In particular, we cover the following three methods.

*Metaheuristics*: We provide an overview of the metaheuristics that have been used previously to do BDD variable reordering. Furthermore, we show this can be translated into the context of I/O-efficient variable reordering via the algorithms from Chapter 9.

*Sifting*: We show how the algorithm in Chapter 9 for jumps can be repurposed to implement Rudell's famous sifting algorithm [161] within the context of I/O-efficient BDD manipulation.

*Parallel Sifting*: To further improve the asymptotic performance, we also describe how to use adjacent variable swaps algorithm from Chapter 9 to implement a sifting-like procedure. The result is in many ways akin to Felt's et al. 2-window algorithm [77].

This proves it possible to do variable reordering efficiently in external memory, even though the BDDs from Chapter 3 are stored in a radically different way.

**Chapter 11:** We successfully dealt with large BDDs In Chapter 3 and scaled it down to moderate instances in Chapter 5. This leaves the small instances, where the conventional algorithms are much faster.

To address this, we suggest the following.

*Time-forwarding across a Unique Node Table*: We show how the time-forward processing algorithms from Chapters 3 and 6 can be adapted to use a unique node table as an input and/or the output.

As a result, we show how our time-forward processing algorithms themselves can bridge between the recursive algorithms at the small scale and our I/O-efficient approach at the large scale. If implemented, this would make Adiar a BDD package that is efficient across the entire spectrum of BDD sizes.

This is of course not an exhaustive list of all future work left. But, these chapters address the most pressing remaining issues. This being the case is especially evident in the feedback on our work in Part II by peer reviewers, i.e. the community, consistently include requests to resolve variable ordering and performance on small instances.

Part III does not cover how to add some other BDD operations, e.g. functional composition. Yet, these are possible via the nested sweeping framework in Chapter 7. Part III neither includes how to extend our I/O-efficient algorithms to also support the other variants of BDDs such as the ones in Section 1.3. Yet, our approach easily extends to BDDs with complement edges [106, 124] and multi-terminal [79], edge-valued [115, 185], quantum multi-valued [139], list [65], and many other types of decision diagrams. In fact, Adiar's codebase has already been written with these features and extensions in mind.

## 2.2 Adiar: An External Memory BDD Library

To evaluate the algorithms and optimisations in Part II, we have implemented them in C++ on top of the TPIE [144, 196] library which, among other things, provides an I/O-efficient implementation of files, sorting algorithms, and priority queues. The result of our efforts is the BDD package *Adiar* which is publicly available under a permissive license at the following URL:

github.com/ssoelvsten/adiar

Adiar is implemented as a high-quality production-ready BDD library. In part, this is to ensure that our experimental results are meaningful; there is little value in our algorithms being faster than other BDD implementations if it is at the cost of making them unusable in practice. Furthermore, the use of proper software engineering practices when implementing Adiar follows the notion of "*slow is smooth, smooth is fast*" advocated by the Navy SEALs and others. In particular, if the implementation was treated as a prototype, then bugs would be harder to find and technical debt would accumulate. Both would in the long run slow down implementation and any progress towards obtaining the research results in Part II.

What follows is an overview of some aspects of Adiar that have not been given much or any attention in our published papers. Not only have these aspects supported its long-term development but they also increase the value it provides to the research community.

### Modular Design

Adiar is implemented with a highly modular design. In Chapter 4 [176] this allows it to support both BDDs and ZDDs by separating the logic specific to each kind of decision diagram from the external memory approach that are used to manipulate them both. Modularity further allowed us to implement levelised cuts in Chapter 5 [175, 177] and levelised random access in Chapter 6 [181] with few lines of code and without any code duplication. In fact, this modularity allowed us to implement levelised random access in a single week. Finally, the highly modular design allowed use to implement the nested sweeping framework in Chapter 7 [178] with only $\sim 1.300$ lines of reusable templated C++ code.

At current time of writing, Adiar consist of $\sim 19.000$ lines of C++ code which has been extensively documented and already designed with multiple extensions in mind, e.g. Chapters 9 and 10 and the many variations of decision diagrams in Section 1.3.

### Unit and System Testing

Adiar's modular design also allows each component to be tested in isolation. Each data type, auxiliary data structure, and algorithm has been thoroughly unit tested as part of their development. At the current time of writing, Adiar's source code is accompanied by a total of 3.462 hand-crafted unit tests that collectively provide a

code coverage of 97%; the remaining few percent are either aliases of other functions or branches that require large decision diagrams to be executed. These lines of code are easily checked by hand.

Furthermore, the many benchmarks created for the experimental evaluations in Part II have been repurposed as system integration tests. These catch obscure bugs that upon discovery have been turned into unit tests. Furthermore, they have allowed us to keep track of the library's performance after each code change.

### A Generic and Flexible API

Great care has also been made to make Adiar's API generic, support multiple use-cases, and to follow the C++ programming style. The three examples below show how this addresses issues in the C++ API of other popular BDD packages.

**Variable Substitution** Figure 2.2 shows the function for *variable substitution* ($[x \mapsto y]$) in the C++ APIs of Adiar v2.1, BuDDy v2.4, and Sylvan v1.8. See also Chapters 8 and 9 for more details on this function.

BuDDy requires the user to specify the substitution as a `bddPair*`, a data structure particular to BuDDy. This list of pairs has to be constructed using the `bdd_newpair` and `bdd_setpair` functions from BuDDy's C API. Since it is not part of its C++ API, the list of pairs also has to be manually memory managed by means of `bdd_freepair`.

Sylvan's API is slightly better in this regard. Here, the memory for the variable pairs is managed by means of the lifetime of the two `std::vectors` provided as arguments. Yet, this API forces one to provide the pairs as two separate vectors, `from` and `to`. One cannot provide a list of pairs, i.e. a single `std::vector<...>` of `std::pair<uint32_t,uint32_t>`. One can neither use any other data structure, e.g. a `std::set`, even if it would better fit one's use-case. Not only that, but one has to provide two vectors of unsigned 32-bit integers. The compiler will complain, if one provides a vector with any other integral type, signed or unsigned. Hence, the end user is either forced to convert their own data structure or to design their entire application such that it uses the data structure dictated by Sylvan.

```
// Adiar
bdd bdd_replace(bdd&, std::function<int(int)>&)

// BuDDy
bdd bdd_replace(bdd&, bddPair*)

// Sylvan
Bdd Bdd::Permute(std::vector<uint32_t>& from,
                 std::vector<uint32_t>& to)
```

Figure 2.2: API for Variable Substitution in different BDD packages

```
// Adiar
bdd bdd_exists(bdd&, int)
bdd bdd_exists(bdd&, predicate<int>&)
bdd bdd_exists(bdd&, generator<int>&)
bdd bdd_exists(bdd&, InputIt begin, InputIt end)

// BuDDy
bdd bdd_exist(bdd&, bdd&)

// CUDD
BDD BDD::ExistAbstract(BDD&)

// Sylvan
Bdd Bdd::ExistAbstract(BddSet&)
```

Figure 2.3: API for Existential Quantification in different BDD packages

Adiar's API, on the other hand, represents the variable substitution as a *function* of type `int` $\rightarrow$ `int`. Doing so, the user can provide a function that looks up the result in any data structure they like. This also implicitly handles any conversion of integers, unlike the requirement on using `uint32_t` for Sylvan. Furthermore, it also allows one to provide simple substitutions that are quicker to compute on the fly than to store. For example, one can provide the substitutions $x \mapsto x - 1$ and $x \mapsto x - n/2$ which are used a lot within symbolic model checking applications.

**Existential Quantification** Figure 2.3 shows the API for *existential quantification* in Adiar v2.1, BuDDy v2.4, CUDD v3.0, and Sylvan v1.8. What quickly becomes apparent is that Adiar makes heavy use of overloading to provide different ways to specify the to-be quantified variables. BuDDy, CUDD, and Sylvan, on the other hand, require the user to provide these variables as another BDD. Similar to variable substitution discussed above, this forces the end-user to implement a conversion from their own data structure(s) into a symbolic representation. Furthermore, such an interface is not intuitive to the end-user – why should the set of to-be quantified variables be provided as a Boolean formula?

Adiar's API on the other hand, provides multiple generalised methods of input. If only a single variable needs to be quantified, then one merely has to provide its identifier (`int`). Otherwise, one can provide a predicate (`predicate<int>`) or a co-routine (`generator<int>`), i.e. a function that respectively identifies or generates the to-be quantified variables. This provides an easy way to integrate Adiar with any data structure of ones choosing. In C++, said data structure would usually provide iterators. Hence, Adiar also allows the user to provide the variables in the idiomatic way for C++ by means of a pair of iterators, `begin` and `end`.

```
// Adiar
bdd        bdd_satmin(bdd&)
void       bdd_satmin(bdd&, consumer<pair<int, bool>>&)
OutputIt   bdd_satmin(bdd&, OutputIt iter)

// CUDD
BDD  BDD::PickOneMinterm(std::vector<BDD>)
void BDD::PickOneCube(char*)

// Sylvan
Bdd                  Bdd::PickOneCube()
std::vector<bool>    Bdd::PickOneCube(BddSet&)
```

Figure 2.4: API for Picking a Satisfying Cube in different BDD packages

**Satisfying Assignment** Where Figs. 2.2 and 2.3 were examples of the user passing input to the BDD library, Fig. 2.4 shows the functions in Adiar v2.1, CUDD v3.0, and Sylvan v1.8 for passing information back out. In particular, we show their respective functions to obtain a cube of satisfying assignment(s).

All three BDD packages support a symbolic output, i.e. a return value of type BDD. In the case of CUDD, the user has to provide the variables of interest as a `std::vector` of `BDD`s to the `BDD::PickOneMinterm` member function. Similar to the other two examples above, this again restricts the user's choice of data structure. Also similar to quantification, this cannot be said to be intuitive: this treats `BDD`s both as Boolean formulæ and also as individual variables.

If the user wants to obtain a non-symbolic representation of the satisfying assignment, CUDD requires the user to provide a C-style output array (`char*`). This means the user will have to use escape hatches from the C++ abstractions, e.g. the `data()` member function of a `std::vector<char>`. In turn, this lets go of the memory management in C++ data structures. For example, the user has to remember to `reserve()` enough memory in a `std::vector` to fit the entire output. Otherwise, they would get a segmentation fault. In Sylvan, managing the C-style array is at least taken care of by the BDD library itself. Yet, to obtain it, the user has to provide the set of BDD variables they are interested in. They then have to manually infer which variable corresponds to which index in the `std::vector` that was returned. In Adiar, on the other hand, the user can pass the output to any data structure of their liking by providing a `consumer` function. This callback function also includes the variable in question. Alternatively, the user can also obtain the output by means of iterators, e.g. one can use a `std::back_inserter` for a `std::vector` or a `std::inserter` for a `std::map`.

### Low-level Performance

The most popular BDD libraries, e.g. CUDD [182] and Sylvan [65], are also the result of many engineering hours spent on tweaking their performance. Hence, Adiar also

```
inline ptr_uint64 cnot(ptr_uint64& p, bool negate)
{
  uint64_t shifted_negate =
    ((uint64_t) negate) << ptr_uint64::data_shift;

  return p.is_terminal() ? p._raw ^ shifted_negate : p._raw;
}
```

Figure 2.5: Conditional Negation of a Unique Identifier.

includes such low-level tweaks to make the comparison in Part II as fair as possible.

One place where this becomes very apparent is the *unique identifier* described in Chapter 3 [179]. Intuitively, a unique identifier is a pointer with additional information. As shown in, Fig. 10 of Chapter 3, this is implemented by means of bit-manipulation of a single 64-bit unsigned integer; the exact bit-layout has changed slightly with Chapter 5 [175] and Chapter 7 [178] but the key idea persists.

These unique identifiers are implemented in Adiar with the `ptr_uint64` class and can be manipulated in various ways. For example, the on-the-fly negation of terminals (see Section 3.3 in Chapter 3 for more details) is implemented with the conditional negation (`cnot`) operation shown in Fig. 2.5.

Since this function is so short and is run extensively, the relative overhead of calling this function is considerable. Hence, all functions for the `ptr_uint64` class, including `cnot`, are provided as `inline` functions in a header file directly. This allows the compiler to include the content of these function content as part of the surrounding algorithms. This improves Adiar's performance by $\sim 6\%$.

The `cnot` operation works by abusing the fact that the Boolean variable `negate` is guaranteed to be 0 or 1. Hence, the Boolean can be converted to a 64-bit unsigned integer, shifted to the correct bit-position (`<<`), and then be used to flip the terminal value with a single exclusive or (`^`) operation. This requires 2 CPU cycles to compute. Furthermore, this skips a branching statement on which terminal value `p` contains. This is quite important since branch mispredictions can have a major impact on performance. In fact, the `cnot` in Fig. 2.5 can be computed without any branching since the ternary expression (`... ? ... : ...`) can be turned into a conditional move (`cmov`) instruction.

Another example is the `shift_replace` function on `ptr_uint64`. This implements the affine variable substitution in Chapter 8. It has been written with just as much attention to the final machine instructions as the `cnot` function. Where a naive implementation uses 16 machine instructions and 4 branches, our implementation uses only 12 instructions and a single branch. The one remaining branching statement can also be turned into a conditional move instruction.

## 2.3 A BDD Benchmarking Suite

To the best of our knowledge, there has at no point throughout the past 35 years been created a common set of benchmarks for BDD packages. While the Model Checking Contest [111] and the ISCAS [86], IWLS [7], EPFL [8] and other benchmark collections provide real-world inputs, no one has designed a set of easy-to-extend BDD-based applications with which to solve them.

This has put the burden onto each BDD package's developer to implement a set of benchmarks. Yet, creating and designing such benchmarks has, for various reasons, been given small amounts of attention. As a result, many publications on BDD packages include quite unsatisfactory experimental evaluations. As recently pointed out by Pastva and Henzinger, a considerable number of publications on new BDDs packages are only evaluated on "*pathological worst-case*" examples [153].

For example, all three BDD packages in [25]*, [50], and [113]† are evaluated on the *n-queens* problem. Since they share an evaluation on the same problem, one would at least expect that a coarse-grained comparison is possible after a quick glance at the papers. But, this is not the case since the same problem is solved in different ways. For example, the evaluation in [25] uses a CNF-based encoding which is not well-suited for BDDs. In [50, 113], a BDD-friendly encoding is used instead. As a result, the space usage and running times in [113] and [25] are incomparable.

Furthermore, to compare with other BDD packages, the *n*-queens benchmark was in [25] implemented separately for each BDD package. If care is not taken while implementing or updating these duplicated implementations, there could be a small but important difference between each. This could potentially nullify the validity of the experimental results.

### An Extensible and Reusable Benchmarking Suite for BDDs

As part of implementing yet another BDD package, we also had to implement yet another set of benchmarks. But, we did so with the following goals in mind:

- The respective benchmarks and BDD implementations should be modularised such that the benchmarks are independent of the BDD packages. This guarantees that each BDD package will run the exact same set of BDD operations.
- It should be simple to set up, use, and to extend with your own BDD package.
- This modularity should only exist at compile-time. The final executable should be equivalent to having hand-written an implementation for each BDD package.

*The following ought to be noted in defence of PJBDD [25]. The accompanying artifact shows that an earlier version of [25] paper was submitted to the international conference on *Tools and Algorithms for the Construction and Analysis of Systems*. Most importantly, this earlier version of the artifact also includes an integration of PJBDD into the CPACHECKER [23] verification tool.

†Kunkle, Slavici and Cooperman also evaluate their BDD package on a second combinatorial problem. But, this second problem is also a "*pathological worst-case*".

- The benchmarks ought to include enough features to be representative of real-world applications of BDDs.

- The benchmarks should, on the other hand, stay as simple as possible to not complicate the interpretation of the experimental results.

The resulting nine benchmarks have been designed to focus on the relevant BDD operations in each respective chapter of Part II. Furthermore, we have tried to provide a wide range of use-cases. In particular, these benchmarks not only cover singular BDD operations or combinatorial problems but they also include an equality checking of Boolean circuits, a solver for quantified Boolean formulæ (which are a superset of SAT problems), and the foundations for a model checker.

The benchmarks are made independent of the respective BDD package via *adapters*, i.e. header-only C++ classes that translate a common interface into the function(s) in each respective BDD package. Next to Adiar, we have implemented adapters for the depth-first BDD packages BuDDy [119], CUDD [182], and Sylvan [65]. Furthermore, thanks to Ioannis Filippidis and Tom van Dijk, we have obtained the source code for CAL [165]. After having caught up with 20 years of no maintenance, we have both revived CAL and added it to the benchmarking suite.

The entire benchmarking suite is publicly available at the following git repository.

github.com/ssoelvsten/bdd-benchmark

For the evaluation of the OxiDD [93] BDD package, this benchmarking suite was extended by Nils Husung with adapters for the Lib-BDD [19] and OxiDD [93] BDD libraries. To follow-up on the results in [75], he also extended the benchmarking suite with a CNF SAT benchmark.

### Future Work

This benchmarking suite can be extended with additional real-world applications such as the analysis of network configurations in [5, 6, 37, 122], and a simplified solver for Flix's type system in [125, 126]. The latter also addresses the current bias in the benchmarks towards large-scale BDD computations. Furthermore, while it currently only includes BDD packages written in C [65, 119, 165, 182], C++ [180], or Rust [19, 93], it can use BDD libraries from other languages, e.g. JDD [193], BeeDeeDee [123], and PJBDD [25] in Java via the *Java Native Interface*.

This benchmarking suite can also be reused for the following projects.

- *BDD Survey:* It is 27 years ago that Yang et al. [199] published their thorough experimental study on BDD performance. This benchmarking suite – especially if even more benchmarks are added – could be used as the foundation for an updated study on BDD performance and applications.

- *BDD Fuzzing:* Similar to property-based testing with QuickCheck [56], we can automatically generate BDDs and then check whether two BDD implementations agree on the result of a BDD operation. To generate useful BDDs, we can derive probability distribution based on the many unit tests for Adiar.

- *BDD Competition:* The SAT community has seen vast improvements in the performance of their solvers due to an annual competition. A similar competition could potentially push the performance of BDD packages just as much.

  Our benchmark implementations can be used for the initial creation of inputs for such a competition.

# Part II

# Publications

# Efficient Binary Decision Diagram Manipulation in External Memory*

Steffan Christ Sølvsten Jaco van de Pol

Aarhus University, Denmark {soelvsten,jaco}@cs.au.dk

Anna Blume Jakobsen Mathias Weller Berg Thomasen

February 20, 2025

**Abstract**

We follow up on the idea of Lars Arge to rephrase the Reduce and Apply procedures of Binary Decision Diagrams (BDDs) as iterative I/O-efficient algorithms. We identify multiple avenues to simplify and improve the performance of his proposed algorithms. Furthermore, we extend the technique to other common BDD operations, many of which are not derivable using Apply operations alone, and we provide asymptotic improvements for the procedures that can be derived using Apply.

These algorithms are implemented in a new BDD package, named Adiar. We see very promising results when comparing the performance of Adiar with conventional BDD packages that use recursive depth-first algorithms. For instances larger than 8.2 GiB[1], our algorithms, in parts using the disk, are 1.47 to 3.69 times slower compared to CUDD and Sylvan, exclusively using main memory. Yet, our proposed techniques are able to obtain this performance at a fraction of the main memory needed by conventional BDD packages to function. Furthermore, with Adiar we are able to manipulate BDDs that outgrow main memory and so surpass the limits of other BDD packages.

## 1 Introduction

A Binary Decision Diagram (BDD) provides a canonical and concise representation of a boolean function as an acyclic rooted graph. This turns manipulation of boolean functions into manipulation of directed acyclic graphs [10, 11].

*This is the full version of the TACAS 2022 paper [42]

[1]An error in the data analysis of resulted in this threshold being reported as 9.5 GiB in [42]. This larger number was the maximum size of the entire BDD forest rather than only the largest single BDD as was intended.

Their ability to compress the representation of a boolean function has made them widely used within the field of verification. BDDs have especially found use in model checking, since they can efficiently represent both the set of states and the state-transition function [11]. Examples are the symbolic model checkers NuSMV [17, 18], MCK [22], LTSmin [25], and MCMAS [31] and the recently envisioned symbolic model checking algorithms for CTL* in [3] and for CTLK in [24]. Bryant and Marijn [13–15] also recently devised how to use BDDs to construct extended resolution proofs to verify the result of SAT and QBF-solvers. Hence, continuous research effort is devoted to improve the performance of this data structure. For example, despite the fact that BDDs were initially envisioned back in 1986, BDD manipulation was first parallelised in 2014 by Velev and Gao [44] for the GPU and in 2016 by Dijk and Van de Pol [19] for multi-core processors [12].

The most widely used implementations of decision diagrams make use of recursive depth-first algorithms and a unique node table [9, 19, 26, 30, 43]. Lookup of nodes in this table and following pointers in the data structure during recursion both pause the entire computation while missing data is fetched [27, 34]. For large enough instances, data has to reside on disk and the resulting I/O-operations that ensue become the bottle-neck. So in practice, the limit of the computer's main memory becomes the limit on the size of the BDDs.

## 1.1 Related Work

Prior work has been done to overcome the I/Os spent while computing on BDDs. Ben-David et al. [8] and Grumberg, Heyman, and Schuster [23] have made distributed symbolic model checking algorithms that split the set of states between multiple computation nodes. This makes the BDD on each machine of a manageable size, yet it only moves the problem from upgrading main memory of a single machine to expanding the number of machines. David Long [32] achieved a performance increase of a factor of two by blocking all nodes in the unique node table based on their time of creation, i.e. with a depth-first blocking. But, in [6] this was shown to only improve the worst-case behaviour by a constant. Minato and Ishihara [34] got BDD manipulation to work on disk by serializing the depth-first traversal of the BDDs, where hash tables in main memory were used to identify prior visited nodes in the input and output streams. With limited main memory these tables could not identify all prior constructed nodes and so the serialization may include the same subgraphs multiple times. This breaks canonicity of the BDDs and may also in the worst-case result in an exponential increase of the constructed BDDs.

Ochi, Yasuoka, and Yajima [37] made in 1993 the BDD manipulation algorithms breadth-first to thereby exploit a levelwise locality on disk. Their technique has been heavily improved by Ashar and Cheong [7] in 1994 and further improved by Sanghavi et al. [40] in 1996 to produce the BDD library CAL capable of manipulating BDDs larger than the main memory. Kunkle, Slavici and Cooperman [29] extended in 2010 the breadth-first approach to distributed BDD manipulation.

The breadth-first algorithms in [7, 37, 40] are not optimal in the I/O-model, since they still use a single hash table for each level. This works well in practice, as long as a single level of the BDD can fit into main memory. If not, they still exhibit the same worst-case I/O behaviour as other algorithms [6].

In 1995, Arge [5, 6] proposed optimal I/O algorithms for the basic BDD operations Apply and Reduce. To this end, he dropped all use of hash tables. Instead, he exploited a total and topological ordering of all nodes within the graph. This is used to store all recursion requests in priority queues, s.t. they are synchronized with the iteration through the sorted input stream of nodes. Martin Šmérek attempted to implement these algorithms in 2009 exactly as they were described in [5, 6]. But, the performance of the resulting external memory symbolic model checking algorithms was disapointing, since the size of the unreduced BDD and the number of isomorphic nodes to merge grew too large and unwieldy in practice [personal communication, Sep 2021].

## 1.2 Contributions

Our work directly follows up on the theoretical contributions of Arge in [5,6]. We simplify his I/O-optimal Apply and Reduce algorithms. In particular, we modify the intermediate representation, to prevent data duplication and to save on the number of sorting operations. Furthermore, we are able to prune the output of the Apply and decrease the number of elements needed to be placed in the priority queue of Reduce, which in practice improves space use and running time performance. We also provide I/O-efficient versions of several other standard BDD operations, where we obtain asymptotic improvements for the operations that are derivable from Apply. Finally, we reduce the ordering of nodes to a mere ordering of integers and we propose a priority queue specially designed for these BDD algorithms, to obtain a considerable constant factor improvement in performance.

Our proposed algorithms and data structures have been implemented to create a new easy-to-use and open-source BDD package, named Adiar. Our experimental evaluation shows that these techniques enable the manipulation of BDDs larger than the given main memory, with only an acceptable slowdown compared to a conventional BDD package running exclusively in main memory.

## 1.3 Overview

The rest of the paper is organised as follows. Section 2 covers preliminaries on the I/O-model and Binary Decision Diagrams. We present our algorithms for I/O-efficient BDD manipulation in Section 3 together with ways to improve their performance. Section 4 provides an overview of the resulting BDD package, Adiar, and Section 5 contains the experimental evaluation of our proposed algorithms. Finally, we present our conclusions and future work in Section 6.

# 2 Preliminaries

## 2.1 The I/O-Model

The I/O-model [1] allows one to reason about the number of data transfers between two levels of the memory hierarchy, while abstracting away from technical details of the hardware, to make a theoretical analysis manageable.

An I/O-algorithm takes inputs of size $N$, residing on the higher level of the two, i.e. in *external storage* (e.g on a disk). The algorithm can only do computations on data that reside on the lower level, i.e. in *internal storage* (e.g. main memory). This internal storage can only hold a smaller and finite number of $M$ elements. Data is transferred between these two levels in blocks of $B$ consecutive elements [1]. Here, $B$ is a constant size not only encapsulating the page size or the size of a cache-line but more generally how expensive it is to transfer information between the two levels. The cost of an algorithm is the number of data transfers, i.e. the number of *I/O-operations* or just *I/Os*, it uses.

For all realistic values of $N$, $M$, and $B$ the following inequality holds.

$$N/B < \text{sort}(N) \ll N \ ,$$

where $\text{sort}(N) \triangleq N/B \cdot \log_{M/B}(N/B)$ [1] is the sorting lower bound, i.e. it takes $\Omega(\text{sort}(N))$ I/Os in the worst-case to sort a list of $N$ elements [1]. With an $M/B$-way merge sort algorithm, one can obtain an optimal $O(\text{sort}(N))$ I/O sorting algorithm [1], and with the addition of buffers to lazily update a tree structure, one can obtain an I/O-efficient priority queue capable of inserting and extracting $N$ elements in $O(\text{sort}(N))$ I/Os [4].

### 2.1.1 Cache-Oblivious Algorithms

An algorithm is cache-oblivious if it is I/O-efficient without explicitly making any use of the variables $M$ or $B$; i.e. it is I/O-efficient regardless of the specific machine in question and across all levels of the memory hierarchy [20]. We furthermore assume the relationship between $M$ and $B$ satisfies the *tall cache assumption* [20], which is that

$$M = \Omega(B^2) \ .$$

With a variation of the merge sort algorithm one can make it cache-oblivious [20]. Furthermore, Sanders [39] demonstrated how to design a cache-oblivious priority queue.

### 2.1.2 TPIE

The TPIE software library [45] provides an implementation of I/O-efficient algorithms and data structures such that the management of the $B$-sized buffers is completely transparent to the programmer.

Elements can be stored in files that act like lists and are accessed with iterators. Each iterator can **write** new elements at the end of a file or on top of prior written elements in the file. For our purposes, we only need to **push** elements to a file, i.e. **write** them at the end of a file. The iterator can traverse the file in both directions by reading the **next** element, provided **has_next** returns true. One can also **peek** the next element without moving the read head.

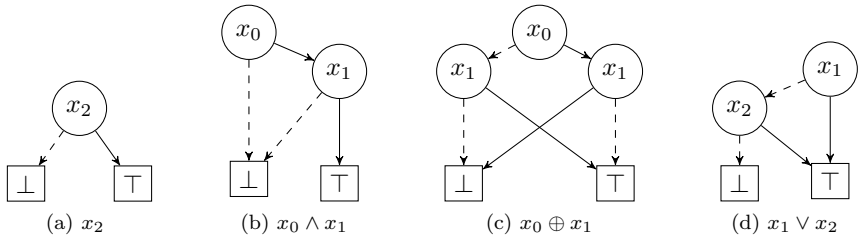
TPIE provides an optimal $O(\text{sort}(N))$ external memory merge sort algorithm for its files. It also provides a merge sort algorithm for non-persistent data that saves an initial $2 \cdot N/B$ I/Os by sorting the $B$-sized base cases before flushing them to disk the first time. Furthermore, it provides an implementation of the I/O-efficient priority queue of [39] as developed in [38], which supports the **push**, **top** and **pop** operations.

## 2.2 Binary Decision Diagrams

A Binary Decision Diagram (BDD) [10], as depicted in Fig. 1, is a rooted directed acyclic graph (DAG) that concisely represents a boolean function $\mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} = \{\top, \bot\}$. The leaves contain the boolean values $\bot$ and $\top$ that define the output of the function. Each internal node contains the *label* $i$ of the input variable $x_i$ it represents, together with two outgoing arcs: a *low* arc for when $x_i = \bot$ and a *high* arc for when $x_i = \top$. We only consider Ordered Binary Decision Diagrams (OBDD), where each unique label may only occur once and the labels must occur in sorted order on all paths. The set of all nodes with label $j$ is said to belong to the $j$th *level* in the DAG.

If one exhaustively (1) skips all nodes with identical children and (2) removes any duplicate nodes then one obtains the *Reduced Ordered Binary Decision Diagram* (ROBDD) of the given OBDD. If the variable order is fixed, this reduced OBDD is a unique canonical form of the function it represents. [10]

The two primary algorithms for BDD manipulation are called Apply and Reduce. The Apply computes the OBDD $h = f \odot g$ where $f$ and $g$ are OBDDs and $\odot$ is a function $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$. This is essentially done by recursively computing the product construction of the two BDDs $f$ and $g$ and applying $\odot$ when




(a) $x_2$ (b) $x_0 \wedge x_1$ (c) $x_0 \oplus x_1$ (d) $x_1 \vee x_2$

Figure 1: Examples of Reduced Ordered Binary Decision Diagrams. Leaves are drawn as boxes with the boolean value and internal nodes as circles with the decision variable. *Low* edges are drawn dashed while *high* edges are solid.

recursing to pairs of leaves. The Reduce applies the two reduction rules on an OBDD bottom-up to obtain the corresponding ROBDD. [10]

### 2.2.1 I/O-Complexity of Binary Decision Diagrams

Common implementations of BDDs use recursive depth-first procedures that traverse the BDD and the unique nodes are managed through a hash table [9, 19,26,30,43]. The latter allows one to directly incorporate the Reduce algorithm of [10] within each node lookup [9, 35]. They also use a memoisation table to minimise the number of duplicate computations [19, 30, 43]. If the size $N_f$ and $N_g$ of two BDDs are considerably larger than the memory $M$ available, each recursion request of the Apply algorithm will in the worst case result in an I/O-operation when looking up a node within the memoisation table and when following the low and high arcs [6, 27]. Since there are up to $N_f \cdot N_g$ recursion requests, this results in up to $O(N_f \cdot N_g)$ I/Os in the worst case. The Reduce operation transparently built into the unique node table with a *find-or-insert* function can also cause an I/O for each lookup within this table [27]. This adds yet another $O(N)$ I/Os, where $N$ is the number of nodes in the unreduced BDD.

For example, the BDD package BuDDy [30] uses a linked-list implementation for its unique node table's buckets. The index to the first node of the $i$th bucket is stored together with the $i$th node in the table. Figure 2 shows the performance of BuDDy solving the *Tic-Tac-Toe* benchmark for $N = 21$ when given variable amounts of memory (see Section 5 for a description of this benchmark). This experiment was done on a machine with 8 GiB of memory and 8 GiB of swap. Since a total of 3075 MiB of nodes are processed and all nodes are placed consecutively in memory, then all BDD nodes easily fit into the machine's main memory. That means, the slowdown from 37 seconds to 49 minutes in Fig. 2 is purely due to random access into swap memory caused by the *find-or-insert* function of the implicit Reduce.

Lars Arge provided a description of an Apply algorithm that is capable of using only $O(\text{sort}(N_f \cdot N_g))$ I/Os and a Reduce algorithm that uses $O(\text{sort}(N))$ I/Os [5,6]. He also proves these to be optimal for the level ordering of nodes on

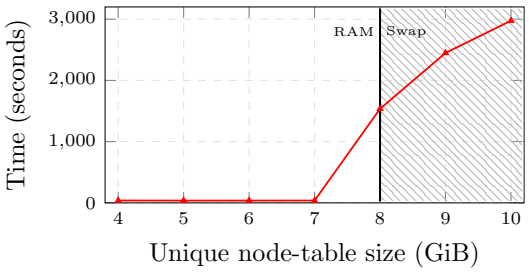



Figure 2: Running time of BuDDy [30] solving Tic-Tac-Toe for $N = 21$ on a laptop with 8 GiB memory and 8 GiB swap (lower is better).

disk used by both algorithms [6]. These algorithms do not rely on the value of $M$ or $B$, so they can be made cache-aware or cache-oblivious when using underlying sorting algorithms and data structures with those characteristics. We will not elaborate further on his original proposal, since our algorithms are simpler and better convey the algorithmic technique used. Instead, we will mention where our Reduce and Apply algorithms differ from his.

# 3 BDD Operations by Time-forward Processing

Our algorithms exploit the total and topological ordering of the internal nodes in the BDD depicted in (1) below, where parents precede their children. It is topological by ordering a node by its *label*, $i : \mathbb{N}$, and total by secondly ordering on a node's *identifier*, $id : \mathbb{N}$. This identifier only needs to be unique on each level as nodes are still uniquely identifiable by the combination of their label and identifier.

$$(i_1, id_1) < (i_2, id_2) \equiv i_1 < i_2 \vee (i_1 = i_2 \wedge id_1 < id_2) \tag{1}$$

We write the *unique identifier* $(i, id) : \mathbb{N} \times \mathbb{N}$ for a node as $x_{i,id}$.

BDD nodes do not contain an explicit pointer to their children but instead the children's unique identifier. Following the same notion, leaf values are stored directly in the leaf's parents. This makes a node a triple $(uid, low, high)$ where $uid : \mathbb{N} \times \mathbb{N}$ is its unique identifier and *low* and *high* : $(\mathbb{N} \times \mathbb{N}) + \mathbb{B}$ are its children. The ordering in (1) is lifted to compare the *uid*s of two nodes, and so a BDD is represented by a file with BDD nodes in sorted order. For example, the BDDs in Fig. 1 would be represented as the lists depicted in Fig. 3. This ordering of nodes can be exploited with the *time-forward processing* technique, where recursive calls are not executed at the time of issuing the request but instead when the element in question is encountered later in the iteration through the given input file. This is done with one or more priority queues that follow the same ordering as the input and deferring recursion by pushing the request into the priority queues.

1a: $[\ (x_{2,0}, \bot, \top)\ ]$
1b: $[\ (x_{0,0}, \bot, x_{1,0})\ ,\ (x_{1,0}, \bot, \top)\ ]$
1c: $[\ (x_{0,0}, x_{1,0}, x_{1,1})\ ,\ (x_{1,0}, \bot, \top)\ ,\ (x_{1,1}, \top, \bot)\ ]$
1d: $[\ (x_{1,0}, x_{2,0}, \top)\ ,\ (x_{2,0}, \bot, \top)\ ]$

Figure 3: In-order representation of BDDs of Fig. 1

The Apply algorithm in [6] produces an unreduced OBDD, which is turned into an ROBDD with Reduce. The original algorithms of Arge solely work on a node-based representation. Arge briefly notes that with an arc-based representation, the Apply algorithm is able to output its arcs in the order needed by the following Reduce, and vice versa. Here, an arc is a triple $(source, is\_high, target)$ (written as $source \xrightarrow{is\_high} target$) where $source : \mathbb{N} \times \mathbb{N}$, $is\_high : \mathbb{B}$, and

*target* : $(\mathbb{N} \times \mathbb{N}) + \mathbb{B}$, i.e. the *source* contains the unique identifier of internal nodes while the *target* either contains a unique identifier or the value of a leaf. We have further pursued this idea of an arc-based representation and can conclude that the algorithms indeed become simpler and more efficient with an arc-based output from Apply. On the other hand, we see no such benefit over the more compact node-based representation in the case of Reduce. Hence as is depicted in Fig. 4, our algorithms work in tandem by cycling between the node-based and arc-based representation.

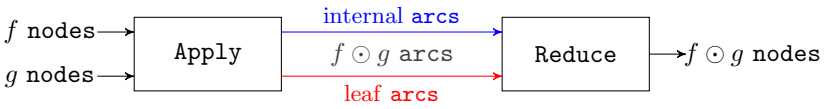



Figure 4: The Apply–Reduce pipeline of our proposed algorithms

Notice that our Apply outputs two files containing arcs: arcs to internal nodes (blue) and arcs to leaves (red). Internal arcs are output at the time of their target are processed, and since nodes are processed in ascending order, internal arcs end up being sorted with respect to the unique identifier of their target. This groups all in-going arcs to the same node together and effectively reverses internal arcs. Arcs to leaves, on the other hand, are output at the time of their source is processed, which groups all out-going arcs to leaves together. These two outputs of Apply represent a semi-transposed graph, which is exactly of the form needed by the following Reduce. For example, the Apply on the node-based ROBDDs in Fig. 1a and 1b with logical implication as the operator will yield the arc-based unreduced OBDD depicted in Fig. 5.

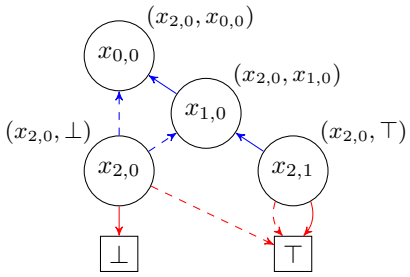
For simplicity, we will ignore any cases of leaf-only BDDs in our presentation of the algorithms. They are easily extended to also deal with those cases.

## 3.1 Apply

Our Apply algorithm works by a single top-down sweep through the input DAGs. Internal arcs are reversed due to this top-down nature, since an arc between two internal nodes can first be resolved and output at the time of the arc's target. These arcs are placed in the file $F_{internal}$. Arcs from nodes to leaves are placed in the file $F_{leaf}$.

The algorithm itself essentially works like the standard Apply algorithm. Given a recursion request for the pair of input nodes $v_f$ from $f$ and $v_g$ from $g$, a single node $v$ is created with label $\min(v_f.uid.label, v_g.uid.label)$ and recursion requests $r_{low}$ and $r_{high}$ are creatd for its two children. If the label of $v_f.uid$ and $v_g.uid$ are equal, then $r_{low} = (v_f.low, v_g.low)$ and $r_{high} = (v_f.high, v_g.high)$. Otherwise, $r_{low}$, resp. $r_{high}$, contains the *uid* of the low child, resp. the high child, of $\min(v_f, v_g)$, whereas $\max(v_f.uid, v_g.uid)$ is kept as is.

The **RequestsFor** function computes the requests $r_{low}$ and $r_{high}$ described above as shown in Fig. 6. It resolves the request for the tuple $(t_f, t_g)$ where at




(a) Semi-transposed graph. (pairs indicate nodes in Fig. 1a and 1b, respectively)

| internal `arcs` | leaf `arcs` |
|---|---|
| $[\ x_{0,0} \xrightarrow{\top} x_{1,0}$ , | $[\ x_{2,0} \xrightarrow{\bot} \top$ , |
| $x_{1,0} \xrightarrow{\bot} x_{2,0}$ , | $x_{2,0} \xrightarrow{\top} \bot$ , |
| $x_{0,0} \xrightarrow{\bot} x_{2,0}$ , | $x_{2,1} \xrightarrow{\bot} \top$ , |
| $x_{1,0} \xrightarrow{\top} x_{2,1}\ ]$ | $x_{2,1} \xrightarrow{\top} \top\ ]$ |

(b) In-order arc-based representation.

Figure 5: Unreduced output of Apply when computing $x_2 \Rightarrow (x_0 \wedge x_1)$

least one of them identify an internal node. The arguments $v_f$ and $v_g$ are the nodes currently read, where we can only guarantee $t_f = v_f.uid$ or $t_g = v_g.uid$ but not necessarily that both match. If the label of $t_f$ and $t_g$ are the same but only one of them matches, then *low* and *high* contains the children of the other.

The surrounding pieces of the Apply algorithm shown in Fig. 7 are designed such that all the information needed by **RequestsFor** is made available in an I/O-efficient way. Input nodes $v_f$ and $v_g$ are read on lines 12 – 14 from $f$ and $g$ as a merge of the two sorted lists based on the ordering in (1). Specifically, these lines maintain that $t_{seek} \leq v_f$ and $t_{seek} \leq v_g$, where the $t_{seek}$ variable itself is a monotonically increasing unique identifier that changes based on the given recursion requests for a pair of nodes $(t_f, t_g)$. These requests are synchronised with this traversal of the nodes by use of the two priority queues $Q_{app:1}$ and $Q_{app:2}$. $Q_{app:1}$ has elements of the form $(s \xrightarrow{is\_high} (t_f, t_g))$ while $Q_{app:2}$ has elements of the form $(s \xrightarrow{is\_high} (t_f, t_g), low, high)$. The use for the boolean *is_high* and the unique identifiers $s$, *low*, and *high* will become apparent below.

The priority queue $Q_{app:1}$ is aligned with the node $\min(t_f, t_g)$, i.e. the one of $t_f$ and $t_g$ that is encountered first. That is, its elements are sorted in ascending order based on $\min(t_f, t_g)$ of each request. Requests to the same $(t_f, t_g)$ are grouped together by secondarily sorting the tuples lexicographically. The algorithm maintains the following invariant between the current nodes $v_f$ and $v_g$ and the requests within $Q_{app:1}$.

$$\forall\ (s \xrightarrow{is\_high} (t_f, t_g)) \in Q_{app:1} : v_f \leq t_f \wedge v_g \leq t_g\ .$$

The second priority queue $Q_{app:2}$ is used in the case of $t_f.label = t_g.label$ and

```
1   RequestsFor((t_f, t_g), v_f, v_g, low, high, ⊙)
2     r_low, r_high
3
4     // Combine accumulated information for requests
5     if t_g ∈ {⊥,⊤} ∨ (t_f ∉ {⊥,⊤} ∧ t_f.label < t_g.label)
6     then r_low ← (v_f.low, t_g); r_high ← (v_f.high, t_g)
7     else if t_f ∈ {⊥,⊤} ∨ (t_f.label > t_g.label)
8     then r_low ← (t_f, v_g.low); r_high ← (t_f, v_g.high)
9     else
10      if t_f = v_f.uid ∧ t_g = v_g.uid
11      then r_low ← (v_f.low, v_g.low); r_high ←(v_f.high, v_g.high)
12      else if t_f = v_f.uid
13      then r_low ← (v_f.low, low); r_high ← (v_f.high, high)
14      else r_low ← (low, v_g.low); r_high ← (high, v_g.high)
15
16    // Apply ⊙ if need be
17    if r_low[0] ∈ {⊥,⊤} ∧ r_low[1] ∈ {⊥,⊤}
18    then r_low ← r_low[0] ⊙ r_low[1]
19
20    if r_high[0] ∈ {⊥,⊤} ∧ r_high[1] ∈ {⊥,⊤}
21    then r_high ← r_high[0] ⊙ r_high[1]
22
23    return r_low, r_high
```

Figure 6: The `RequestsFor` subroutine for Apply

$t_f.id \neq t_g.id$, i.e. when **RequestsFor** needs information from both nodes to resolve the request but they are not guaranteed to be visited simultaneously. To this end, it is sorted by $\max(t_f, t_g)$ in ascending order, i.e. the second of the two to be visited, and ties are again broken lexicographically. The invariant for this priority queue is comparatively more intricate.

$$\begin{aligned}\forall(s \xrightarrow{is\_high} (t_f, t_g), low, high) \in Q_{app:2} \; : \; & t_f.label = t_g.label = \min(v_f, v_g).label \\ & \wedge\, t_f.id \neq t_g.id \\ & \wedge\, t_f < t_g \implies t_f \leq v_g \leq t_g \\ & \wedge\, t_g < t_f \implies t_g \leq v_f \leq t_f\end{aligned}$$

In the case that a request is made for a tuple $(t_f, t_g)$ with the same label but different identifiers, then they are not necessarily available at the same time. Rather, the node with the unique identifier $\min(t_f, t_g)$ is visited first and the one with $\max(t_f, t_g)$ some time later. Hence, requests $s \xrightarrow{is\_high} (t_f, t_g)$ are moved on lines 19 – 23 from $Q_{app:1}$ into $Q_{app:2}$ when $\min(t_f, t_g)$ is encountered. Here, the request is extended with the *low* and *high* of the node $\min(v_f, v_g)$ such that the children of $\min(v_f, v_g)$ are available at $\max(v_f, v_g)$, despite the fact that $\min(v_f, v_g).uid < \max(v_f, v_g).uid$.

The requests from $Q_{app:1}$ and $Q_{app:2}$ are merged on lines 10 with the **TopOf**

```
1  Apply(f, g, ⊙)
2    F_internal ← [ ]; F_leaf ← [ ]; Q_app:1 ← ∅; Q_app:2 ← ∅
3    v_f ← f.next(); v_g ← g.next(); id ← 0; label ← undefined
4
5    /* Insert request for root (v_f, v_g) */
6    Q_app:1.push(NIL --undefined--> (v_f.uid, v_g.uid))
7
8    /* Process requests in topological order */
9    while Q_app:1 ≠ ∅ ∨ Q_app:2 ≠ ∅ do
10     (s --is_high--> (t_f, t_g), low, high) ← TopOf(Q_app:1, Q_app:2)
11
12     t_seek ← if low, high = NIL then min(t_f, t_g) else max(t_f, t_g)
13     while v_f.uid < t_seek ∧ f.has_next() do v_f ← f.next() od
14     while v_g.uid < t_seek ∧ g.has_next() do v_g ← g.next() od
15
16     if low = NIL ∧ high = NIL ∧ t_f ∉ {⊥, ⊤} ∧ t_g ∉ {⊥, ⊤}
17                  ∧ t_f.label = t_g.label ∧ t_f.id ≠ t_g.id
18     then /* Forward information of min(t_f, t_g) to max(t_f, t_g) */
19       v ← if t_seek = v_f then v_f else v_g
20       while Q_app:1.top() matches _ → (t_f, t_g) do
21         (s --is_high--> (t_f, t_g)) ← Q_app:1.pop()
22         Q_app:2.push(s --is_high--> (t_f, t_g), v.low, v.high)
23       od
24     else /* Process request (t_f, t_g) */
25       id ← if label ≠ t_seek.label then 0 else id+1
26       label ← t_seek.label
27
28       /* Forward or output out-going arcs */
29       r_low, r_high ← RequestsFor((t_f, t_g), v_f, v_g, low, high, ⊙)
30       (if r_low ∈ {⊥, ⊤} then F_leaf else Q_app:1).push(x_label,id --⊥--> r_low)
31       (if r_high ∈ {⊥, ⊤} then F_leaf else Q_app:1).push(x_label,id --⊤--> r_high)
32
33       /* Output in-going arcs */
34       while Q_app:1 ≠ ∅ ∧ Q_app:1.top() matches (_ → (t_f, t_g)) do
35         (s --is_high--> (t_f, t_g)) ← Q_app:1.pop()
36         if s ≠ NIL then F_internal.push(s --is_high--> x_label,id)
37       od
38       while Q_app:1 ≠ ∅ ∧ Q_app:2.top() matches (_ → (t_f, t_g), _, _) do
39         (s --is_high--> (t_f, t_g), _, _) ← Q_app:2.pop()
40         if s ≠ NIL then F_internal.push(s --is_high--> x_label,id)
41       od
42   od
43   return F_internal, F_leaf
```

Figure 7: The Apply algorithm

function such that they are synchronised with the order of nodes in $f$ and $g$ and maintain the above invariants. If both $Q_{app:1}$ and $Q_{app:2}$ are non-empty, then let $r_1 = (s_1 \xrightarrow{b_1} (t_{f:1}, t_{g:1}))$ be the top element of $Q_{app:1}$ and let the top element of $Q_{app:2}$ be $r_2 = (s_2 \xrightarrow{b_2} (t_{f:2}, t_{g:2}), low, high)$. Then $\mathbf{TopOf}(Q_{app:1}, Q_{app:2})$ outputs $(r_1, \texttt{Nil}, \texttt{Nil})$ if $\min(t_{f:1}, t_{g:1}) < \max(t_{f:2}, t_{g:2})$ and $r_2$ otherwise. If either one is empty, then it equivalently outputs the top request of the other.

When a request is resolved, then the newly created recursion requests $r_{low}$ and $r_{high}$ to its children are placed at the end of $F_{leaf}$ or pushed into $Q_{app:1}$ on lines 28 – 31, depending on whether it is a request to a leaf or not. All ingoing arcs to the resolved request are output on lines 33 – 41. To output these ingoing arcs the algorithm uses the last two pieces of information that was forwarded: the unique identifier $s$ for the source of the request and the boolean *is_high* for whether the request was following a high arc.

The arc-based output greatly simplifies the algorithm compared to the original proposal of Arge in [6]. Our algorithm only uses two priority queues rather than four. Arge's algorithm, like ours, resolves a node before its children, but due to the node-based output it has to output this entire node before its children. Hence, it has to identify its children by the tuple $(t_f, t_g)$ which doubles the amount of space used and it also forces one to relabel the graph afterwards costing yet another $O(\text{sort}(N))$ I/Os. Instead, the arc-based output allows us to output the information at the time of the children and hence we are able to generate the label and its new identifier for both parent and child. Arge's algorithm neither forwarded the source $s$ of a request, so repeated requests to the same pair of nodes were merely discarded upon retrieval from the priority queue, since they carried no relevant information. Our arc-based output, on the other hand, makes every element placed in the priority queue forward a source $s$, vital for the creation of the transposed graph.

**Proposition 3.1** (Following Arge 1996 [6])**.** *The Apply algorithm in Fig. 7 has I/O complexity $O(sort(N_f \cdot N_g))$ and $O((N_f \cdot N_g) \cdot \log(N_f \cdot N_g))$ time complexity, where $N_f$ and $N_g$ are the respective sizes of the BDDs for $f$ and $g$.*

*Proof.* It only takes $O((N_f + N_g)/B)$ I/Os to read the elements of $f$ and $g$ once in order. There are at most $2 \cdot N_f \cdot N_g$ many arcs being outputted into $F_{internal}$ or $F_{leaf}$, resulting in at most $O((N_f \cdot N_g)/B)$ I/Os spent on writing the output. Each element creates up to two requests for recursions placed in $Q_{app:1}$, each of which may be reinserted in $Q_{app:2}$ to forward data across the level, which totals $O(\text{sort}(N_f \cdot N_g))$ I/Os. All in all, an $O(\text{sort}(N_f \cdot N_g))$ number of I/Os are used.

The worst-case time complexity is derived similarly, since next to the priority queues and reading the input only a constant amount of work is done per request. □

### 3.1.1 Pruning by Short Circuiting the Operator

The Apply procedure as presented above, like Arge's original algorithm in [4,6], follows recursion requests until a pair of leaves are met. Yet, for example in

Fig. 5 the node for the request $(x_{2,0}, \top)$ is unnecessary to resolve, since all leaves of this subgraph trivially will be $\top$ due to the implication operator. The later Reduce will remove any nodes with identical children bottom-up and so this node will be removed in favour of the $\top$ leaf.

This observation implies we can resolve the requests earlier for any operator that can short circuit the resulting leaf. To this end, one only needs to extend the **RequestsFor** to be the **ShortCircuitedRequestsFor** function in Fig. 8 which immediately outputs a request to the relevant leaf value. This decreases the number of requests placed in $Q_{app:1}$ and $Q_{app:2}$. Furthermore, it decreases the size of the final unreduced BDD, which again in turn will speed up the following Reduce.

```
1  ShortCircuitingRequestsFor((t_f , t_g) , v_f , v_g , low , high , ⊙)
2    (r_low , r_high) ← RequestsFor((t_f , t_g) , v_f , v_g , low , high , ⊙)
3
4    if r_low matches (t'_f, t'_g)
5    then if t'_f ∈ {⊥, ⊤} ∧ t'_f ⊙ ⊥ = t'_f ⊙ ⊤
6         then r_low ← t'_f ⊙ ⊥
7
8         if t'_g ∈ {⊥, ⊤} ∧ ⊥ ⊙ t'_g = ⊤ ⊙ t'_g
9         then r_low ← ⊥ ⊙ t'_g
10
11   if r_high matches (t'_f, t'_g)
12   then if t'_f ∈ {⊥, ⊤} ∧ t'_f ⊙ ⊥ = t'_f ⊙ ⊤
13        then r_high ← t'_f ⊙ ⊥
14
15        if t'_g ∈ {⊥, ⊤} ∧ ⊥ ⊙ t'_g = ⊤ ⊙ t'_g
16        then r_high ← ⊥ ⊙ t'_g
17
18   return r_low , r_high
```

Figure 8: The `ShortCircuitingRequestsFor` subroutine for Apply

## 3.2 Reduce

Our Reduce algorithm in Fig. 9 works like other explicit variants with a single bottom-up sweep through the unreduced OBDD. Since the nodes are resolved and output in a bottom-up descending order then the output is exactly in the reverse order as it is needed for any following Apply. We have so far ignored this detail, but the only change necessary to the Apply algorithm in Section 3.1 is for it to read the list of nodes of $f$ and $g$ in reverse.

The priority queue $Q_{red}$ is used to forward the reduction result of a node $v$ to its parents in an I/O-efficient way. $Q_{red}$ contains arcs from unresolved sources $s$ in the given unreduced OBDD to already resolved targets $t'$ in the ROBDD under construction. The bottom-up traversal corresponds to resolving all nodes

```
1  Reduce(F_internal , F_leaf )
2    F_out ←[ ]; Q_red ← ∅
3    while Q_red ≠ ∅ ∨ F_leaf . has_next () do
4      j ← max(Q_red . top (). source . label , F_leaf . peek (). source . label )
5      id ← MAX_ID;
6      F_j ←[ ]; F_j:1 ←[ ]; F_j:2 ←[ ]
7
8      while Q_red . top (). source . label = j do
9        e_high ← PopMax(Q_red , F_leaf )
10       e_low ← PopMax(Q_red , F_leaf )
11       if e_high . target = e_low . target
12       then F_j:1 . push ([e_low . source ↦ e_low . target ])
13       else F_j . push ((e_low . source , e_low . target , e_high . target ))
14     od
15
16     sort v ∈ F_j by v . low and secondly by v . high
17     v' ← undefined
18     for each v ∈ F_j do
19       if v' is undefined or v . low ≠ v' . low or v . high ≠ v' . high
20       then
21         id ← id − 1
22         v' ← (x_j,id , v . low , v . high )
23         F_out . push (v)
24       F_j:2 . push ([v . uid ↦ v' . uid ])
25     od
26
27     sort [uid ↦ uid' ]∈ F_j:2 by uid in descending order
28     for each [uid ↦ uid' ] ∈ MergeMaxUid(F_j:1 , F_j:2 ) do
29       while arcs from F_internal . peek () matches _ -?-> uid do
30         (s --is_high--> uid ) ← F_internal . next ()
31         Q_red . push (s --is_high--> uid' )
32       od
33     od
34   od
35   return F_out
```

Figure 9: The Reduce algorithm

in descending order. Hence, arcs $s \xrightarrow{is\_high} t'$ in $Q_{red}$ are first sorted on $s$ and secondly on *is_high*; the latter simplifies retrieving low and high arcs on lines 9 and 10, since the high arc is always retrieved first. The base-cases for the Reduce algorithm are the arcs to leaves in $F_{leaf}$, which follow the exact same ordering. Hence, on lines 9 and 10, arcs in $Q_{red}$ and $F_{leaf}$ are merged using the **PopMax** function that retrieves the arc that is maximal with respect to this ordering.

Since nodes are resolved in descending order, $F_{internal}$ follows this ordering on the arc's target when elements are read in reverse. The reversal of arcs in

$F_{internal}$ makes the parents of a node $v$, to which the reduction result is to be forwarded, readily available on lines 27 – 33.

The algorithm otherwise proceeds similarly to other Reduce algorithms. For each level $j$, all nodes $v$ of that level are created from their high and low arcs, $e_{high}$ and $e_{low}$, taken out of $Q_{red}$ and $F_{leaf}$. The nodes are split into the two temporary files $F_{j:1}$ and $F_{j:2}$ that contain the mapping $[uid \mapsto uid']$ from the unique identifier $uid$ of a node in the given unreduced BDD to the unique identifier $uid'$ of the equivalent node in the output. $F_{j:1}$ contains the mapping from a node $v$ removed due to the first reduction rule and is populated on lines 8 – 14: if both children of $v$ are the same then the mapping $[v.uid \mapsto v.low]$ is pushed to $F_{j:1}$. That is, the node $v$ is not output and is made equivalent to its single child. $F_{j:2}$ contains the mappings for the second rule and is resolved on lines 16 – 25. Here, the remaining nodes are placed in an intermediate file $F_j$ and sorted by their children. This makes duplicate nodes immediate successors in $F_j$. Every new unique node encountered in $F_j$ is written to the output $F_{out}$ before mapping itself and all its duplicates to it in $F_{j:2}$. Since nodes are output out-of-order compared to the input and without knowing how many will be output for said level, they are given new decreasing identifiers starting from the maximal possible value `MAX_ID`. Finally, $F_{j:2}$ is sorted back in order of $F_{internal}$ to forward the results in both $F_{j:1}$ and $F_{j:2}$ to their parents on lines 27 – 33. Here, **MergeMaxUid** merges the mappings $[uid \mapsto uid']$ in $F_{j:1}$ and $F_{j:2}$ by always taking the mapping with the largest $uid$ from either file.

Since the original algorithm of Arge in [6] takes a node-based OBDD as an input and only internally uses node-based auxiliary data structures, his Reduce algorithm had to create two copies of the input to create the transposed subgraph: one where the copies were sorted by the nodes' low child and one where they are sorted by their high child. The reversal of arcs in $F_{internal}$ merges these auxiliary data structures of Arge into a single set of arcs easily generated by the preceding Apply. This not only simplifies the algorithm but also more than halves the memory used and it eliminates two expensive sorting steps. We also apply the first reduction rule before the sorting step, thereby decreasing the number of nodes involved in the remaining expensive computation of that level.

Another consequence of his node-based representation is that his algorithm had to move all arcs to leaves into $Q_{red}$ rather than merging requests from $Q_{red}$ with the base-cases from $F_{leaf}$. Let $N_\ell \leq 2N$ be the number of arcs to leaves then all internal arcs is $2N - N_\ell$, where $N$ is the total number of nodes. Our semi-transposed input considerably decreases the number I/Os used and time spent, which makes it worthwhile in practice.

**Lemma 3.2.** *Use of $F_{leaf}$ saves $\Theta(sort(N_\ell))$ I/Os.*

*Proof.* Without $F_{leaf}$, the number of I/Os spent on elements in $Q_{red}$ is

$$\Theta(\text{sort}(2N)) = \Theta(\text{sort}((2N - N_\ell) + N_\ell)) = \Theta(\text{sort}(2N - N_\ell)) + \Theta(\text{sort}(N_\ell))$$

whereas with $F_{leaf}$ it is $\Theta(\text{sort}(2N - N_\ell)) + N_\ell/B$. It now suffices to focus on the latter half concerning $N_\ell$. The constant $c$ involved in the $\Theta(\text{sort}(N_\ell))$ of

the priority queue must be so large that the following inequality holds for $N_\ell$ greater than some given $n_0$.

$$0 < N_\ell/B \leq c_1 \cdot \text{sort}(N_\ell) \leq c_2 \cdot \text{sort}(N_\ell) \ .$$

The difference $c_1 \cdot \text{sort}(N_\ell/B) - N_\ell/B$ is the least number of saved I/Os, whereas $c_2 \cdot \text{sort}(N_\ell/B) - N_\ell/B$ is the maximal number of saved I/Os. The $O(\text{sort}(N_\ell/B))$ bound follows easily as shown below for $N_\ell > n_0$.

$$c_2 \cdot \text{sort}(N_\ell/B) - N_\ell/B < c_2 \cdot \text{sort}(N_\ell/B)$$

For the $\Omega(\text{sort}(N_\ell/B))$ lower bound, we need to find a $c_1'$ such that $c_1' \cdot \text{sort}(N_\ell) \leq c_1 \cdot \text{sort}(N_\ell/B) - N_\ell/B$ for $N_\ell$ large enough to satisfy the following inequality.

$$c_1' \leq c_1 - \left(\log_{M/B}(N_\ell/B)\right)^{-1}$$

The above is satisfied with $c_1' \triangleq c_1/2$ for $N_\ell > \max(n_0, B(\frac{M}{B})^{2/c_1})$ since then $\left(\log_{M/B}(N_\ell/B)\right)^{-1} < \frac{c_1}{2}$. □

Depending on the given BDD, this decrease can result in an improvement in performance that is notable in practice as our experiments in Section 5.3.3 show. Furthermore, the fraction $N_\ell/(2N)$ only increases when recursion requests are pruned; our experiments in Section 5.3.4 show that pruning increases this fraction by a considerable amount.

**Proposition 3.3** (Following Arge 1996 [6])**.** *The Reduce algorithm in Fig. 9 has an $O(sort(N))$ I/O complexity and an $O(N \log N)$ time complexity.*

*Proof.* Up to $2N$ arcs are inserted in and later again extracted from $Q_{red}$, while $F_{internal}$ and $F_{leaf}$ is scanned once. This totals an $O(\text{sort}(4N) + N/B) = O(\text{sort}(N))$ number of I/Os spent on the priority queue. On each level all nodes are sorted twice, which when all levels are combined amounts to another $O(\text{sort}(N))$ I/Os. One arrives with similar argumentation at the $O(N \log N)$ time complexity. □

Arge proved in [5] that this $O(\text{sort}(N))$ I/O complexity is optimal for the input, assuming a levelwise ordering of nodes (See [6] for the full proof). While the $O(N \log N)$ time is theoretically slower than the $O(N)$ depth-first approach using a unique node table and an $O(1)$ time hash function, one should note the log factor depends on the sorting algorithm and priority queue used where I/O-efficient instances have a large $M/B$ log factor.

## 3.3 Other Algorithms

Arge only considered Apply and Reduce, since most other BDD operations can be derived from these two operations together with the Restrict operation, which fixes the value of a single variable. We have extended the technique to create

a time-forward processed Restrict that operates in $O(\text{sort}(N))$ I/Os, which is described below. Furthermore, by extending the technique to other operations, we obtain more efficient BDD algorithms than by computing it using Apply and Restrict alone.

### 3.3.1 Node and Variable Count

The Reduce algorithm in Section 3.2 can easily be extended to also provide the number of nodes $N$ generated and the number of levels $L$. Using this, it only takes $O(1)$ I/Os to obtain the node count $N$ and the variable count $L$.

**Lemma 3.4.** *The number of nodes $N$ and number of variables $L$ occuring in the BDD for $f$ is obtainable in $O(1)$ I/Os and time.*

### 3.3.2 Negation

A BDD is negated by inverting the value in its nodes' leaf children. This is an $O(1)$ I/O-operation if a *negation flag* is used to mark whether the nodes should be negated on-the-fly as they are read from the stream.

**Proposition 3.5.** *Negation has I/O, space, and time complexity $O(1)$.*

*Proof.* Even though $O(N)$ extra work has to be spent on negating every nodes inside of all other procedures, this extra work can be accounted for in the big-O of the other operations. □

This is a major improvement over the $O(\text{sort}(N))$ I/Os spent by Apply to compute $f \oplus \top$, where $\oplus$ is exclusive disjunction; especially since space can be shared between the BDD for $f$ and $\neg f$.

### 3.3.3 Equality Checking

To check for $f \equiv g$ one has to check the DAG of $f$ being isomorphic to the one for $g$ [10]. This makes $f$ and $g$ trivially inequivalent when at least one of the following three statements are violated.

$$N_f = N_g \qquad L_f = L_g \qquad \forall i : L_{f,i} = L_{g,i} \wedge N_{f,i} = N_{g,i} \ , \tag{2}$$

where $N_f$ and $N_g$ is the number of nodes in the BDD for $f$ and $g$, $L_f$ and $L_g$ the number of levels, $L_{f,i}$ and $L_{g,i}$ the respective labels of the $i$th level, and $N_{f,i}$ and $N_{g,i}$ the number of nodes on the $i$th level. This can respectively be checked in $O(1)$, $O(1)$, and $O(L/B)$ number of I/Os with meta-information easily generated by the Reduce algorithm in Section 3.2.

In what follows, we will address equality checking the non-trivial cases. Assuming the constraints in (2) we will omit $f$ and the $g$ in the subscript and just write $N$, $L$, $L_i$, and $N_i$.

**An $O(\mathbf{sort}(N))$ equality check.** If $f \equiv g$, then the isomorphism relates the roots of the BDDs for $f$ and $g$. Furthermore, for any node $v_f$ of $f$ and $v_g$ of $g$, if $(v_f, v_g)$ is uniquely related by the isomorphism then so should $(v_f.low, v_g.low)$ and $(v_f.high, v_g.high)$. Hence, the Apply algorithm can be adapted to not output any arcs. Instead, it returns $\bot$ and terminates early when one of the following two conditions are violated.

- The children of the given recursion request $(t_f, t_g)$ should both be a leaf or an internal node. Furthermore, pairs of internal nodes should agree on the label while pairs of leaves should agree on the value.
- On level $i$, exactly $N_i$ unique recursion requests should be retrieved from the priority queues. If more than $N_i$ are given, then prior recursions have related one node of $f$, resp. $g$, to two different nodes in $g$, resp. $f$. This implies that $G_f$ and $G_g$ cannot be isomorphic.

If no recursion requests fail, then the first condition of the two guarantees that $f \equiv g$ and so $\top$ is returned. The second condition makes the algorithm terminate earlier on negative cases and lowers the provable complexity bound.

**Proposition 3.6.** *If the BDDs for $f$ and $g$ satisfy the constraints in (2), then the above equality checking procedure has I/O complexity $O(sort(N))$ and time complexity $O(N \cdot \log N)$, where $N$ is the size of both BDDs.*

*Proof.* Due to the second termination case, no more than $N_i$ unique requests are resolved for each level $i$ in the BDDs for $f$ and $g$. This totals at most $N$ requests are processed by the above procedure regardless of whether $f \equiv g$ or not. This bounds the number of elements placed in the priority queues to be $2N$. The two complexity bounds then directly follow by similar analysis as given in the proof of Proposition 3.1 □

This is an asymptotic improvement on the $O(\mathrm{sort}(N^2))$ equality checking algorithm by computing $f \leftrightarrow g$ with the Apply procedure and then checking whether the reduced BDD is the $\top$ leaf. Furthermore, it is also a major improvement in the practical efficiency, since it does not need to run the Reduce algorithm, $s$ and *is_high* in the $s \xrightarrow{is\_high} (t_f, t_g)$ recursion requests are irrelevant and memory is saved by omitting them, it has no $O(N^2)$ intermediate output, and it only needs a single request for each $(t_f, t_g)$ to be moved into the second priority queue (since $s$ and *is_high* are omitted). In practice, it performs even better since it terminates on the first violation.

**A $2 \cdot N/B$ equality check.** One can achieve an even faster equality checking procedure, by strengthening the constraints on the node ordering. On top of the ordering in (1) on page 7, we require leaves to come in-order after internal nodes:

$$\forall x_{i,id} \ : \ x_{i,id} < \bot < \top \ , \tag{3}$$

and we add the following constraint onto internal nodes $(x_{i,id_1}, low_1, high_1)$ and $(x_{i,id_2}, low_2, high_2)$ on level $i$:

$$id_1 < id_2 \equiv low_1 < low_2 \vee (low_1 = low_2 \wedge high_1 < high_2) \tag{4}$$

This makes the ordering in (1) into a lexicographical three-dimensional ordering of nodes based on their label and their children. The key idea here is, since the second reduction rule of Bryant [10] removes any duplicate nodes then the *low* and *high* children directly impose a total order on their parents.

We will say that the identifiers are *compact* if all nodes on level $i$ have identifiers within the interval $[0, N_i)$. That is, the $j$th node on the $i$th level has the identifier $j - 1$.

**Proposition 3.7.** *Let $G_f$ and $G_g$ be two ROBDDs with compact identifiers that respect the ordering of nodes in (1) extended with (3,4). Then, $f \equiv g$ if and only if for all levels $i$ the $j$th node on level $i$ in $G_f$ and $G_g$ match numerically.*

*Proof.* Suppose that all nodes in $G_f$ and $G_g$ match. This means they describe the same DAG, are hence trivially isomorphic, and so $f \equiv g$.

Now suppose that $f \equiv g$, which means that $G_f$ and $G_g$ are isomorphic. We will prove the stronger statement that for all levels $i$ the $j$th node on the $i$th level in $G_f$ and $G_g$ not only match numerically but are also the root to the very same subgraphs. We will prove this by strong induction on levels $i$, starting from the deepest level $\ell_{\max}$ up to the root with label $\ell_{\min}$.

For $i = \ell_{\max}$, since an ROBDD does not contain duplicate nodes, there exists only one or two nodes in $G_f$ and $G_g$ on this level: one for $x_{\ell_{\max}}$ and/or one describing $\neg x_{\ell_{\max}}$. Since $\bot < \top$, the extended ordering in (4) gives us that the node describing $x_{\ell_{\max}}$ comes before $\neg x_{\ell_{\max}}$. The identifiers must match due to compactness.

For $\ell_{\min} \leq i < \ell_{\max}$, assume per induction that for all levels $i' > i$ the $j'$th node on level $i'$ describe the same subgraphs. Let $v_{f,1}, v_{f,2}, \ldots, v_{f,N_i}$ and $v_{g,1}, v_{g,2}, \ldots, v_{g,N_i}$ be the $N_i$ nodes in order on the $i$th level of $G_f$ and $G_g$. For every $j$, the isomorphism between $G_f$ and $G_g$ provides a unique $j'$ s.t. $v_{f,j}$ and $v_{g,j'}$ are the respective equivalent nodes. From the induction hypotheses we have that $v_{f,j}.low = v_{g,j'}.low$ and $v_{f,j}.high = v_{g,j'}.high$. We are left to show that their identifiers match and that $j = j'$, which due to compactness are equivalent statements. The non-existence of duplicate nodes guarantees that all other nodes on level $i$ in both $G_f$ and $G_g$ differ in either their low and/or high child. The total ordering in (4) is based on these children alone and so from the induction hypothesis the number of predecessors, resp. successors, to $v_{f,j}$ must be the same as for $v_{g,j'}$. That is, $j$ and $j'$ are the same. □

The Reduce algorithm in Figure 9 already outputs the nodes on each level in the order that satisfies the constraint (3) and (4). It also outputs the $j$th node of each level with the identifier $\texttt{MAX\_ID} - N_i + j$, i.e. with *compact* identifiers shifted by $\texttt{MAX\_ID} - N_i$. Hence, Proposition 3.7 applies to the ROBDDs constructed by this Reduce and they can be compared in $2 \cdot N/B$ I/Os with a single iteration

through all nodes. This iteration fails at the same pair of nodes, or even earlier, than the prior $O(\text{sort}(N))$ algorithm. Furthermore, this linear scan of both BDDs is optimal – even with respect to the constant – since any deterministic comparison procedure has to look at every element of both inputs at least once.

As is apparent in the proof of Proposition 3.7, the ordering of internal nodes relies on the ordering of leaf values. Hence, the negation algorithm in Section 3.3.2 breaks this property when the negation flags on $G_f$ and $G_g$ do not match.

**Corollary 3.8.** *If $G_f$ and $G_g$ are outputs of Fig. 9 and their negation flag are equal, then equality checking of $f \equiv g$ can be done in $2 \cdot N/B$ I/Os and $O(N)$ time, where $N$ is the size of $G_f$ and $G_g$.*

One could use the Reduce algorithm to make the negated BDD satisfy the preconditions of Proposition 3.7. However, the $O(\text{sort}(N))$ equality check will be faster in practice, due to its efficiency in both time, I/O, and space.

### 3.3.4 Path and Satisfiability Count

The number of paths that lead from the root of a BDD to the $\top$ leaf can be computed with a single priority queue that forwards the number of paths from the root to a node $t$ through one of its ingoing arcs. At $t$ these ingoing paths are accumulated, and if $t$ has a $\top$ leaf as a child then the number of paths to $t$ added to a total sum of paths. The number of ingoing paths to $t$ is then forwarded to its non-leaf children.

This easily extends to counting the number of satisfiable assignments by knowing the number of variables of the domain and extending the request in the priority queue with the number of levels visited.

**Proposition 3.9.** *Counting the number of paths and satisfying assignments in the BDD for $f$ has I/O complexity $O(sort(N))$ and time complexity $O(N \log N)$, where $N$ is the size of the BDD for $f$.*

*Proof.* Every node of $f$ is processed in-order and all $2N$ arcs are inserted and extracted once from the priority queue. This totals $O(\text{sort}(N)) + N/B$ I/Os.

For all $N$ nodes there spent is $O(1)$ time on every in-going arc. There are a total of $2N$ arcs, so $O(N)$ amount of work is done next to reading the input and using the priority queue. Hence, the $O(N \log N)$ time spent on the priority queue dominates the asymptotic running time. □

Both of these operations not possible to compute with the Apply operation.

### 3.3.5 Evaluating and Obtaining Assignments

Evaluating the BDD for $f$ given some $\vec{x}$ boils down to following a path from the root down to a leaf based on $\vec{x}$. The levelized ordering makes it possible to follow a path starting from the root in a single scan of all nodes, where irrelevant nodes are ignored.

**Lemma 3.10.** *Evaluating $f$ for some assignment $\vec{x}$ has I/O complexity $N/B + \vec{x}/B$ and time complexity $O(N + |\vec{x}|)$, where $N$ is the size of $f$'s BDD.*

Similarly, finding the lexicographical smallest or largest $\vec{x}$ that makes $f(\vec{x}) = \top$ also corresponds to providing the trace of one specific path in the BDD [28].

**Lemma 3.11.** *Obtaining the lexicographically smallest (or largest) assignment $\vec{x}$ such that $f(\vec{x}) = \top$ has I/O complexity $N/B + \vec{x}/B$ and time complexity $O(N)$, where $N$ is the size of the BDD for $f$.*

If the number of Levels $L$ is smaller than $N/B$ then this uses more I/Os compared to the conventional algorithms that uses random access; when jumping from node to node at most one node on each of the $L$ levels is visited, and so at most one block for each level is retrieved.

### 3.3.6 If-Then-Else

The If-Then-Else procedure is an extension of the idea of the Apply procedure where requests are triples $(t_f, t_g, t_h)$ rather than tuples and three priority queues are used rather than two. The idea of pruning in Section 3.1.1 can also be applied here: if $t_f \in \{\bot, \top\}$ then either $t_g$ or $t_h$ is irrelevant and it can be replaced with `Nil` such that these requests are merged into one node, and if $t_g = t_h \in \{\bot, \top\}$ then $t_f$ does not need to be evaluated and the leaf can immediately be output.

**Proposition 3.12.** *The If-Then-Else procedure has I/O complexity $O(sort(N_f \cdot N_g \cdot N_h))$ and time complexity $O((N_f \cdot N_g \cdot N_h)\log(N_f \cdot N_g \cdot N_h))$, where $N_f$, $N_g$, and $N_h$ are the respective sizes of the BDDs for $f$, $g$, and $h$.*

*Proof.* The proof follows by similar argumentation as for Proposition 3.1. □

This is an $O(N_f)$ factor improvement over the $O(\mathrm{sort}(N_f^2 \cdot N_g \cdot N_h))$ I/O complexity of using Apply to compute $(f \wedge g) \vee (\neg f \wedge h)$.

### 3.3.7 Restrict

Given a BDD $f$, a label $i$, and a boolean value $b$ the function $f|_{x_i=b}$ can be computed in a single top-down sweep. The priority queue contains arcs from a source $s$ to a target $t$. When encountering the target node $t$ in $f$, if $t$ does not have label $i$ then the node is kept and the arc $s \xrightarrow{is\_high} t$ is output, as is. If $t$ has label $i$ then the arc $s \xrightarrow{is\_high} t.low$ is forwarded in the queue if $b = \bot$ and $s \xrightarrow{is\_high} t.high$ is forwarded if $b = \top$.

This algorithm has one problem with the pipeline in Fig. 4. Since nodes are skipped, and the source $s$ is forwarded, then the leaf arcs may end up being produced and output out of order. In those cases, the output arcs in $F_{leaf}$ need to be sorted before the following Reduce can begin.

**Proposition 3.13.** *Computing the Restrict of $f$ for a single variable $x_i$ has I/O complexity $O(sort(N))$ and time complexity $O(N \log N)$, where $N$, is the size of the BDD for $f$.*

*Proof.* The priority queue requires $O(\text{sort}(2N))$ I/Os since every arc in the priority queue is related one-to-one with one of the $2N$ arcs of the input. The output consists of at most $2N$ arcs, which takes $2 \cdot N/B$ I/Os to write and at most $O(\text{sort}(2N))$ to sort. All in all, the algorithm uses $N/B + O(\text{sort}(2N)) + 2 \cdot N/B + O(\text{sort}(2N))$ I/Os. □

This trivially extends to multiple variables in a single sweep by being given the assignments $\vec{x}$ in ascending order relative to the labels $i$.

**Corollary 3.14.** *Computing the Restrict of $f$ for multiple variables $\vec{x}$ has I/O complexity $O(sort(N) + |\vec{x}|/B)$ and time complexity $O(N \log N + |\vec{x}|)$, where $N$, is the size of the BDD for $f$.*

*Proof.* The only extra work added is reading $\vec{x}$ in-order and to decide for all $L \leq N$ levels whether to keep the level's nodes or bridge over them. This constitutes another $|\vec{x}|/B$ I/Os and $O(N)$ time. □

### 3.3.8 Quantification

Given $Q \in \{\forall, \exists\}$, a BDD $f$, and a label $i$ the BDD representing $Qb \in \mathbb{B} : f|_{x_i=b}$ is computable using $O(\text{sort}(N^2))$ I/Os by combining the ideas for the Apply and the Restrict algorithms.

Requests are either a single node $t$ in $f$ or past level $i$ a tuple of nodes $(t_1, t_2)$ in $f$. A priority queue initially contains requests from the root of $f$ to its children. Two priority queues are used to synchronise the tuple requests with the traversal of nodes in $f$. If the label of a request $t$ is $i$, then, similar to Restrict, its source $s$ is linked with a single request to $(t.low, t.high)$. Tuples are handled as in Apply where $\odot$ is $\vee$, resp. $\odot$ is $\wedge$, for $Q = \exists$, resp. for $Q = \forall$.

Both conjunction and disjunction are commutative operations, so the order of $t_1$ and $t_2$ is not relevant. Hence, the tuple $(t_1, t_2)$ can be kept sorted such that $t_1 < t_2$. This improves the performance of the comparator used within the priority queues since $\min(t_1, t_2) = t_1$ and $\max(t_1, t_2) = t_2$.

The idea of pruning in Section 3.1.1 also applies here. But, in the following cases can a request to a tuple be turned back into a request to a single node $t$.

- Any request $(t_1, t_2)$ where $t_1 = t_2$ is equivalent to the request to $t_1$.
- As we only consider operators $\vee$ and $\wedge$ then any leaf $v \in \{\bot, \top\}$ that does not short circuit $\odot$ is irrelevant for the leaf values of the entire subtree. Hence, any such request $(t, v)$ can safely be turned into a request to $t$.

This collapses three potential subtrees into one, which decreases the size of the output and the number of elements placed in the priority queues.

**Proposition 3.15.** *Computing the Exists and Forall of $f$ for a single variable $x_i$ has I/O complexity $O(sort(N^2))$ and time complexity $O(N^2 \log N^2)$, where $N$ is the size of the BDD for $f$.*

*Proof.* A request for a node $t$ is equivalent to a request to $(t, t)$. There are at most $N^2$ pairs to process which results in at most $2 \cdot N^2$ number of arcs in the output and in the priority queues. The two complexity bounds follows by similar analysis as in the proof of Proposition 3.1 and 3.13. □

This could also be computed with Apply and Restrict as $f|_{x_i=\bot} \odot f|_{x_i=\top}$, which would also only take an $O(\mathrm{sort}(N)+\mathrm{sort}(N)+\mathrm{sort}(N^2))$ number of I/Os, but this requires three separate runs of Reduce and makes the intermediate inputs to Apply of up to $2N$ in size.

### 3.3.9 Composition

The composition $f|_{x_i=g}$ of two BDDs $f$ and $g$ can be computed by reusing the ideas and optimisations from the Quantification and the If-Then-Else procedures. Requests start out as tuples $(t_g, t_f)$ and at level $i$ they are turned into triples $(t_g, t_{f_1}, t_{f_2})$ which are handled similar to the If-Then-Else. The idea of merging recursion requests based on the value of $t_{f_1}$ and $t_{f_2}$ as for Quantification still partially applies here: if $t_{f_1} = t_{f_2}$ or $t_g$ is a leaf, then the entire triple can boiled down to $t_{f_1}$ or $t_{f_2}$.

**Proposition 3.16.** *Composition has I/O complexity $O(sort(N_f^2 \cdot N_g))$ and time complexity $O(N_f^2 \cdot N_g)\log(N_f \cdot N_g))$, where $N_f$ and $N_g$ are the respective sizes of the BDDs for $f$ and $g$.*

*Proof.* There are $N_f^2 \cdot N_g$ possible triples, each of which can be in the output and in the priority queue. The two complexity bounds follows by similar analysis as in the proof of Proposition 3.12. □

Alternatively, this is computable with the If-Then-Else $g \ ? \ f|_{x_i=\top} \ : \ f|_{x_i=\bot}$ which would also take $O(2 \cdot \mathrm{sort}(N_f) + \mathrm{sort}(N_f^2 \cdot N_g))$ number of I/Os, or $O(\mathrm{sort}(N_f^2 \cdot N_g^2))$ I/Os if the Apply is used. But as argued above, computing the If-Then-Else with Apply is not optimal. Furthermore, similar to quantification, the constants involved in using Restrict for intermediate outputs is also costly.

## 3.4 Levelized Priority Queue

In practice, sorting a set of elements with a sorting algorithm is considerably faster than with a priority queue [36]. Hence, the more one can use mere sorting instead of a priority queue the faster the algorithms will run.

Since all our algorithms resolve the requests level by level then any request pushed to the priority queues $Q_{app:1}$ and $Q_{red}$ are for nodes on a yet-not-visited level. That means, any requests pushed when resolving level $i$ do not change the order of any elements still in the priority queue with label $i$. Hence, for $L$ being the number of levels and $L < M/B$ one can have a *levelized priority queue*[2] by

[2]The name is chosen as a reference to the independently conceived but reminiscent queue of Ashar and Cheong [7]. It is important to notice that they split the queue for functional correctness whereas we do so to improve the performance.

maintaining $L$ buckets and keep one block for each bucket in memory. If a block is full then it is sorted, flushed from memory and replaced with a new empty block. All blocks for level $i$ are merge-sorted into the final order of requests when the algorithm reaches the $i$th level.

While $L < M/B$ is a reasonable assumption between main memory and disk it is not at the cache-level. Yet, most arcs only span very few levels, so it suffices for the levelized priority queue to only have a small preset $k$ number of buckets and then use an *overflow* priority queue to sort the few requests that go more than $k$ levels ahead. A single block of each bucket fits into the cache for very small choices of $k$, so a cache-oblivious levelized priority queue would not lose its characteristics. Simultaneously, smaller $k$ allows one to dedicate more internal memory to each individual bucket, which allows a cache-aware levelized priority queue to further improve its performance.

## 3.5 Memory Layout and Efficient Sorting

A unique identifier can be represented as a single 64-bit integer as shown in Fig. 10. Leaves are represented with a 1-flag on the most significant bit, its value $v$ on the next 62 bits and a boolean flag $f$ on the least significant bit. A node is identified with a 0-flag on the most significant bit, the next $\ell$-bits dedicated to its label followed by $62 - \ell$ bits with its identifier, and finally the boolean flag $f$ available on the least-significant bit.


(a) Unique identifier of a leaf v (b) Unique identifier of an internal node

Figure 10: Bit-representations. The least significant bit is right-most.

A node is represented with 3 64-bit integers: two of them for its children and the third for its label and identifier. An arc is represented by 2 64-bit numbers: the source and target each occupying 64-bits and the *is_high* flag stored in the $f$ flag of the source. This reduces all the prior orderings to a mere trivial ordering of 64-bit integers. A descending order is used for a bottom-up traversal while the sorting is in ascending order for the top-down algorithms.

A reasonable value for the number $\ell$ bits dedicated to the label is 24. Here there are still $2^{62-\ell} = 2^{38}$ number of nodes available per level, which is equivalent to 6 TiB of data in itself.

# 4 Adiar: An Implementation

The algorithms and data structures described in Section 3 have been implemented in a new BDD package, named Adiar[3], that supports the operations in Table 1. The source code for Adiar is publicly available at

[3] **adiar** ⟨ portuguese ⟩ (*verb*) : to defer, to postpone

| Adiar function | Operation | I/O complexity | Justification |
|---|---|---|---|
| BDD Base constructors | | | |
| `bdd_sink(b)` | $b$ | $O(1)$ | |
| `bdd_true()` | $\top$ | | |
| `bdd_false()` | $\bot$ | | |
| `bdd_ithvar(`$i$`)` | $x_i$ | $O(1)$ | |
| `bdd_nithvar(`$i$`)` | $\neg x_i$ | $O(1)$ | |
| `bdd_and(`$\vec{i}$`)` | $\bigvee_{i \in \vec{i}} x_i$ | $O(k/B)$ | |
| `bdd_or(`$\vec{i}$`)` | $\bigwedge_{i \in \vec{i}} x_i$ | $O(k/B)$ | |
| `bdd_counter(`$i,j,t$`)` | $\#^{j}_{k=i} x_k = t$ | $O((i-j) \cdot t/B)$ | |
| BDD Manipulation | | | |
| `bdd_apply(`$f,g,\odot$`)` | $f \odot g$ | $O(\mathrm{sort}(N_f \cdot N_g))$ | Prop. 3.1, 3.3 |
| `bdd_and(`$f,g$`)` | $f \wedge g$ | | |
| `bdd_nand(`$f,g$`)` | $\neg(f \wedge g)$ | | |
| `bdd_or(`$f,g$`)` | $f \vee g$ | | |
| `bdd_nor(`$f,g$`)` | $\neg(f \vee g)$ | | |
| `bdd_xor(`$f,g$`)` | $f \oplus g$ | | |
| `bdd_xnor(`$f,g$`)` | $\neg(f \oplus g)$ | | |
| `bdd_imp(`$f,g$`)` | $f \rightarrow g$ | | |
| `bdd_invimp(`$f,g$`)` | $f \leftarrow g$ | | |
| `bdd_equiv(`$f,g$`)` | $f \equiv g$ | | |
| `bdd_diff(`$f,g$`)` | $f \wedge \neg g$ | | |
| `bdd_less(`$f,g$`)` | $\neg f \wedge g$ | | |
| `bdd_ite(`$f,g,h$`)` | $f \ ?\ g\ :\ h$ | $O(\mathrm{sort}(N_f \cdot N_g \cdot N_h))$ | Prop. 3.12, 3.3 |
| `bdd_restrict(`$f,i,v$`)` | $f\vert_{x_i=v}$ | $O(\mathrm{sort}(N_f))$ | Prop. 3.13, 3.3 |
| `bdd_restrict(`$f,\vec{x}$`)` | $f\vert_{(i,v)\in\vec{x}\ .\ x_i=v}$ | $O(\mathrm{sort}(N_f) + \vert\vec{x}\vert/B)$ | Prop. 3.14, 3.3 |
| `bdd_exists(`$f,i$`)` | $\exists v : f\vert_{x_i=v}$ | $O(\mathrm{sort}(N_f^2))$ | Prop. 3.15, 3.3 |
| `bdd_forall(`$f,i$`)` | $\forall v : f\vert_{x_i=v}$ | $O(\mathrm{sort}(N_f^2))$ | Prop. 3.15, 3.3 |
| `bdd_not(`$f$`)` | $\neg f$ | $O(1)$ | Prop. 3.5 |
| Counting | | | |
| `bdd_pathcount(`$f$`)` | #paths in $f$ to $\top$ | $O(\mathrm{sort}(N_f))$ | Prop. 3.9 |
| `bdd_satcount(`$f$`)` | $\#\vec{x} : f(\vec{x})$ | $O(\mathrm{sort}(N_f))$ | Prop. 3.9 |
| `bdd_nodecount(`$f$`)` | $N_f$ | $O(1)$ | Lem. 3.4 |
| `bdd_varcount(`$f$`)` | $L_f$ | $O(1)$ | Lem. 3.4 |
| Equivalence Checking | | | |
| $f$ `==` $g$ | $f \equiv g$ | $O(\mathrm{sort}(\min(N_f, N_g)))$ | Prop. 3.6 |
| BDD traversal | | | |
| `bdd_eval(`$f,\vec{x}$`)` | $f(\vec{x})$ | $O((N_f + \vert\vec{x}\vert)/B)$ | Lem. 3.10 |
| `bdd_satmin(`$f$`)` | $\min\{\vec{x} \mid f(\vec{x})\}$ | $O((N_f + \vert\vec{x}\vert)/B)$ | Lem. 3.11 |
| `bdd_satmax(`$f$`)` | $\max\{\vec{x} \mid f(\vec{x})\}$ | $O((N_f + \vert\vec{x}\vert)/B)$ | Lem. 3.11 |

Table 1: Supported operations in Adiar together with their I/O-complexity. $N$ is the number of nodes and $L$ is the number of levels in a BDD. An assignment $\vec{x}$ is a list of tuples $(i, v) : \mathbb{N} \times \mathbb{B}$.

github.com/ssoelvsten/adiar

and the documentation is available at

ssoelvsten.github.io/adiar/ .

Interaction with the BDD package is done through C++ programs that include the `<adiar/adiar.h>` header file and are built and linked with CMake. Its two dependencies are the Boost library and the TPIE library; the latter is included as a submodule of the Adiar repository, leaving it to CMake to build TPIE and link it to Adiar.

The BDD package is initialised by calling the `adiar_init(memory, temp_dir)` function, where `memory` is the memory (in bytes) dedicated to Adiar and `temp_dir` is the directory where temporary files will be placed, which could be a dedicated harddisk. After use, the BDD package is deinitialised and its given memory is freed by calling the `adiar_deinit()` function.

The `bdd` object in Adiar is a container for the underlying files to represent each BDD. A `__bdd` object is used for possibly unreduced arc-based OBDD outputs. Both objects use reference counting on the underlying files to both reuse the same files and to immediately delete them when the reference count decrements 0. By use of implicit conversions between the `bdd` and `__bdd` objects and an overloaded assignment operator, this garbage collection happens as early as possible, making the number of concurrent files on disk minimal.

A BDD can also be constructed explicitly with the `node_writer` object in $O(N/B)$ I/Os by supplying it all nodes in reverse of the ordering in (1).

# 5 Experimental Evaluation

We assert the viability of our techniques by investigating the following questions.

1. How well does our technique perform on BDDs larger than main memory?
2. How big an impact do the optimisations we propose have on the computation time and memory usage?
   (a) Use of the levelized priority queue.
   (b) Separating the node-to-leaf arcs of Reduce from the priority queue.
   (c) Pruning the output of Apply by short circuiting the given operator.
   (d) Use of our improved equality checking algorithms.
3. How well do our algorithms perform in comparison to conventional BDD libraries that use depth-first recursion, a unique table, and caching?

## 5.1 Benchmarks

To evaluate the performance of our proposed algorithms we have created implemented multiple benchmarks for Adiar and other BDD packages, where the BDDs are constructed in a similar manner and the same variable ordering is used. The source code for all benchmarks is publicly available at the following url:

github.com/ssoelvsten/bdd-benchmark

The raw data and data analysis has been made available at [41].

**N-Queens.** The solution to the $N$-Queens problem is the number of arrangements of $N$ queens on an $N \times N$ board, such that no queen is threatened by another. Our benchmark follows the description in [29]: the variable $x_{ij}$ represents whether a queen is placed on the $i$th row and the $j$th column and so the solution corresponds to the number of satisfying assignments of the formula

$$\bigwedge_{i=0}^{N-1} \bigvee_{j=0}^{N-1} (x_{ij} \wedge \neg \mathit{has\_threat}(i,j)) \quad ,$$

where $\mathit{has\_threat}(i,j)$ is true, if a queen is placed on a tile $(k,l)$, that would be in conflict with a queen placed on $(i,j)$. The ROBDD of the innermost conjunction can be directly constructed, without any BDD operations.

**Tic-Tac-Toe.** In this problem from [29] one must compute the number of possible draws in a $4 \times 4 \times 4$ game of Tic-Tac-Toe, where only $N$ crosses are placed and all other spaces are filled with naughts. This amounts to counting the number of satisfying assignments of the following formula.

$$\mathit{init}(N) \wedge \bigwedge_{(i,j,k,l) \in L} ((x_i \vee x_j \vee x_k \vee x_l) \wedge (\overline{x_i} \vee \overline{x_j} \vee \overline{x_k} \vee \overline{x_l})) \quad ,$$

where $\mathit{init}(N)$ is true iff exactly $N$ out of the 76 variables are true, and $L$ is the set of all 76 lines through the $4 \times 4 \times 4$ cube. To minimise the time and space to solve, lines in $L$ are accumulated in increasing order of the number of levels they span. The ROBDDs for both $\mathit{init}(N)$ and the 76 line formulas can be directly constructed without any BDD operations. Hence, this benchmark always consists of 76 uses of Apply to accumulate the line constraints onto $\mathit{init}(N)$.

**Picotrav.** The *EPFL* Combinational Benchmark Suite [2] consists of 23 combinatorial circuits designed for logic optimisation and synthesis. 20 of these are split into the two categories *random/control* and *arithmetic*, and each of these original circuits $C_o$ are distributed together with one optimised for size $C_s$ and one for optimised for depth $C_d$. The last three benchmarks are the *More than a Million Gates* benchmarks, which we will ignore as they come

without optimised versions. They also are generated randomly and hence they are unrealistic anyway.

We have recreated a subset of the *Nanotrav* BDD application as distributed with CUDD [43]. Here, we verify the functional equivalence between each output gate of the original circuit $C_o$ and the corresponding gate of an optimised circuit $C_d$ or $C_s$. Every input gate is represented by a decision variable and recursively the BDD representing each gate is computed. Memoisation ensures the same gate is not computed twice, while a reference counter is maintained for each gate to clear the memoisation table; this allows for garbage collection of intermediate BDDs. Finally, every pair of BDDs that should represent the same output are tested for equality.

The variable order used was chosen based on what produced the smallest BDDs during our initial experiments. We had the *random/control* benchmarks use the order in which the inputs were declared while the *arithmetic* benchmarks derived an ordering based on the deepest reference within the optimised circuit to their respective input gate; ties for the same level are resolved by a DFS traversal of the same circuit.

## 5.2 Hardware

We have run experiments on the following two very different kinds of machines.

- *Consumer grade laptop* with one quadro-core 2.6 GHz Intel i7-4720HQ processor, 8 GiB of RAM, 230 GiB of available SSD disk, running Ubuntu, and compiling code with GCC 9.3.0.
- *Grendel server node* with two 48-core 3.0 GHz Intel Xeon Gold 6248R processors, 384 GiB of RAM, 3.5 TiB of available SSD disk, running CentOS Linux, and compiling code with GCC 10.1.0.

The former, due to its limited RAM, has until now been incapable of manipulating larger BDDs. The latter, on the other hand, has enough RAM and disk to allow all BDD implementations to solve large instances on comparable hardware.

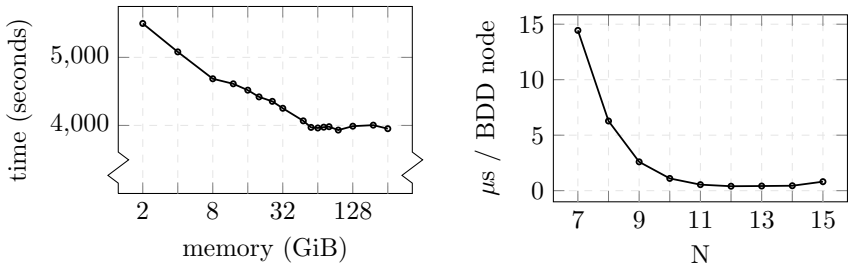
## 5.3 Experimental Results

All but the largest benchmarks were run multiple times and the *minimum* measured running time is reported, since it minimises any error caused by slowdown and overhead on the CPU and the memory [16]. Using the *average* or *median* value instead has only a negligible impact on the resulting numbers.

### 5.3.1 Research Question 1:

Fig. 11a shows the 15-Queens problem being solved with Adiar on the Grendel server node with variable amounts of available main memory. *cgroups* are used to enforce the machines memory limit, including its file system cache, to be only

a single GiB more than what is given to Adiar. During these computations, the largest reduced BDD is 19.4 GiB which makes its unreduced input at least 25.9 GiB in size. That is, the input and output of the largest run of Reduce alone occupies at least 45.3 GiB. Yet, we only see a 39.1% performance decrease when decreasing the available memory from 256 GiB to 2 GiB. This change in performance primarily occurs in the 2 GiB to 64 GiB interval where data needs to be fetched from disk; more than half of this decrease is in the 2 GiB to 12 GiB interval.




(a) Computing 15-Queens with different amount of memory available.

(b) Computing $N$-Queens solutions with 7 GiB of memory available.

Figure 11: Performance of Adiar in relation to available memory.

To confirm the seamless transition to disk, we investigate different $N$, fix the memory, and normalise the data to be $\mu$s per node. The current version of Adiar is implemented purely using the external memory algorithms of TPIE. These perform poorly when given small amounts of data; the time it takes to initialise the larger memory makes it by several orders of magnitude slower than if it used equivalent internal memory algorithms. This overhead is apparent for $N \leq 11$.

The consumer grade laptop's memory cannot contain the 19.4 GiB BDD and its SSD the total of 250.3 GiB of data generated by the 15-Queens problem. Yet, as Fig. 11b shows for $N = 7, \ldots, 15$ the computation time per node only slows down by a factor of 1.8 when crossing the memory barrier from $N = 14$ to 15. Furthermore, this is only a slowdown at $N = 15$ by a factor of 2.02 compared to the lowest recorded computation time per node at $N = 12$.

### 5.3.2 Research Question 2a:

Table 2 shows the average running time of the $N$-Queens problem for $N = 14$, resp. the Tic-Tac-Toe problem for $N = 22$, when using the levelized priority queue compared to the priority queue of TPIE.

Performance increases by 25.3% for the Queens and by 37.0% for the Tic-Tac-Toe benchmark when switching from the TPIE priority queue to the levelized priority queue with a single bucket. The BDDs generated in either benchmark have very few (if any) arcs going across more than a single level, which explains the lack of any performance increase past the first bucket.

| Priority Queue | Time (s) |
|---|---|
| TPIE | 908 |
| L-PQ(1) | 678 |
| L-PQ(4) | 677 |

(a) N-Queens ($N = 14$)

| Priority Queue | Time (s) |
|---|---|
| TPIE | 1003 |
| L-PQ(1) | 632 |
| L-PQ(4) | 675 |

(b) Tic-Tac-Toe ($N = 22$)

Table 2: Performance increase by use of the levelized priority queue with $k$ buckets (L-PQ($k$)) compared to the priority queue of TPIE.

#### 5.3.3 Research Question 2b:

To answer this question, we move the contents of $F_{leaf}$ into $Q_{red}$ at the start of the Reduce algorithm. This is not possible with the levelized priority queue, so the experiment is conducted on the consumer grade laptop with 7 GiB of ram dedicated to an older version of Adiar with the priority queue of TPIE. The node-to-leaf arcs make up 47.5% percent of all arcs generated in the 14-Queens benchmark. The average time to solve this goes down from 1258 to 919 seconds. This is an improvement of 27.0% which is 0.57% for every percentile the node-to-leaf arcs contribute to the total number of arcs generated. Compensating for the performance increase in Research Question 2a this only amounts to 20.12%, i.e. 0.42% per percentile.

#### 5.3.4 Research Question 2c:

Like in Research Question 2b, it is to be expected that this optimisation is dependant on the input. Table 3 shows different benchmarks run on the consumer grade laptop with and without pruning.

For $N$-Queens the largest unreduced BDD decreases in size by at least 23% while the median is unaffected. The $x_{ij} \wedge \neg \mathit{has\_threat}(i, j)$ base case consists mostly of $\bot$ leaves, so they can prune the outermost conjunction of rows but not the disjunction within each row. The relative number of node-to-leaf arcs is also at least doubled, which decreases the number of elements placed in the priority queues. This, together with the decrease in the largest unreduced BDD, explains how the computation time decreases by 49% for $N = 15$. Considering our results for Research Question 2b above at most half of that speedup can be attributed to increasing the percentage of node-to-leaf arcs.

We observe the opposite for the Tic-Tac-Toe benchmark. The largest unreduced size is unaffected while the median size decreases by at least 20%. Here, the BDD for each line has only two arcs to the $\bot$ leaf that can short circuit the accumulated conjunction, while the largest unreduced BDDs are created when accumulating the very last few lines, that span almost all levels. Still, there is a doubling in the total ratio of node-to-leaf arcs and we see at least an 11.6% decrease in the computation time.

| N | | | |
|---|---|---|---|
| 12 | 13 | 14 | 15 |
| Time to solve (s) | | | |
| $3.70 \cdot 10^1$ | $2.09 \cdot 10^2$ | $1.35 \cdot 10^3$ | $1.21 \cdot 10^4$ |
| $2.25 \cdot 10^1$ | $1.23 \cdot 10^2$ | $7.38 \cdot 10^2$ | $6.34 \cdot 10^3$ |
| Largest unreduced size (#nodes) | | | |
| $9.97 \cdot 10^6$ | $5.26 \cdot 10^7$ | $2.76 \cdot 10^8$ | $1.70 \cdot 10^9$ |
| $7.33 \cdot 10^6$ | $3.93 \cdot 10^7$ | $2.10 \cdot 10^8$ | $1.29 \cdot 10^9$ |
| Median unreduced size (#nodes) | | | |
| $2.47 \cdot 10^3$ | $3.92 \cdot 10^3$ | $6.52 \cdot 10^3$ | $1.00 \cdot 10^4$ |
| $2.47 \cdot 10^3$ | $3.92 \cdot 10^3$ | $6.52 \cdot 10^3$ | $1.00 \cdot 10^4$ |
| Node-to-leaf arcs ratio | | | |
| 16.9% | 17.4% | 17.6% | 17.4% |
| 43.9% | 46.2% | 47.5% | 48.1% |

(a) N-Queens

| N | | | |
|---|---|---|---|
| 20 | 21 | 22 | 23 |
| Time to solve (s) | | | |
| $2.80 \cdot 10^1$ | $1.57 \cdot 10^2$ | $7.99 \cdot 10^2$ | $1.20 \cdot 10^4$ |
| $2.38 \cdot 10^1$ | $1.39 \cdot 10^2$ | $6.96 \cdot 10^2$ | $8.91 \cdot 10^3$ |
| Largest unreduced size (# nodes) | | | |
| $2.44 \cdot 10^6$ | $1.26 \cdot 10^7$ | $5.97 \cdot 10^7$ | $2.59 \cdot 10^8$ |
| $2.44 \cdot 10^6$ | $1.26 \cdot 10^7$ | $5.97 \cdot 10^7$ | $2.59 \cdot 10^8$ |
| Median size (# nodes) | | | |
| $1.13 \cdot 10^4$ | $1.52 \cdot 10^4$ | $1.87 \cdot 10^4$ | $2.18 \cdot 10^4$ |
| $8.79 \cdot 10^3$ | $1.19 \cdot 10^4$ | $1.47 \cdot 10^4$ | $1.73 \cdot 10^4$ |
| Node-to-leaf arcs ratio | | | |
| 8.73% | 7.65% | 6.67% | 5.80% |
| 22.36% | 18.34% | 14.94% | 12.29% |

(b) Tic-Tac-Toe

Table 3: Effect of pruning on performance. The first row for each feature is *without* pruning and the second is *with* pruning.

| | depth | size |
|---|---|---|
| Time (s) | 5862 | 5868 |
| $O(\text{sort}(N))$ | 496 | 476 |
| $O(N/B)$ | 735 | 755 |
| $N$ (BDD nodes) | $2.68 \cdot 10^{10}$ | |
| $N$ (MiB) | 614313 | |

(a) *mem_ctrl*

| | depth | size |
|---|---|---|
| Time (s) | 3.89 | 3.27 |
| $O(\text{sort}(N))$ | 22 | 22 |
| $O(N/B)$ | 3 | 3 |
| $N$ (BDD nodes) | $8.02 \cdot 10^{6}$ | |
| $N$ (MiB) | 3589 | |

(b) *sin*

| | depth | size |
|---|---|---|
| Time (s) | 0.058 | 0.006 |
| $O(\text{sort}(N))$ | 1 | 0 |
| $O(N/B)$ | 0 | 1 |
| $N$ (BDD nodes) | $2.51 \cdot 10^{5}$ | |
| $N$ (MiB) | 5.74 | |

(c) *voter*

Table 4: Running time for equivalence testing. $O(\text{sort}(N))$ and $O(N/B)$ is the number of times the respective algorithm in Section 3.3.3 was used.

### 5.3.5 Research Question 2d:

Table 4 shows the time to do equality checking on the three largest circuits verified with the Picotrav application run on the Grendel server nodes using Adiar with 300 GiB available. The number of nodes and size reported in the table is of a single set of BDDs that describe the final circuit. That is, since Adiar does not share nodes, then the set of final BDDs for the specification and the optimised circuit are distinct; the *mem_ctrl* benchmark requires 1231 isomorphism checks on a total of $2 \cdot 2.68 \cdot 10^{10}$ nodes which is equivalent to 1.17 TiB of data. In these benchmarks all equality checking was possible to do with a weighted average performance of 0.109 $\mu$s/node.

The *voter* benchmark is especially interesting, since it consists only of a single output gate and the $O(\text{sort}(N))$ and $O(N/B)$ algorithm are used respectively in the depth and size optimised instance. As witnessed in Section 5.3.1, Adiar has a bad performance for smaller instances. Yet, despite only a total of 11.48 MiB of data is compared, the $O(\text{sort}(N))$ algorithm runs at 0.116 $\mu$s/node and the $O(N/B)$ scan at only 0.012 $\mu$s/node. That is, the $O(N/B)$ algorithm can compare at least $2 \cdot 5.75$ MiB/0.006 s = 1.86 GiB/s.

Table 5 shows the running time of equality checking by instead computing whether $f \leftrightarrow g = \top$. Not even taking the speedup due to the priority queue and separation of node-to-leaf arcs in Section 5.3.2 and 5.3.3 into account, this approach, as is necessary with Arge's original algorithms, is 2.42 – 63.5 slower than using our improved algorithms.

Out of the 3861 output gates checked for equality throughout the 23 verified circuits the $O(N/B)$ linear scan could be used for 71.6% of them.

| | depth | size |
|---|---|---|
| Time (s) | 44233.2 | 44263.4 |

(a) *mem_ctrl*

| | depth | size |
|---|---|---|
| Time (s) | 10.165 | 7.911 |

(b) *sin*

| | depth | size |
|---|---|---|
| Time (s) | 0.380 | 0.381 |

(c) *voter*

Table 5: Running time for checking equality with $f \leftrightarrow g = \top$

#### 5.3.6 Research Question 3:

We have compared the performance of *Adiar* 1.0.1 with the *BuDDy* 2.4 [30], the *CUDD* 3.0.0 [43], and the *Sylvan* 1.5.0 [19] BDD package.

To this end, we ran all our benchmarks on Grendel server nodes, which were set to use 350 GiB of the available RAM, while each BDD package is given 300 GiB of it. Sylvan was set to not use any parallelisation and given a ratio between the unique node table and the computation cache of 64:1. BuDDy was set to the same cache-ratio and the size of the CUDD cache was set such it would have an equivalent ratio when reaching its 300 GiB limit. The I/O analysis in Section 2.2.1 is evident in the running time of Sylvan's implicit Reduce, which increases linearly with the size of the node table[4]. Hence, Sylvan has been set to start its table $2^{12}$ times smaller than the final 262 GiB it may occupy, i.e. at first with a table and cache that occupies 66 MiB.

As is evident in Section 5.3.1, the slowdown for Adiar for small instances, due to its use of external memory algorithms, makes it meaningless to compare its performance to other BDD packages when the largest BDD is smaller than 32 MiB. Hence, we have chosen to omit these instances from this report, though the full data set (including these instances) is publicly available.

**N-Queens.** Fig. 12 shows for each BDD package their running time computing the $N$-Queens benchmark for $12 \leq N \leq 17$. At $N = 13$, where Adiar's computation time per node is the lowest, Adiar runs by a factor of 5.1 slower than BuDDy, 2.3 than CUDD, and 2.6 than Sylvan. The gap in performance of Adiar and other packages gets smaller as instances grow larger: for $N = 15$, which is the largest instance solvable by CUDD and Sylvan, Adiar is only slower than CUDD, resp. Sylvan, by a factor of 1.47, resp. 2.15.

Adiar outperforms all three libraries in terms of successfully computing very large instances. The largest BDD constructed by Adiar for $N = 17$ is 719 GiB in size, whereas Sylvan with $N = 15$ only constructs BDDs up to 12.9 GiB in size. Yet, at $N = 17$, where Adiar has to make heavy use of the disk, Adiar's

[4]Experiments using *perf* on Sylvan show that dereferencing a bucket in the unique node table and using x86 locks to obtain exclusive ownership of cache lines are one of the main reasons for the slowdown. We hypothesise that the overhead of *mmap* is the main culprit.

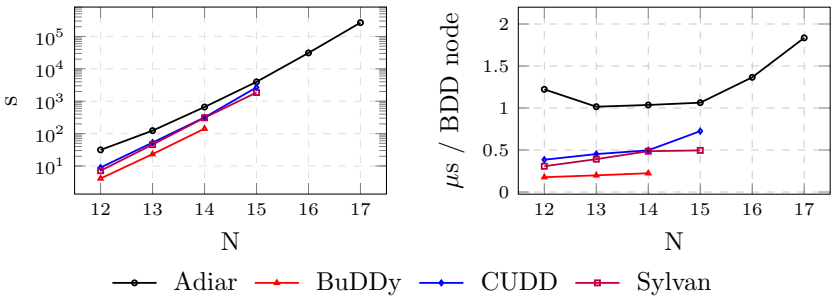



Figure 12: Running time solving $N$-Queens (lower is better).

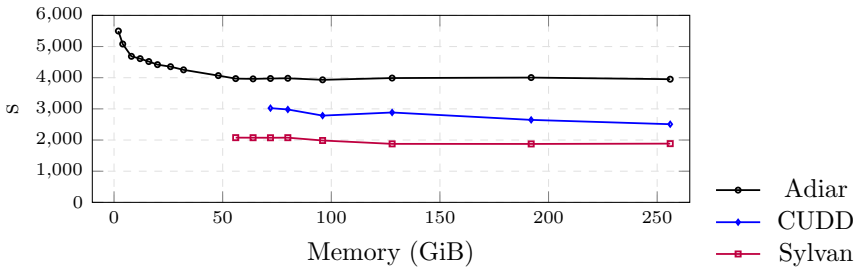



Figure 13: Running time of 15-Queens with variable memory (lower is better).

computation time per node only slows down by a factor of 1.8 compared to its performance at $N = 13$.

Conversely, Adiar is also able to solve smaller instances with much less memory than other packages. Fig 13 shows the running time for both Adiar, CUDD, and Sylvan solving the 15-Queens problem depending on the available memory. Sylvan was first able to solve this problem when given 56 GiB of memory while CUDD, presumably due to its larger node-size and multiple data structures, requires 72 GiB of memory to be able to compute the solution.

**Tic-Tac-Toe.** The running times we obtained for this benchmark, as shown in Fig. 14, paint the same picture as for Queens above: the factor with which Adiar runs slower than the other packages decreases as the size of the instance increases. At $N = 24$, which is the largest instance solved by CUDD, Adiar runs slower than CUDD by a factor of 2.38. The largest instance solved by Sylvan is $N = 25$ where the largest BDD created by Sylvan is 34.4 GiB in size, and one incurs a 2.50 factor slowdown by using Adiar instead.

Adiar was able to solve the instance of $N = 29$, where the largest BDD created was 902 GiB in size. Yet, even though the disk was extensively used, Adiar's computation time per node only slows down by a factor of 2.49 compared to its performance at $N = 22$.

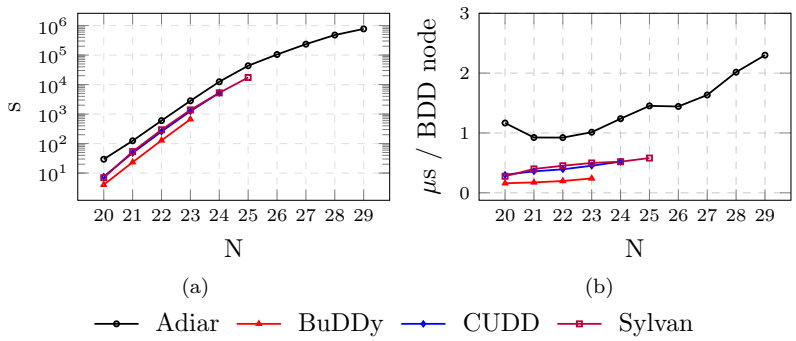



Figure 14: Running time solving Tic-Tac-Toe (lower is better).

**Picotrav.** Table 6 shows the number of successfully verified circuits by each BDD package and the number of benchmarks that were unsuccessful due to a full node table or a full disk and the ones that timed out after 15 days of computation time.

| | # solved | # out-of-space | # time-out |
|---|---|---|---|
| Adiar | 23 | 6 | 11 |
| BuDDy | 19 | 20 | 1 |
| CUDD | 20 | 19 | 1 |
| Sylvan | 20 | 13 | 7 |

Table 6: Number of verified *arithmetic* and *random/control* circuits from [2]

If Sylvan's unique node table and computation cache are immediately instantiated to their full size of 262 GiB then it is able to verify 3 more of the 40 circuits within the 15 days time limit. One of these three is the *arbiter* benchmark optimised with respect to size that BuDDy and CUDD are able to solve in a few seconds. Yet, BuDDy also exhibits a similar slowdown when it has to double its unique node table to its full size. We hypothesise this slowdown is due to the computation cache being cleared when nodes are rehashed into the doubled node table, while this benchmark consists of a lot of repeated computations. The other are the *mem_ctrl* benchmark optimised with respect to size and depth.

The performance of Adiar compared to the other packages is reminiscent to our results from the two other benchmarks. For example, the *voter* benchmark, where the largest BDD for Adiar is 8.2 GiB in size, it is 3.69 times slower than CUDD and 3.07 times slower than Sylvan. In the *mem_ctrl* benchmark optimised for depth which Sylvan barely was able to solve by changing its cache ratio, Adiar is able to construct the BDDs necessary for the comparison 1.01 times faster than Sylvan. This is presumably due to the large overhead of Sylvan

in having to repeatedly run its garbage collection algorithms.

Despite the fact that the disk available to Adiar is only 12 times larger than internal memory, Adiar has to explicitly store both the unreduced and reduced BDDs, and many of the benchmarks have hundreds of BDDs concurrently alive, Adiar is still able to solve the same benchmarks as the other packages. For example, the largest solved benchmark, *mem_ctrl*, has at one point 1231 different BDDs in use at the same time.

# 6 Conclusions and Future Work

We propose I/O-efficient BDD manipulation algorithms in order to scale BDDs beyond main memory. These new iterative BDD algorithms exploit a total topological sorting of the BDD nodes, by using priority queues and sorting algorithms. All recursion requests for a single node are processed at the same time, which fully circumvents the need for a memoisation table. If the underlying data structures and algorithms are cache-aware or cache-oblivious, then so are our algorithms by extension. The I/O complexity of our time-forward processing algorithms is compared to the conventional recursive algorithms on a unique node table with complement edges [9] in Table 7.

The performance of Adiar is very promising in practice for instances larger than a few hundred MiB. As the size of the BDDs increase, the performance of Adiar gets closer to conventional recursive BDD implementations. When the BDDs involved in the computation exceeds a few GiB then the use of Adiar only results in a 3.69 factor slowdown compared to Sylvan and CUDD – it was only 1.47 times slower than CUDD in the largest Queens benchmark that CUDD could solve. Simultaneously, the design of our algorithms allow us to compute on BDDs that outgrow main memory with only a 2.49 factor slowdown, which is negligible in comparison to the slowdown that conventional BDD packages experience when using swap memory.

This performance comes at the cost of not sharing any nodes and so not being able to compare for functional equivalence in $O(1)$ time. We have improved the asymptotic behaviour of equality checking to only be an $O(\text{sort}(N))$ algorithm which in practice is negligible compared to the time to construct the BDDs involved. For 71.6% of all the output gates we verified from *EPFL* Combinational Benchmark Suite [2] we were even able to do so with a simple $O(N/B)$ linear scan that can compare more than 1.86 GiB/s. This number is realistic, since modern SSDs, depending on block size used, have a sequential transfer rate of 1 GiB/s and 2.8 GiB/s.

In practice, the fact that nodes are not shared does not negatively impact the ability of Adiar to solve a problem in comparison to conventional BDD packages. This is despite the ratio between disk and main memory is smaller than the number of BDDs in use. Furthermore, garbage collection becomes a trivial and cheap deletion of files on disk.

This lays the foundation on which we intend to develop external memory BDD algorithms usable in the fixpoint algorithms for symbolic model check-

ing. We will, to this end, attempt to improve further on the non-constant equality checking, improve the performance of quantifying multiple variables, and design an I/O-efficient relational product function. Furthermore, we intend to improve Adiar's performance for smaller instances by processing them exclusively in internal memory and generalise its implementation to also include Multi-Terminal [21] and Zero-suppressed [33] Decision Diagrams.

| Algorithm | | Depth-first | Time-forwarded |
|---|---|---|---|
| Reduce | | $O(N)$ | $O(\text{sort}(N))$ |
| BDD Manipulation | | | |
| Apply | $f \odot g$ | $O(N_f N_g)$ | $O(\text{sort}(N_f N_g))$ |
| If-Then-Else | $f \ ? \ g \ : \ h$ | $O(N_f N_g N_h)$ | $O(\text{sort}(N_f N_g N_h))$ |
| Restrict | $f\|_{x_i=v}$ | $O(N_f)$ | $O(\text{sort}(N_f))$ |
| Negation | $\neg f$ | $O(1)$ | $O(1)$ |
| Quantification | $\exists/\forall v : f\|_{x_i=v}$ | $O(N_f^2)$ | $O(\text{sort}(N_f^2))$ |
| Composition | $f\|_{x_i=g}$ | $O(N_f^2 N_g)$ | $O(\text{sort}(N_f^2 N_g))$ |
| Counting | | | |
| Count Paths | #paths in $f$ to $\top$ | $O(N_f)$ | $O(\text{sort}(N_f))$ |
| Count SAT | $\#x : f(x)$ | $O(N_f)$ | $O(\text{sort}(N_f))$ |
| Other | | | |
| Equality | $f \equiv g$ | $O(1)$ | $O(\text{sort}(\min(N_f, N_g)))$ |
| Evaluate | $f(x)$ | $O(L_f)$ | $O(N_f/B)$ |
| Min SAT | $\min\{x \mid f(x)\}$ | $O(L_f)$ | $O(N_f/B)$ |
| Max SAT | $\max\{x \mid f(x)\}$ | $O(L_f)$ | $O(N_f/B)$ |

Table 7: I/O-complexity of conventional depth-first algorithms compared to the time-forwarded we propose. Here, $N/B < \text{sort}(N) \triangleq N/B \cdot \log_{M/B}(N/B) \ll N$, where $N$ is the number of nodes, and $L$ the number of levels in the BDD.

## Acknowledgements

Thanks to the late Lars Arge for his input and advice and to Mathias Rav for helping us with TPIE and his inputs on the levelized priority queue. Thanks to Gerth S. Brodal, Asger H. Drewsen for their help, Casper Rysgaard for his help with Lemma 3.2, and to Alfons W. Laarman, Tom van Dijk and the anonymous peer reviewers for their valuable feedback. Finally, thanks to the Centre for Scientific Computing Aarhus, (phys.au.dk/forskning/cscaa/) to allow us to run our benchmarks on their Grendel cluster.

## References

[1] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–

1127, 1988.

[2] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL combinational benchmark suite. In *24th International Workshop on Logic & Synthesis*, 2015.

[3] Elvio Amparore, Susanna Donatelli, and Francesco Gallà. A CTL* model checker for Petri nets. In *Application and Theory of Petri Nets and Concurrency*, volume 12152 of *Lecture Notes in Computer Science*, pages 403–413. Springer, 2020.

[4] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Workshop on Algorithms and Data Structures (WADS)*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345, Berlin, Heidelberg, 1995. Springer.

[5] Lars Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *6th International Symposium on Algorithms and Computations (ISAAC)*, volume 1004 of *Lecture Notes in Computer Science*, pages 82–91, 1995.

[6] Lars Arge. The I/O-complexity of ordered binary-decision diagram. In *BRICS RS preprint series*, volume 29. Department of Computer Science, University of Aarhus, 1996.

[7] Pranav Ashar and Matthew Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 622–627. IEEE Computer Society Press, 1994.

[8] Shoham Ben-David, Tamir Heyman, Orna Grumberg, and Assaf Schuster. Scalable distributed on-the-fly symbolic model checking. In *Formal Methods in Computer-Aided Design*, pages 427–441, Berlin, Heidelberg, 2000. Springer.

[9] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th Design Automation Conference (DAC)*, pages 40–45. Association for Computing Machinery, 1990.

[10] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[11] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[12] Randal E. Bryant. Binary decision diagrams. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 191–217. Springer International Publishing, Cham, 2018.

[13] Randal E. Bryant and Marijn J. H. Heule. Dual proof generation for quantified Boolean formulas with a BDD-based solver. In *28th International Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 433–449. Springer International Publishing, 2021.

[14] Randal E. Bryant and Marijn J. H. Heule. Generating extended resolution proofs with a BDD-based sat solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12651 of *Lecture Notes in Computer Science*, pages 76–93. Springer International Publishing, 2021.

[15] Randal E. Bryant and Marijn J. H. Heule. Generating extended resolution proofs with a BDD-based sat solver. arXiv, 2021.

[16] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. arXiv, 2016.

[17] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364, Berlin, Heidelberg, 2002. Springer.

[18] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:410–425, 2000.

[19] Tom Van Dijk and Jaco Van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19:675–696, 2016.

[20] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (FOCS 1999)*, pages 285–297. IEEE Computer Society Press, 1999.

[21] M. Fujita, P.C. McGeer, and J.C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169, 1997.

[22] Peter Gammie and Ron Van der Meyden. MCK: Model checking the logic of knowledge. In *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 479–483, Berlin, Heidelberg, 2004. Springer.

[23] Orna Grumberg, Tamir Heyman, and Assaf Schuster. Distributed symbolic model checking for $\mu$-calculus. *Formal Methods in System Design*, 26:197–219, 2005.

[24] Leifeng He and Guanjun Liu. Petri net based symbolic model checking for computation tree logic of knowledge. arXiv, 2020.

[25] Gijs Kant, Alfons Laarman, Jeroenn Meijer, Jaco Van de Pol, Stefan Blom, and Tom Van Dijk. LTSmin: High-performance language-independent model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *Lecture Notes in Computer Science*, pages 692–707, Berlin, Heidelberg, 2015. Springer.

[26] Kevin Karplus. Representing Boolean functions with if-then-else DAGs. Technical report, University of California at Santa Cruz, USA, 1988.

[27] Nils Klarlund and Theis Rauhe. BDD algorithms and cache misses. In *BRICS Report Series*, volume 26, 1996.

[28] Donald E Knuth. *The art of computer programming, volume 4A: combinatorial algorithms, part 1.* Addison-Wesley Professional, 2011.

[29] Daniel Kunkle, Vlad Slavici, and Gene Cooperman. Parallel disk-based computation for large, monolithic binary decision diagrams. In *4th International Workshop on Parallel Symbolic Computation (PASCO)*, pages 63–72, 2010.

[30] Jørn Lind-Nielsen. BuDDy: A binary decision diagram package. Technical report, Department of Information Technology, Technical University of Denmark, 1999.

[31] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: an open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, 19:9–30, 2017.

[32] David E. Long. The design of a cache-friendly BDD library. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 639–645. Association for Computing Machinery, 1998.

[33] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th Design Automation Conference (DAC)*, pages 272–277. Association for Computing Machinery, 1993.

[34] Shin-ichi Minato and Shinya Ishihara. Streaming BDD manipulation for large-scale combinatorial problems. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 702–707, 2001.

[35] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *27th Design Automation Conference (DAC)*, pages 52–57. Association for Computing Machinery, 1990.

[36] Thomas Mølhave. Using TPIE for processing massive data sets in C++. *SIGSPATIAL Special*, 4(2):24–27, 2012.

[37] Hiroyuki Ochi, Koichi Yasuoka, and Shuzo Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *International Conference on Computer Aided Design (ICCAD)*, pages 48–55. IEEE Computer Society Press, 1993.

[38] Lars Hvam Petersen. External priority queues in practice. Master's thesis, Department of Computer Science, University of Aarhus, 2007.

[39] Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5:7–32, 2000.

[40] Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierarchy. In *33rd Design Automation Conference (DAC)*, pages 635–640. Association for Computing Machinery, 1996.

[41] Steffan Christ Sølvsten and Jaco Van de Pol. Adiar v1.0.1 : Experiment data. Zenodo, 2021.

[42] Steffan Christ Sølvsten, Jaco van de Pol, Anna Blume Jakobsen, and Mathias Weller Berg Thomasen. Adiar: Binary decision diagrams in external memory. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13244 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2022.

[43] Fabio Somenzi. CUDD: CU decision diagram package, 3.0. Technical report, University of Colorado at Boulder, 2015.

[44] Miroslav N. Velev and Ping Gao. Efficient parallel GPU algorithms for BDD manipulation. In *19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 750–755. IEEE Computer Society Press, 2014.

[45] Darren Erik Vengroff. A Transparent Parallel I/O Environment. In *DAGS Symposium on Parallel Computation*, pages 117–134, 1994.

# Adiar 1.1
## Zero-Suppressed Decision Diagrams in External Memory

Steffan Christ Sølvsten( ) and Jaco van de Pol

Aarhus University, Aarhus, Denmark
{soelvsten,jaco}@cs.au.dk

**Abstract.** We outline how support for Zero-suppressed Decision Diagrams (ZDDs) has been achieved for the external memory BDD package Adiar. This allows one to use ZDDs to solve various problems despite their size exceed the machine's limit of internal memory.

**Keywords:** Zero-suppressed Decision Diagrams · External Memory Algorithms

## 1 Introduction

Minato introduced Zero-suppressed Decision Diagrams (ZDDs) [15] as a variation on Bryant's Binary Decision Diagrams (BDDs) [5]. ZDDs provide a canonical description of a Boolean $n$-ary function $f$ that is more compact than the corresponding BDD when $f$ is a characteristic function for a family $F \subseteq \{0,1\}^n$ of sparse vectors over some universe of $n$ variables. This makes ZDDs not only useful for solving combinatorial problems [15] but they can also surpass BDDs in the context of symbolic model checking [21] and they are the backbone of the PolyBoRi library [4] used in algebraic cryptoanalysis.

The Adiar BDD package [19] provides an implementation of BDDs in C++17 that is I/O-efficient [1]. This allows Adiar to manipulate BDDs that outgrow the size of the machine's internal memory, i.e., RAM, by efficiently exploiting how they are stored in external memory, i.e., on the disk. The source code for Adiar is publicly available at

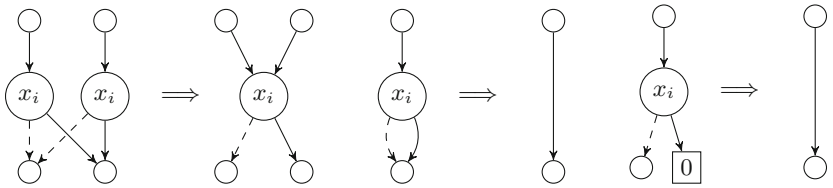
github.com/ssoelvsten/adiar

All 1.$x$ versions of Adiar have only been tested on Linux with GCC. But, with version 2.0, it is ensured that Adiar supports the GCC, Clang, and MSVC compilers on Linux, Mac, and Windows.

We have added in Adiar 1.1 support for the basic ZDD operations while also aiming for the following two criteria: the addition of ZDDs should (1) avoid any code duplication to keep the codebase maintainable and (2) not negatively impact the performance of existing functionality. Section 2 describes how this was achieved and Sect. 3 provides an evaluation.

Other mature BDD packages also support ZDDs, e.g., CUDD [20], BiDDy [13], Sylvan [8] and PJBDD [2], but unlike Adiar none of these support manipulation of ZDDs beyond main memory. The only other BDD package designed for out-of-memory BDD manipulation, CAL [16], does not support ZDDs.


K. Y. Rozier and S. Chaudhuri (Eds.): NFM 2023, LNCS 13903, pp. 464–471, 2023.
https://doi.org/10.1007/978-3-031-33170-1_28




(a) Node Merging (b) BDD Rule (c) ZDD Rule

**Fig. 1.** Reduction Rules for BDDs and ZDDs.

## 2 Supporting both BDDs and ZDDs

The Boolean function $f : \{0,1\}^n \to \{0,1\}$ is the characteristic function for the set of bitvectors $F = \{\boldsymbol{x} \in \{0,1\}^n \mid f(\boldsymbol{x}) = 1\}$. Each bitvector $\boldsymbol{x}$ is equivalent to a conjunction of the indices set to 1 and hence $F$ can quite naturally be described as a DNF formula, i.e., a set of set of variables.

A decision diagram is a rooted directed acyclic graph (DAG) with two sinks: a 0-leaf and a 1-leaf. Each internal node has two children and contains the label $i \in \mathbb{N}$ to encode the *if-then-else* of a variable $x_i$. The decision diagram is *ordered* by ensuring each label only occurs once and in sorted order on all paths from the root. The diagram is also *reduced* if duplicate nodes are merged as shown in Fig. 1a. Furthermore as shown respectively in Fig. 1b and 1c, BDDs and ZDDs also suppress a certain type of nodes as part of their reduction to further decrease the diagram's size. The suppression rule for ZDDs in Fig. 1c ensures each path in the diagram corresponds one-to-one to a term of the DNF it represents.

Both BDDs and ZDDs provide a succinct way to manipulate Boolean formulae by computing on their graph-representation instead. The difference in the type of node being suppressed in each type of decision diagram has an impact on the logic within these graph algorithms. For example, applying a binary operator, e.g., *and* for BDDs and *intersection* for ZDDs, is a product construction for both types of decision diagrams. But since the *and* operator is shortcutted by the 0-leaf, the computation depends on the shape of the suppressed nodes.

Hence, as shown in Fig. 2, we have generalized the relevant algorithms in Adiar with a *policy-based design*, i.e., a compile-time known *strategy pattern*, so the desired parts of the code can be varied internally. For example, most of the logic within the BDD product construction has been moved to the templated `product_construction` function. The code-snippets that distinguish the `bdd_apply` from the corresponding ZDD operation `zdd_binop` are encapsulated within the two *policy* classes: `apply_prod_policy` and `zdd_prod_policy`. This ensures that no code duplication is introduced. This added layer of abstraction has no negative impact on performance, since the function calls are known and inlined at compile-time. No part of this use of templates is exposed to the end-user, by ensuring that each templated algorithm is compiled into its final algorithms within Adiar's `.cpp` files.

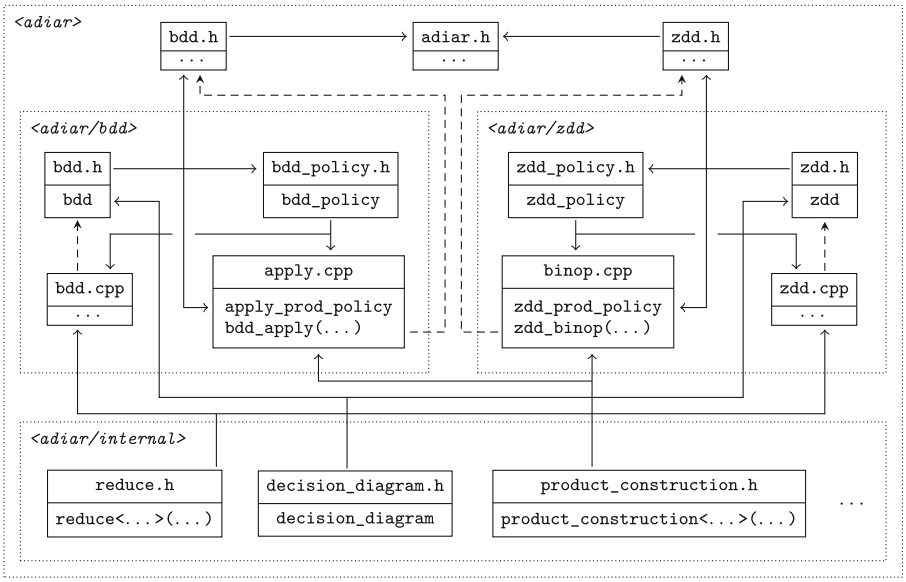



**Fig. 2.** Architecture of Adiar v1.1. Solid lines are direct inclusions of one file in another while dashed lines represent the implementation of a declared function.

**Table 1.** Supported ZDD operations in Adiar v1.1. The semantics views a ZDD as a set of sets of variables in *dom*.

| Adiar ZDD function | Operation Semantics | Generalised BDD function |
|---|---|---|
| ZDD Manipulation | | |
| `zdd_binop`$(A, B, \otimes)$ | $\{x \mid x \in A \otimes x \in B\}$ | `bdd_apply` |
| `zdd_change`$(A,$ `vars`$)$ | $\{(a \setminus vars) \cup (vars \setminus a) \mid a \in A\}$ | |
| `zdd_complement`$(A,$ `dom`$)$ | $\mathcal{P}(dom) \setminus A$ | |
| `zdd_expand`$(A,$ `vars`$)$ | $\bigcup_{a \in A} \{a \cup v \mid v \in \mathcal{P}(vars)\}$ | |
| `zdd_offset`$(A,$ `vars`$)$ | $\{a \in A \mid vars \cap a = \emptyset\}$ | `bdd_restrict` |
| `zdd_onset`$(A,$ `vars`$)$ | $\{a \in A \mid vars \subseteq a\}$ | `bdd_restrict` |
| `zdd_project`$(A,$ `vars`$)$ | $proj_{vars}(A)$ | `bdd_exists` |
| Counting | | |
| `zdd_size`$(A)$ | $\lvert A \rvert$ | `bdd_pathcount` |
| `zdd_nodecount`$(A)$ | $N_A$ | `bdd_nodecount` |
| `zdd_varcount`$(A)$ | $L_A$ | `bdd_varcount` |
| Predicates | | |
| `zdd_equal`$(A,\ B)$ | $A = B$ | `bdd_equal` |
| `zdd_unequal`$(A,\ B)$ | $A \neq B$ | `bdd_equal` |
| `zdd_subseteq`$(A,\ B)$ | $A \subseteq B$ | `bdd_equal` |
| `zdd_disjoint`$(A,\ B)$ | $A \cap B = \emptyset$ | `bdd_equal` |
| Set elements | | |
| `zdd_contains`$(A,$ `vars`$)$ | $vars \in A$ | `bdd_eval` |
| `zdd_minelem`$(A)$ | $\min(A)$ | `bdd_satmin` |
| `zdd_maxelem`$(A)$ | $\max(A)$ | `bdd_satmax` |
| Conversion | | |
| `zdd_from`$(f,$ `dom`$)$ | $\{x \in \mathcal{P}(dom) \mid f(x) = \top\}$ | |
| `bdd_from`$(A,$ `dom`$)$ | $\boldsymbol{x} : \mathcal{P}(dom) \mapsto \boldsymbol{x} \in A$ | |

For each type of decision diagram there is a class, e.g., `bdd`, and a separate policy, e.g., `bdd_policy`, that encapsulates the common logic for that type of decision diagram, e.g., the reduction rule in Fig. 1b and the `bdd` type. This policy is used within the `bdd` and `zdd` class to instantiate the specific variant of the Reduce algorithm that is applied after each operation. The algorithm policies, e.g., the two product construction policies above, also inherit information from this diagram-specific policy. This ensures the policies can provide the information needed by the algorithm templates.

Table 1 provides an overview of all ZDD operations provided in Adiar 1.1, including what BDD operations they are generalized from. All but five of these ZDD operations could be implemented by templating the current codebase. The remaining five operations required the addition of only a single new algorithm of similar shape to those in [19]; the differences among these five could be encapsulated within a policy for each operation.

## 3 Evaluation

### 3.1 Cost of Modularity

Table 2a shows the size of the code base, measured in lines of code (LOC), and Table 2b the number of unique operations in the public API with and without aliases. Due to the added modularity and features the entire code base grew by a factor $\frac{6305}{3961} = 1.59$. Yet, the size of the public API excluding aliases increased by a factor of $\frac{23+24}{22} = 2.14$; including aliases the public API grew by a factor of 1.98.

### 3.2 Experimental Evaluation

**Impact on BDD Performance.** Table 3 shows the performance of Adiar before and after implementing the architecture in Sect. 2. These two benchmarks, $N$-Queens and Tic-Tac-Toe, were used in [19] to evaluate the performance of its BDDs – specifically to evaluate its `bdd_apply` and `reduce` algorithms. The choice of $N$ is based on limitations in Adiar v1.0 and v1.1 (which are resolved in Adiar v1.2). We ran these benchmarks on a consumer grade laptop with a 2.6 GHz Intel i7-4720HQ processor, 8 GiB of RAM (4 of which was given to Adiar) and 230 GiB SSD disk.

**Table 2.** Lines of Code compared to number of functions in Adiar's API.

| Folder | v1.0 | v1.1 |
|---|---|---|
| `adiar` | 3939 | 1643 |
| `adiar/bdd` | – | 1019 |
| `adiar/zdd` | – | 1052 |
| `adiar/internal` | – | 2568 |

(a) Lines of Code.

| Adiar's API | v1.0 | v1.1 |
|---|---|---|
| BDD | 22 | 23 |
| (w/aliases) | +20 | +22 |
| ZDD | – | 24 |
| (w/aliases) | – | +14 |

(b) Size of the Public API.

**Table 3.** Minimum running time (s) before and after the changes in Sect. 2.

| $N$ | Before (v1.0) | After (v1.1) |
|---|---|---|
| 13 | 107.8 | 108.9 |
| 14 | 680.8 | 625.2 |

(a) Queens

| $N$ | Before (v1.0) | After (v1.1) |
|---|---|---|
| 22 | 616.8 | 517.9 |
| 23 | 3202.9 | 2881.1 |

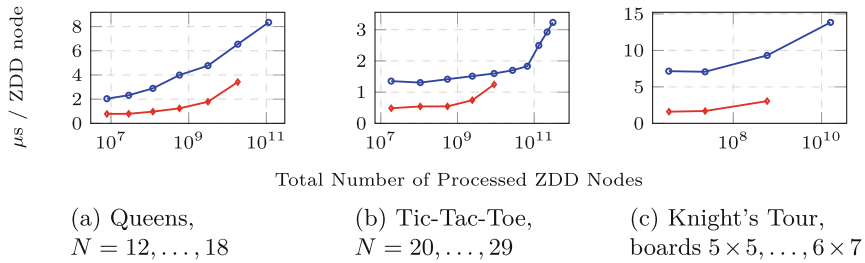
(b) Tic-Tac-Toe

The 1% slowdown for the 13-Queens problem is well within the experimental error introduced by the machine's hardware and OS. Furthermore, the three other benchmarks show a performance increase of 9% or more. Hence, it is safe to conclude that the changes to Adiar have not negatively affected its performance.

**ZDD Performance.** We have compared Adiar 1.1's and CUDD 3.0's [20] performance manipulating ZDDs. Our benchmarks are, similar to Sect. 2, templated with *adapters* for each BDD package. Sylvan 1.7 [8] and BiDDy 2.2 [13] are not part of this evaluation since they have no C++ interface; to include them, we essentially would have to implement a free/protect mechanism for ZDDs for proper garbage collection.

Figure 3 shows the normalized minimal running time of solving three combinatorial problems: the $N$-Queens and the Tic-Tac-Toe benchmarks from earlier and the (open) Knight's Tour problem based on [6]. We focus on combinatorial problems due to what functionality is properly supported by Adiar at time of writing. These experiments were run on the server nodes of the *Centre for Scientific Computing, Aarhus*. Each node has two 3.0 GHz Intel Xeon Gold 6248R processors, 384 GiB of RAM (300 of which was given to the BDD package), 3.5 TiB of available SSD disk, runs CentOS Linux, and uses GCC 10.1.0.

Adiar is significantly slower than CUDD for small instances due to the overhead of initialising and using external memory data structures. Hence, Fig. 3 only shows the instances where the largest ZDD involved is 10 MiB or larger since these meaningfully compare the algorithms in Adiar with the ones in CUDD.




(a) Queens, $N = 12, \ldots, 18$ (b) Tic-Tac-Toe, $N = 20, \ldots, 29$ (c) Knight's Tour, boards $5 \times 5, \ldots, 6 \times 7$

**Fig. 3.** Normalised minimal running time of Adiar (blue) and CUDD (red). (Color figure online)

Similar to the results in [19], also for ZDDs the gap in running time between Adiar and CUDD shrinks as the instances grow. When solving the 15-Queens problem, Adiar is 3.22 times slower than CUDD whereas for the 17-Queens problem it is only 1.91 times slower. The largest Tic-Tac-Toe instance solved by CUDD was $N = 24$ where Adiar was only 1.22 times slower. In both benchmarks, Adiar handles more instances than CUDD: 18-Queens, resp. Tic-Tac-Toe for $N = 29$, results in a single ZDD of 512.8 GiB, resp. 838.9 GiB, in size.

The Knight's Tour benchmark stays quite benign up until a chess board of $6 \times 6$. From that point, the computation time and size of the ZDDs quickly explode. Adiar solved up to the $6 \times 7$ board in 2.5 days, where the largest ZDD was only 2 GiB in size. We could not solve this instance with CUDD within 15 days. For instances also solved by CUDD, Adiar was up to 4.43 times slower.

## 4 Conclusion and Future Work

While the lines of code for Adiar's BDDs has slightly increased, that does not necessarily imply an increase in the code's complexity. Notice that the architecture in Sect. 2 separates the recursive logic of BDD and ZDD manipulation from the logic used to make these operations I/O-efficient. In fact, this separation significantly improved the readability and maintainability of both halves. Furthermore, the C++ templates allow the compiler to output each variant of an algorithm as if it was written by hand. Hence, as Sect. 3 shows, the addition of ZDDs has not decreased Adiar's ability to handle BDDs efficiently.

Adiar can be further modularized by templating diagram nodes to vary their data and outdegree at compile-time. This opens the possibility to support Multi-terminal [9], List [8], Functional [11], and Quantum Multiple-valued [14] Decision Diagrams. If nodes support variadic out-degrees at run-time, then support for Multi-valued [10] and Clock Difference [12] Decision Diagrams is possible and it provides the basis for an I/O-efficient implementation of Annotated Terms [3].

This still leaves a vital open problem posed in [19] as future work: the current technique used to achieve I/O-efficiency does not provide a translation for operations that need to recurse multiple times for a single diagram node. Hence, I/O-efficient dynamic variable reordering is currently not supported. Similarly, `zdd_project` in Adiar v1.1 may be significantly slower than its counterparts in other BDD packages. This also hinders the implementation of other complex operations, such as the multiplication operations in [4,14,15], the generalisation of composition in [5] to multiple variables, and the Restrict operator in [7].

**Acknowledgements.** Thanks to Marijn Heule and Randal E. Bryant for requesting ZDDs are added to= Adiar. Thanks to the Centre for Scientific Computing, Aarhus, (phys.au.dk/forskning/cscaa/) for running our benchmarks.

**Data Availibility Statement.** The data presented in Sect. 3 is available at [18] while the code to obtain this data is provided at [17].

## References


1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988). https://doi.org/10.1145/48529.48535
2. Beyer, D., Friedberger, K., Holzner, S.: PJBDD: a BDD library for Java and multithreading. In: Hou, Z., Ganesh, V. (eds.) ATVA 2021. LNCS, vol. 12971, pp. 144–149. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_10
3. BrandVan den Brand, M.G.J., Jongde Jong, H.A., Klint, P., Olivier, P.: Efficient annotated terms. Softw. Pract. Exp. **30**, 259–291 (2000)
4. Brickenstein, M., Dreyer, A.: PolyBoRi: a framework for Gröbner-basis computations with Boolean polynomials. J. Symb. Comput. **44**(9), 1326–1345 (2009). https://doi.org/10.1016/j.jsc.2008.02.017
5. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
6. Bryant, R.E.: `Cloud-BDD`: Distributed implementation of BDD package (2021). https://github.com/rebryant/Cloud-BDD
7. Coudert, O., Madre, J.C.: A unified framework for the formal verification of sequential circuits. In: 1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers, pp. 126–129 (1990). https://doi.org/10.1109/ICCAD.1990.129859
8. Van Dijk, T., Van de Pol, J.: Sylvan: multi-core framework for decision diagrams. Int. J. Softw. Tools Technol. Transf. **19**, 675–696 (2016). https://doi.org/10.1007/s10009-016-0433-2
9. Fujita, M., McGeer, P., Yang, J.Y.: Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. Formal Methods Syst. Des. **10**, 149–169 (1997). https://doi.org/10.1023/A:1008647823331
10. Kam, T., Villa, T., Brayton, R.K., Alberto, L.S.V.: Multi-valued decision diagrams: theory and applications. Multiple-Valued Log. **4**(1), 9–62 (1998)
11. Kebschull, U., Rosenstiel, W.: Efficient graph-based computation and manipulation of functional decision diagrams. In: European Conference on Design Automation with the European Event in ASIC Design, pp. 278–282 (1993). https://doi.org/10.1109/EDAC.1993.386463
12. Larsen, K.G., Weise, C., Yi, W., Pearson, J.: Clock difference diagrams. In: Nordic Workshop on Programming Theory, Turku, Finland. Aalborg Universitetsforlag (1998). https://doi.org/10.7146/brics.v5i46.19491
13. Meolic, R.: BiDDy - a multi-platform academic BDD package. J. Softw. **7**, 1358–1366 (2012). https://doi.org/10.4304/jsw.7.6.1358-1366
14. Miller, D., Thornton, M.: QMDD: a decision diagram structure for reversible and quantum circuits. In: 36th International Symposium on Multiple-Valued Logic, pp. 30–36 (2006). https://doi.org/10.1109/ISMVL.2006.35
15. Minato, S.I.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proceedings of the 30th International Design Automation Conference, pp. 272–277. DAC 1993, Association for Computing Machinery (1993). https://doi.org/10.1145/157485.164890
16. Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.: High performance BDD package by exploiting memory hierarchy. In: 33rd Design Automation Conference (DAC), pp. 635–640. Association for Computing Machinery (1996). https://doi.org/10.1145/240518.240638

17. Sølvsten, S.C., Jakobsen, A.B.: `SSoelvsten/bdd-benchmark`: NASA formal methods 2023. Zenodo, September 2022. https://doi.org/10.5281/zenodo.7040263
18. Sølvsten, S.C., van de Pol, J.: Adiar 1.1.0: experiment data. Zenodo, March 2023. https://doi.org/10.5281/zenodo.7709134
19. Sølvsten, S.C., de Pol, J., Jakobsen, A.B., Thomasen, M.W.B.: Adiar binary decision diagrams in external memory. In: TACAS 2022. LNCS, vol. 13244, pp. 295–313. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_16
20. Somenzi, F.: CUDD: CU decision diagram package, 3.0. Technical report, University of Colorado at Boulder (2015)
21. Yoneda, T., Hatori, H., Takahara, A., Minato, S.: BDDs vs. zero-suppressed BDDs: for CTL symbolic model checking of Petri nets. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 435–449. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031826

# Predicting Memory Demands of BDD Operations using Maximum Graph Cuts

(*Extended Version*)

Steffan Christ Sølvsten Jaco van de Pol

Aarhus University, Denmark {soelvsten,jaco}@cs.au.dk

May 20, 2025

**Abstract**

The BDD package Adiar manipulates Binary Decision Diagrams (BDDs) in external memory. This enables handling big BDDs, but the performance suffers when dealing with moderate-sized BDDs. This is mostly due to initializing expensive external memory data structures, even if their contents can fit entirely inside internal memory.

The contents of these auxiliary data structures always correspond to a graph cut in an input or output BDD. Specifically, these cuts respect the levels of the BDD. We formalise the shape of these cuts and prove sound upper bounds on their maximum size for each BDD operation.

We have implemented these upper bounds within Adiar. With these bounds, it can predict whether a faster internal memory variant of the auxiliary data structures can be used. In practice, this improves Adiar's running time across the board. Specifically for the moderate-sized BDDs, this results in an average reduction of the computation time by 86.1% (median of 89.7%). In some cases, the difference is even 99.9%. When checking equivalence of hardware circuits from the EPFL Benchmark Suite, for one of the instances the time was decreased by 52 hours.

## 1 Introduction

A Binary Decision Diagrams (BDD) [8] is a data structure that has found great use within the field of combinatorial logic and verification. Its ability to concisely represent and manipulate Boolean formulae is the key to many symbolic model checkers, e.g. [3,14,15,17,18,20,24] and recent symbolic synthesis algorithms [23]. Bryant and Heule recently found a use for BDDs to create SAT and QBF solvers with certification capabilities [9–11] that are better at proof generation than conventional SAT solvers.

Adiar [41] is a redesign of the classical BDD algorithms such that they are optimal in the I/O model of Aggarwal and Vitter [1], based on ideas from Lars

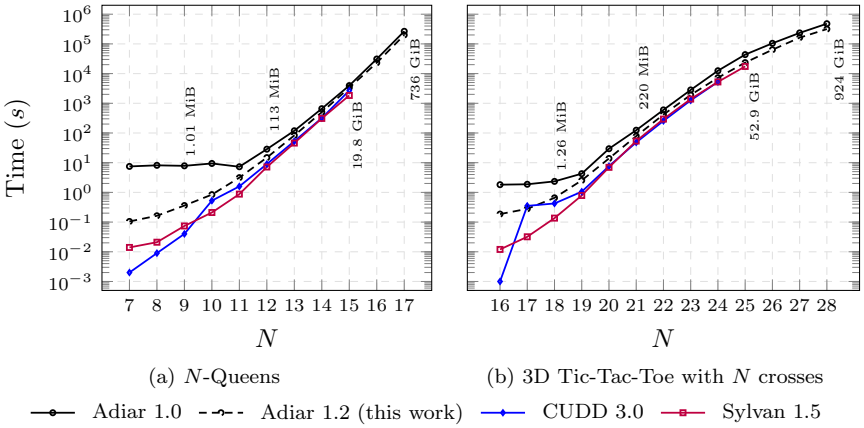



Figure 1: Running time solving combinatorial BDD benchmarks. Some instances are labelled with the size of the largest BDD constructed to solve them.

Arge [4]. As shown in Fig. 1, this enables Adiar to handle BDDs beyond the limits of main memory with only a minor slowdown in performance, unlike conventional BDD implementations. Adiar is implemented on top of the TPIE library [29, 42], which provides external memory sorting algorithms, file access, and priority queues, while making management of I/O transparent to the programmer. These external memory data structures work by loading one or more blocks from files on disk into internal memory and manipulating the elements within these blocks before storing them again on the disk. Their I/O-efficiency stems from a carefully designed order in which these blocks are retrieved, manipulated, and stored. Yet, initializing the internal memory in preparation to do so is itself costly – especially if purely using internal memory would have sufficed. This is evident in Fig. 1 (cf. Section 4.3 for more details) where Adiar's performance is several orders of magnitude worse than conventional BDD packages for smaller instance sizes. In fact, Adiar's performance decreases when the amount of internal memory increases.

This shortcoming is not desirable for a BDD package: while our research focuses on enabling large-scale BDD manipulation, end users should not have to consider whether their BDDs will be large enough to benefit from Adiar. Solving this also paves the way for Adiar to include complex BDD operations where conventional implementations recurse on intermediate results, e.g. *Multi-variable Quantification*, *Relational Product*, and *Variable Reordering*. To implement the same, Adiar has to run multiple sweeps. Yet, each of these sweeps suffer when they unecessarily use external memory data structures. Hence, it is vital to overcome this shortcomming, to ensure that an I/O-efficient implementations of these complex BDD operations will also be usable in practice.

The linearithmic I/O- and time-complexity of Adiar's algorithms also ap-

plies to the lower levels of the memory hierarchy, i.e. between the cache and RAM. Hence, there is no reason to believe that the bad performance for smaller instances is inherently due to the algorithms themselves; if they used an internal memory variant of all auxiliary data structures, then Adiar ought to perform well for much smaller instances. In fact, this begs the question: while we have investigated the applicability of these algorithms at a large-scale in [41], how can they seamlessly handle both small and large BDDs efficiently?

We argue that simple solutions are unsatisfactory: A first idea would be to start running classical, depth-first BDD algorithms until main memory is exhausted. In that case, the computation is aborted and restarted with external memory algorithms. But, this strategy doubles the running time. While it would work well for small instances, the slowdown for large instances would be unacceptable. Alternatively, both variants could be run in parallel. But, this would halve the amount of available memory and again slow down large instances.

A second idea would be to start running Adiar's I/O-efficient algorithms with an implementation of all auxiliary data structures in internal memory. In this case, if memory is exhausted, the data coudl be copied to disk, and the computation could be resumed with external memory. This could be implemented neatly with the *state pattern*: a wrapper switches transparently to the external memory variant when needed. Yet, moving elements from one sorted data structure to another requires at least linear time. Even worse, such a wrapper adds an expensive level of indirection and hinders the compiler in inlining and optimising, since the actual data structure is unknown at compile-time.

Instead, we propose to use the faster, internal-memory version of Adiar's algorithms only when it is guaranteed to succeed. This avoids re-computations, duplicate storage, as well as the costs of indirection. The main research question is how to predict a sound upper bound on the memory required for a BDD operation, and what information to store to compute these bounds efficiently.

## 1.1 Contributions

In Section 3, we introduce the notion of an $i$-level cut for Directed Acyclic Graphs (DAGs). Essentially, the shape of these cuts is constricted to span at most $i$ levels of the given DAG. Previous results in [22] show that for $i \geq 4$ the problem of computing the maximum $i$-level cut is NP-complete. We show that for $i \in \{1, 2\}$ this problem is still computable in polynomial time. These polynomial-time algorithms can be implemented using a linearithmic amount of time and I/Os. But instead, we use over-approximations of these cuts. As described in Section 3.4, their computation can be piggybacked on existing BDD algorithms, which is considerably cheaper: for 1-level cuts, this only adds a 1% linear-time overhead and does not increase the number of I/O operations.

Investigating the structure of BDDs from the perspective of $i$-level cuts for $i \in \{1, 2\}$ in Section 3.1 and 3.2, we obtain sound upper bounds on the maximum $i$-level cuts of a BDD operation's output, purely based on the maximum $i$-level cut of its inputs. Using these upper bounds, Adiar can decide in constant

time whether to run the next algorithm with internal or external memory data structures. Here, only one variant is run, all memory is dedicated to it, and the exact type of the auxiliary data structures are available to the compiler.

Our experiments in Section 4 show that it is a good strategy to compute the 1-level cuts, and to use them to infer an upper bound on the 2-level cuts. This strategy is sufficient to address Adiar's performance issues for the moderate-sized instances while also requiring the least computational overhead. As Fig. 1 shows, adding these cuts to Adiar with version 1.2 removes the overhead introduced by initializing TPIE's external memory data structures and so greatly improves Adiar's performance. For example, to verify the correctness of the small and moderate instances of the EPFL combinational benchmark circuits [2], the use of $i$-level cuts decreases the running time from 56.5 hours down to 4.0 hours.

# 2 Preliminaries

## 2.1 Graph and Cuts

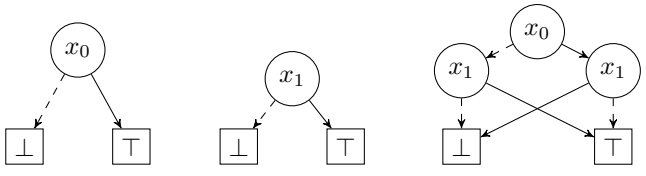
A directed graph is a tuple $(V, A)$ where $V$ is a finite set of vertices and $A \subseteq V \times V$ a set of arcs between vertices. The set of incoming arcs to a vertex $v \in V$ is $in(v) = A \cap (V \times \{v\})$, its outgoing arcs are $out(v) = A \cap (\{v\} \times V)$, and $v$ is a *source* if its indegree $|in(v)| = 0$ and a *sink* if its outdegree $|out(v)| = 0$.

A cut of a directed graph $(V, A)$ is a partitioning $(S, T)$ of $V$ such that $S \cup T = V$ and $S \cap T = \emptyset$. Given a weight function $w : A \to \mathbb{R}$ the weighted maximum cut problem is to find a cut $(S, T)$ such that $\sum_{a \in S \times T \cap A} w(a)$ is maximal, i.e. where the total weight of arcs crossing from some vertex in $S$ to one in $T$ is maximised. Without decreasing the weight of a cut, one may assume that all sources in $V$ are part of the partition $S$ and all sinks are part of $T$. The maximum cut problem is NP-complete for directed graphs [31] and restricting the problem to directed acyclic graphs (DAGs) does not decrease the problem's complexity [22].

If the weight function $w$ merely counts the number of arcs that cross a cut, i.e. $\forall a \in A : w(a) = 1$, the problem above reduces to the *unweighted* maximum cut problem where a cut's weight and size are interchangeable.

## 2.2 Binary Decision Diagrams

A Binary Decision Diagram (BDD) [8], as depicted in Fig. 2, is a DAG $(V, A)$ that represents an $n$-ary Boolean function. It has a single source vertex $r \in V$, usually referred to as the *root*, and up to two sinks for the Boolean values $\mathbb{B} = \{\bot, \top\}$, usually referred to as *terminals* or *leaves*. Each non-sink vertex $v \in V \setminus \mathbb{B}$ is referred to as a BDD *node* and is associated with an input variable $x_i \in \{x_0, x_1, \ldots, x_{n-1}\}$ where $label(v) = i$. Each arc is associated with a Boolean value, i.e. $A \subseteq V \times \mathbb{B} \times V$ (written as $v \xrightarrow{b} v'$ for a $(v, b, v') \in A$), such that each BDD node $v$ represents a binary choice on its input variable. That is,




(a) $x_0$ (b) $x_1$ (c) $x_0 \oplus x_1$

Figure 2: Examples of Reduced Ordered Binary Decision Diagrams. Terminals are drawn as boxes with the Boolean value and BDD nodes as circles with the decision variable. *Low* edges are drawn dashed while *high* edges are solid.

$out(v) = \{v \xrightarrow{\bot} v', v \xrightarrow{\top} v''\}$, reflecting $x_i$ being assigned the value $\bot$, resp. $\top$. Here, $v'$ is said to be $v$'s *low* child while $v''$ is its *high* child.

An Ordered Binary Decision Diagram (OBDD) restricts the DAG such that all paths follow some total variable ordering $\pi$: for every arc $v_1 \to v_2$ between two distinct nodes $v_1$ and $v_2$, $label(v_1)$ must precede $label(v_2)$ according to the order $\pi$. A Reduced Ordered Binary Decision Diagram (ROBDD) further adds the restriction that for each node $v$ where $out(v) = \{v \xrightarrow{\bot} v', v \xrightarrow{\top} v''\}$, (1) $v' \neq v''$ and (2) there exists no other node $u \in V$ such that $label(v) = label(u)$ and $out(u) = \{u \xrightarrow{\bot} v', u \xrightarrow{\top} v''\}$. The first requirement removes *don't care* nodes while the second removes *duplicates*. Assuming a fixed variable ordering $\pi$, an ROBDD is a canonical representation of the Boolean function it represents [8]. Without loss of generality, we will assume $\pi$ is the identity.

This graph-based representation allows one to indirectly manipulate Boolean formulae by instead manipulating the corresponding DAGs. For simplicity, we will focus on the Apply operation in this paper, but our results can be generalised to other operations. Apply computes the ROBDD for $f \odot g$ given ROBDDs for $f$ and $g$ and a binary operator $\odot : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$. This is done with a product construction of the two DAGs, starting from the pair $(r_f, r_g)$ of the roots of $f$ and $g$. If terminals $b_f$ from $f$ and $b_g$ from $g$ are paired then the resulting terminal is $b_f \odot b_g$. Otherwise, when nodes $v_f$ from $f$ and $v_g$ from $g$ are paired, a new BDD node is created with label $\ell = \min(label(v_f), label(v_g))$, and its low and high child are computed recursively from pairs $(v'_f, v'_g)$. For the low child, $v'_f$ is $v_f.low$ if $label(v_f) = \ell$ and $v_f$ otherwise; $v'_g$ is defined symmetrically. The recursive tuple for the high child is defined similarly.

### 2.2.1 Zero-suppressed Decision Diagrams

A Zero-suppressed Decision Diagram (ZDD) [27] is a variation of BDDs where the first reduction rule is changed: a node $v$ for the variable $label(v)$ with $out(v) = \{v \xrightarrow{\bot} v', v \xrightarrow{\top} v''\}$ is not suppressed if $v$ is a *don't care* node, i.e. if $v' = v''$, but rather if it assigns the variable $label(v)$ to $\bot$, i.e. if $v'' = \bot$. This makes ZDDs a better choice in practice than BDDs to represent functions $f$ where its on-set, $\{\vec{x} \mid f(\vec{x}) = \top\}$, is sparse.

The basic notions behind the BDD algorithms persist when translated to

ZDDs, but it is important for correctness that the ZDD operations account for the shape of the suppressed nodes. For example, the *union* operation needs to replace recursion requests for $(v_f, v_g)$ with $(v_f, \bot)$ if $label(v_f) < label(v_g)$ and with $(\bot, v_g)$ if $label(v_f) > label(v_g)$.

### 2.2.2 Levelised Algorithms in Adiar

BDDs and ZDDs are usually manipulated with recursive algorithms that use two hash tables: one for memoisation and another to enforce the two reduction rules [7, 28]. Lars Arge noted in [4, 5] that this approach is not efficient in the I/O-model of Aggarwal and Vitter [1]. He proposed to address this issue by processing all BDDs iteratively level by level with the time-forward processing technique [13, 25]: recursive calls are not executed at the time of issuing the request but are instead deferred with one or more priority queues until the necessary elements are encountered in the inputs. In [41], we implemented this approach in the BDD package Adiar. Furthermore, with version 1.1 we have extended this approach to ZDDs [38].

In Adiar, each decision diagram is represented as a sequence of its BDD nodes. Each BDD node is uniquely identifiable by the pair $(\ell, i)$ of its level $\ell$, i.e. its variable label, and its level-index $i$. And so, each BDD node can be represented as a triple of its own and its two children's unique identifiers (uids). The entire sequence of BDD nodes follows a level by level ordering of nodes which is equivalent to a lexicographical sorting on their uid. For example, the three BDDs in Fig. 2 are stored on disk as the lists in Fig. 3.

2a: $[\ ((0,0), \bot, \top)\ ]$
2b: $[\ ((1,0), \bot, \top)\ ]$
2c: $[\ ((0,0),(1,0),(1,1))\ ,\ ((1,0), \bot, \top)\ ,\ ((1,1), \top, \bot)\ ]$

Figure 3: In-order representation of BDDs of Fig. 2

The conventional recursive algorithms traverse the input (and the output) with random-access as dictated by the call stack. Adiar replaces this stack with a priority queue that is sorted such that it is synchronised with a sequential traversal through the input(s). Specifically, the recursion requests $s \to t$ from a BDD node $s$ to $t$ is sorted on the target $t$ – this way the requests for $t$ are at the top of the priority queue when $t$ is reached in the input. For example, after processing the root $(0,0)$ of the BDD in Fig. 2c, the priority queue includes the arcs $(0,0) \xrightarrow{\bot} (1,0)$ and $(0,0) \xrightarrow{\top} (1,1)$, in that order. Notice, this is exactly in the same order as the sequence of nodes in Fig. 3. Essentially, this priority queue maintains the yet unresolved parts of the recursion tree $(V', A')$ throughout a level by level top-down sweep. Yet, since the ordering of the priority queue groups together requests for the same $t$, the graph $(V', A')$ is not a tree but a DAG.

For BDD algorithms that produce an output BDD, e.g. the Apply algorithm, Adiar first constructs $(V', A')$ level by level. When the output BDD node $t \in V'$

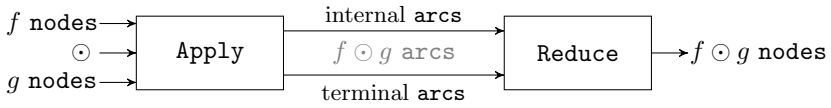



Figure 4: The Apply–Reduce pipeline in Adiar

is created from nodes $v_f \in V_f$ and $v_g \in V_g$, the top of the priority queues provides all ingoing arcs, which are placed in the output. Outgoing arcs to a terminal, $out(t) \cap (V' \times \mathbb{B} \times \mathbb{B})$, are also immediately placed in a separate output. On the other hand, recursion requests from $t$ to its yet unresolved non-terminal children, $out(t) \setminus (V' \times \mathbb{B} \times \mathbb{B})$, have to be processed later. To do so, these unresolved arcs are put back into the priority queue as arcs

$$(t \xrightarrow{b} (v'_f, v'_g)) \in V' \times \mathbb{B} \times (V_f \times V_g) \ ,$$

where the arc's target is the tuple of input nodes $v'_f \in V_f$ and $v'_g \in V_g$. This essentially makes the priority queue contain all the yet unresolved arcs of the output. For example, when using Apply to produce Fig. 2c from Fig. 2a and 2b, the root node of the output is resolved to have uid $(0,0)$ and the priority queue contains arcs $(0,0) \xrightarrow{\bot} (\bot, (1,0))$ and $(0,0) \xrightarrow{\top} (\top, (1,0))$. Both of these arcs are then later resolved, creating the nodes $(1,0)$ and $(1,1)$, respectively.

Yet, these *top-down sweeps* of Adiar produce sequences of arcs rather than nodes. Furthermore, the DAG $(V', A')$ is not necessarily a reduced OBDD. Hence, as shown in Fig. 4, Adiar follows up on the above top-down sweep with a *bottom-up sweep* that I/O-efficiently recreates Bryant's original Reduce algorithm in [8]. Here, a priority queue forwards the uid of $t'$ that is the result from applying the reduction rules to a BDD node $t$ in $(V', A')$ to the to-be reduced parents $s$ of $t$. These parents are immediately available by a sequential reading of $(V', A')$ since $in(t)$ was output together within the prior top-down sweep. Both reduction rules are applied by accumulating all nodes at level $j$ from the arcs in the priority queue, filtering out *don't care* nodes, sorting the remaining nodes such that duplicates come in succession and can be eliminated efficiently, and finally passing the necessary information to their parents via the priority queue.

# 3 Levelised Cuts of a Directed Acyclic Graph

Any DAG can be divided in one or more ways into several *levels*, where all vertices at a given level only have outgoing arcs to vertices at later levels.

**Definition 1.** *Given a DAG $(V, A)$ a levelisation of vertices in $V$ is a function $\mathcal{L} : V \to \mathbb{N} \cup \{\infty\}$ such that for any two vertices $v, v' \in V$, if there exists an arc $v \to v'$ in $A$ then $\mathcal{L}(v) < \mathcal{L}(v')$.*

Intuitively, $\mathcal{L}$ is a labeling of vertices $v \in V$ that respects a topological ordering of $V$. Since $(V, A)$ is a DAG, such a topological ordering always exists

and hence such an $\mathcal{L}$ must also always exist. Specifically, Specifically, let $\pi_V$ in be the longest path in $(V, A)$ (which must be from some source $s \in V$ to a sink $t \in V$) and $\pi_v$ be the longest path any given $v \in V$ to any sink $t \in V$, then $\mathcal{L}(v)$ can be defined to be the difference of their lengths, i.e. $|\pi_V| - |\pi_v|$.

Given a DAG and a levelisation $\mathcal{L}$, we can restrict the freedom of a cut to be constricted within a small window with respect to $\mathcal{L}$. Fig. 5 provides a visual depiction of the following definition.

**Definition 2.** *An* $i$-level cut *for* $i \geq 1$ *is a cut* $(S, T)$ *of a DAG* $(V, A)$ *with levelisation* $\mathcal{L}$ *for which there exists a* $j \in \mathbb{N}$ *such that* $\mathcal{L}(s) < j + i$ *for all* $s \in S$ *and* $\mathcal{L}(t) > j$ *for all* $t \in T$.

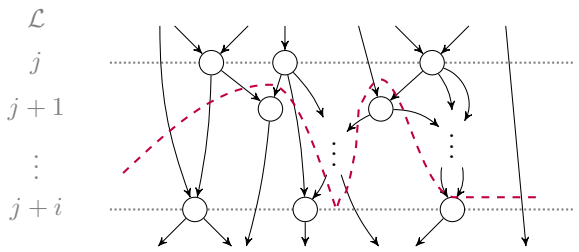



Figure 5: Visualisation of an $i$-level cut.

As will become apparent later, deriving the $i$-level cut with maximum weight for $i \in \{1, 2\}$ will be of special interest. Fig. 6 shows two 1-level cuts and three 2-level cuts in the BDD for the exclusive-or of the two variables $x_0$ and $x_1$. A 1-level cut is by definition a cut between two adjacent levels whereas a 2-level cut allows nodes on level $j + 1$ to be either in $S$ or in $T$. In Fig. 6, both the maximum 1-level and 2-level cuts have size 4.

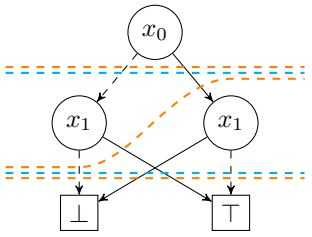



Figure 6: 1-level (cyan) and 2-level (orange) cuts in the $x_0 \oplus x_1$ BDD.

**Proposition 3.** *The maximum* 1*-level cut in a DAG* $(V, A)$ *with levelisation* $\mathcal{L}$ *is computable in polynomial time.*

*Proof.* For a specific $j \in \mathcal{L}(V)$ we can compute the size of the 1-level cut at $j$ in $O(A)$ time by computing the sum of $w((s, t))$ over all arcs $(s, t) \in A$ where $\mathcal{L}(s) \leq j$ and $\mathcal{L}(t) > j$. This cut is by definition unique for $j$ and hence

maximal. Repeating this for each $j \in \mathcal{L}(V)$ we obtain the maximum 1-level cut of the entire DAG in $O(|\mathcal{L}(V)| \cdot |A|) = O(|V| \cdot |A|)$ time. □

**Proposition 4.** *The maximum 2-level cut in a DAG $(V, A)$ with levelisation $\mathcal{L}$ is computable in polynomial time.*

*Proof.* Given a level $j \in \mathcal{L}(V)$, any 2-level cut for $j-1$ has all vertices $v \in V$ with $\mathcal{L}(v) \neq j$ fixed to be in $S$ or in $T$. That is, only vertices $v$ where $\mathcal{L}(v) = j$ may be part of either $S$ or of $T$. A vertex $v$ at level $j$ can greedily be placed in $S$ if $\sum_{a \in out(v)} w(a) < \sum_{a \in in(v)} w(a)$ and in $T$ otherwise. This greedy decision procedure runs in $O(|A|)$ time for each level, resulting in an $O(|\mathcal{L}(V)| \cdot |A|) = O(|V| \cdot |A|)$ total running time. □

Lampis, Kaouri, and Mitsou [22] prove NP-completeness for computing the maximum cut of a DAG by a reduction from the *not-all-equal SAT problem* (NAE3SAT) to a DAG with 5 levels. That is, they prove NP-completeness for computing the size of the maximum $i$-level cut for $i \geq 4$. This still leaves the complexity of the maximum $i$-level cut for $i = 3$ as an open problem.

## 3.1 Maximum Levelised Cuts in BDD Manipulation

For an OBDD, represented by the DAG $(V, A)$, we will consider the levelisation function $\mathcal{L}_{OBDD}$ where all nodes with the same label are on the same level.

$$\mathcal{L}_{OBDD}(v) \triangleq \begin{cases} label(v) & \text{if } v \notin \mathbb{B} \\ \infty & \text{if } v \in \mathbb{B} \end{cases}$$

For a BDD $f$ with the DAG $(V, A)$, let $N_f \triangleq |V \setminus \mathbb{B}|$ be the number of internal nodes in $V$. Let $C_{i:f}$ denote the size of the unweighted maximum $i$-level cut in $(V, A)$; in Section 3.2 we will consider weighted maximum cuts, where one or more terminals are ignored. Finally, we introduce the arc $(-\infty) \to r_f$ to the root. This simplifies the results that follow since $in(v) \neq \emptyset$ for all $v \in V$.

**Lemma 5.** *The maximum cut of a multi-rooted decision diagram $(V, A)$ is less than or equals to $N + r$ where $N = |V \setminus \mathbb{B}|$ is the number of internal nodes and $r \geq 1$ is the number of roots.*

*Proof.* We will prove this by induction on the number of internal nodes, $N$.

For $N = 1$, the decision diagram must be a singly rooted DAG with a single node $v$ with two outgoing arcs to $\mathbb{B}$, e.g. a BDD for the function $x_i$. The largest cut is of size 2 which equals the desired bound.

Assume for $N'$ that any decision diagrams with $N'$ number of internal nodes and some $r'$ number of roots have a maximum cut with a cost of at most $N' + r'$. Consider a decision diagram $(V, A)$ with $N = N' + 1$ internal nodes and $r \geq 1$ number of roots. Let $v$ be one of the $r$ roots. After removing $v$, the resulting decision diagram $(V', A')$ has $r' = r + \delta_{|in(v.low)|=1} + \delta_{|in(v.high)|=1} - 1$ roots where $\delta$ is the indicator function. The number of internal nodes in $(V', A')$ is $N'$ and so the maximum size of its cut is by induction $N' + r'$.

We will now argue, that adding $v$ back into the DAG $(V', A')$ may not increase the cut by more than one. Notice, since each node in a decision diagram is binary, we may assume that ingoing arcs to a node $v'$ are only contributing to a cut if $|in(v')| > 2$. Hence, the arc $v \xrightarrow{\perp} v.low$ may only contribute to the maximum cut in $(V, A)$, if $|in(v.low)| > 2$. By definition, this means $v \xrightarrow{\perp} v.low$ may only contribute to the maximum cut, if $\delta_{|in(v.low)|} = 0$. Symmetrically, $\delta_{|in(v.high)|}$ accounts for whether this very arc may be removed from the cut or not. Since $\delta_{|in(v.low)|=1} + \delta_{|in(v.high)|=1} \leq 2$, we have $r' \leq r + 1$. That is, adding the two arcs of $v$ into $(V', A')$ may only add one arc to the maximum cut that is not associated with a root of the DAG and so $N' + r' \leq N' + r + 1 = N + r$ is an upper bound on the maximum cut of $(V, A)$ as desired. □

By applying this to a single BDD, we obtain the following simple upper bound on any maximum cut of its DAG.

**Theorem 6.** *The maximum cut of the BDD $f$ has a size of at most $N_f + 1$.*

This bound is tight for $i$-level cuts, as is evident from Fig. 6 where the size of the maximum ($i$-level) cut is 4. Yet, in general, one can obtain a better upper bound on the maximum $i$-level cut of the (unreduced) output of each BDD operation when the maximum $i$-level cut of the input is known.

**Theorem 7.** *For $i \in \{1, 2\}$, the maximum $i$-level cut of the (unreduced) output of Apply of $f$ and $g$ is at most $C_{i:f} \cdot C_{i:g}$.*

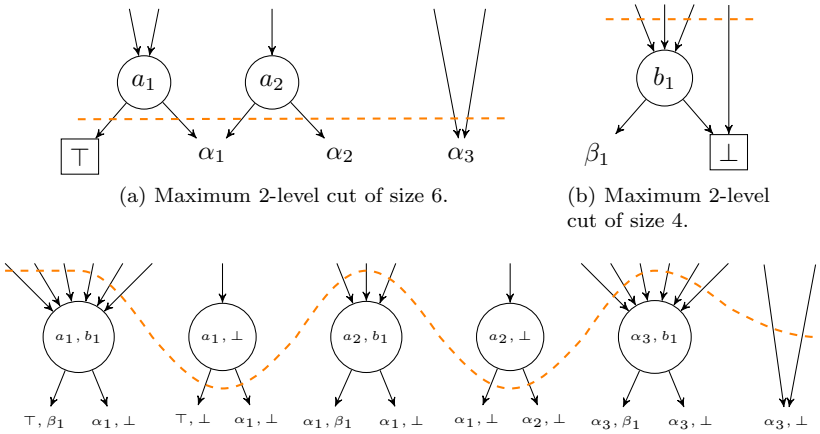
*Proof.* Let us only consider the more complex case of $i = 2$; the proof for $i = 1$ follows from the same line of thought.

Every node of the output represents a tuple $(v_f, v_g)$ where $v_f$, resp. $v_g$, is an internal node of $f$, resp. $g$, or is one of the terminals $\mathbb{B} = \{\perp, \top\}$. An example of this situation is shown in Fig. 7. The node $(v_f, v_g)$ contributes with $\max(|in((v_f, v_g))|, |out((v_f, v_g))|)$ to the maximum 2-level cut at that level. Since it is a BDD node, $|out((v_f, v_g))| = 2$. We have that $|in((v_f, v_g))| \leq |in(v_f)| \cdot |in(v_g)|$ since all combinations of in-going arcs may potentially exist and lead to this product of $v_f$ and $v_g$. Expanding on this, we obtain

$$\begin{aligned} |in((v_f, v_g))| &\leq |in(v_f)| \cdot |in(v_g)| \\ &\leq \max(|in(v_f)|, |out(v_f)|) \cdot \max(|in(v_g)|, |out(v_g)|) \ . \end{aligned}$$

That is, the maximum 2-level cut for a level is less than or equal to the product of the maximum 2-level cuts of the input at the same level. Taking the maximum 2-level cut across all levels we obtain the final product of $C_{2:f}$ and $C_{2:g}$. □

The bounds in Thm. 7 are better than what can be derived from Thm. 6 since $C_{i:f}$ and $C_{i:g}$ are themselves cuts and hence their product must be at most the bound based on the possible number of nodes. They are also tight: the maximum $i$-level cut for $i \in \{1, 2\}$ of the BDDs for the variables $x_0$ and $x_1$ in Fig 2a and 2b both have size 2 while the BDD for the exclusive-or of them in Fig. 2c has, as shown in Fig. 6, a maximum $i$-level cut of size 4.




(a) Maximum 2-level cut of size 6.

(b) Maximum 2-level cut of size 4.

(c) 7a × 7b's maximum 2-level cut of size $21 \leq 6 \cdot 4$.

Figure 7: Relation between the maximal 2-level cut of two BDDs' internal arcs and the maximum 2-level cut of their product.

Since the maximum 1-level cut also bounds the number of outgoing arcs of all nodes on each level, one can derive an upper bound on the output's width. That is, based on Thm. 7 we can obtain the following interesting result.

**Corollary 8.** *The* width *of Apply's output is less than or equal to* $\frac{1}{2} \cdot C_{1:f} \cdot C_{1:g}$.

*Proof.* The $\frac{1}{2}$ compensates for the outdegree of each BDD node. □

This is only an upper bound, as half of the arcs that cross the widest level are also counted. Yet, it is tight, as Fig. 6 has a maximum $i$-level cut of size 4 and a width of 2.

Thm. 7 is of course only an over-approximation. The gap between the upper bound and the actual maximum $i$-level cut arises because Thm. 7 does not account for pairs $(v_f, v_g)$, where node $v_f$ sits above $f$'s maximum 2-level cut and $v_g$ sits below $g$'s maximum 2-level cut, and vice versa. In this case, outgoing arcs of $v_f$ are paired with ingoing arcs of $v_g$, even though this would be strictly larger than the arcs of their product. Furthermore, similar to Thm. 6, this bound does not account for arcs that cannot be paired as they reflect conflicting assignments to one or more input variables. For example, in the case where the out-degree is greater for both nodes, the above bound mistakenly pairs the low arcs with the high arcs and vice versa.

## 3.2 Improving Bounds by Accounting for Terminal Arcs

Some of the imprecision in the over-approximation of Thm. 7 highlighted above can partially be addressed by explicitly accounting for the arcs to each terminal. For $B \subseteq \mathbb{B}$, let $w_B$ be the weight function that only cares for arcs to internal

BDD nodes and to the terminals in $B$.

$$w_B(s \xrightarrow{b} t) = \begin{cases} 1 & \text{if } t \in V \setminus \mathbb{B} \text{ or } t \in B \\ 0 & \text{otherwise} \end{cases} .$$

Let $C^B_{i:f}$ be the maximum $i$-level cut of $f$ with respect to $\mathcal{L}_{OBDD}$ and $w_B$.

The constant hidden within the $O(|V| \cdot |A|)$ running time of the algorithm in the proof of Prop. 3 is smaller than the one in the proof of Prop. 4. Hence, the following slight over-approximations of $C^B_{2:f}$ given $C^B_{1:f}$ may be useful.

**Lemma 9.** *The maximum 2-level cut $C^\emptyset_{2:f}$ is less than or equals to $\frac{3}{2} \cdot C^\emptyset_{1:f}$.*

*Proof.* $C^\emptyset_{1:f}$ is an upper bound on the number of ingoing arcs to nodes on level $j+1$ for any $j$. This places the BDD nodes $v$ with $\mathcal{L}(v) = j+1$ in the $S$ partition of the 1-level cut. The only case where such a $v$ should be moved to the $S$ partition for the maximum 2-level cut at level $j$ is if $|in(v)| = 1$ and $|out(v)| = 2$ in the subgraph only consisting of internal arcs. Since $C^\emptyset_{1:f}$ is also an upper bound on the number of outgoing arcs then at most $C^\emptyset_{1:f}/2$ nodes at level $j$ may be moved to $S$ to then count their $C^\emptyset_{1:f}$ outgoing arcs. This leaves $C^\emptyset_{1:f}/2$ ingoing arcs still to be counted. Combining both, we obtain the bound above. □

**Lemma 10.** *For $B \subseteq \mathbb{B}$, $C_{2:f}$ is at most $\frac{1}{2} \cdot C^\emptyset_{1:f} + C^B_{1:f}$.*

*Proof.* The $C^B_{1:f} - C^\emptyset_{1:f}$ is the number of arcs to terminals. The remaining $C^\emptyset_{1:f}$ may be arcs to a BDD node where up to $C^\emptyset_{1:f}/2$ can, as in Lem. 9, be moved to the other side of the cut to increase the 2-level cut with $C^\emptyset_{1:f}/2$. Simplifying $\frac{3}{2} + (C^B_{1:f} - C^\emptyset_{1:f})$ we obtain the desired bound. □

Finally, we can tighten the bound in Thm. 7 by making sure (1) not to unnecessarily pair terminals in $f$ with terminals in $g$ and (2) not to pair terminals from $f$ and $g$ with nodes of the other when said terminal shortcuts the operator.

**Lemma 11.** *The maximum 2-level cut of the (unreduced) output $f \odot g$ of Apply excluding arcs to terminals, $C^\emptyset_{2:f\odot g}$, is at most*

$$C^{B_{left(\odot)}}_{2:f} \cdot C^\emptyset_{2:g} + C^\emptyset_{2:f} \cdot C^{B_{right(\odot)}}_{2:g} - C^\emptyset_{2:f} \cdot C^\emptyset_{2:g} \ ,$$

*where $B_{left(\odot)}, B_{right(\odot)} \subseteq \mathbb{B}$ are the terminals that do not shortcut $\odot$.*

## 3.3 Maximum Levelised Cuts in ZDD Manipulation

The results in Section 3.1 and 3.2 are loosely yet subtly coupled to the reduction rules of BDDs. Specifically, Thm. 6 is applicable to ZDDs as-is but Thm. 7 and its derivatives provide unsound bounds for ZDDs. This is due to the fact that, unlike for BDDs, a suppressed ZDD node may re-emerge during a ZDD product

construction algorithm. For example in the case of the *union* operation, when processing a pair of nodes with two different levels, its high child becomes the product of a node $v$ in one ZDD and the $\bot$ terminal in the other – even if there was no arc to $\bot$ in the original two cuts for $f$ and $g$.

The solution is to introduce another special arc similar to $(-\infty) \to r_f$ which accounts for this specific case: if there are no arcs to $\bot$ to pair with, then the arc $(-\infty) \to \bot$ is counted as part of the input's cut. That is, all prior results for BDDs apply to ZDDs, assuming $C^B_{i:f}$ is replaced with $ZC^B_{i:f}$ defined to be

$$ZC^B_{i:f} = \begin{cases} C^B_{i:f} + 1 & \text{if } \bot \in B \text{ and } C^B_{i:f} = C^{B\setminus\{\bot\}}_{i:f} \\ C^B_{i:f} & \text{otherwise} \end{cases} .$$

## 3.4 Adding Levelised Cuts to Adiar's Algorithms

The description of Adiar in Section 2.2.2 leads to the following observations.

- The contents of the priority queues in the top-down Apply algorithms are always a 1-level or a 2-level cut of the input or of the output – possibly excluding arcs to one or both terminals.
- The contents of the priority queue in the bottom-up Reduce algorithm are always a 1-level cut of the input, excluding any arcs to terminals.

Specifically, the priority queues always contain an $i$-level cut $(S, T)$, where $S$ is the set of processed diagram nodes and $T$ is the set of yet unresolved diagram nodes. For example, the 2-level cuts depicted in Fig. 6 reflect the states of the top-down priority queue within the Apply to compute the exclusive-or of Fig. 2a and 2b to create Fig. 2c. In turn, the 1-level cuts in Fig. 6 are also the state of the bottom-up priority queue of the Reduce sweep that follows.

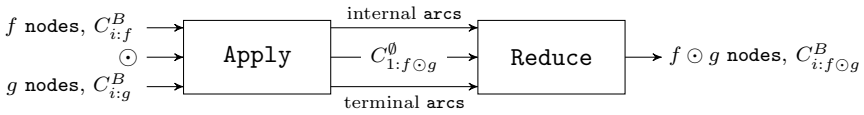
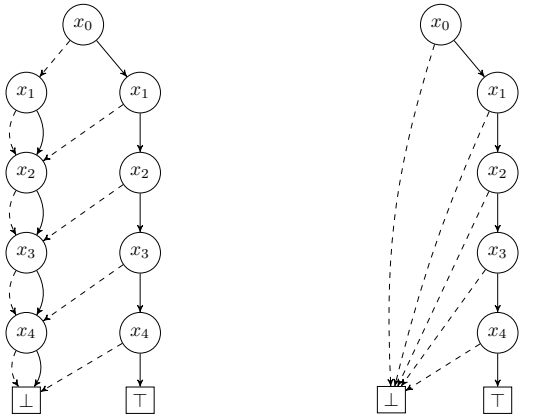



Figure 8: The Apply–Reduce pipeline in Adiar with $i$-level cuts.

Hence, the upper bounds on the 1 and 2-level cuts in Section 3.1, 3.2, and 3.3 are also upper bounds on the size of all auxiliary data structures. That is, upper bounds on the $i$-level cuts of the input can be used to derive a sound guarantee of whether the much faster internal memory variants can fit into memory. To only add a minimal overhead to the performance, computing these $i$-level cuts should be done as part of the preceding algorithm that created the very input. This extends the tandem in Fig. 4 as depicted in Fig. 8 with the $i$-level cuts necessary for the next algorithm.

What is left is to compute within each sweep an upper bound on these cuts.




(a) Unreduced OBDD (b) Reduced OBDD

Figure 9: Example of reduction increasing the 1 and 2-level maximum cut.

### 3.4.1 1-Level Cut within Top-down Sweeps

The priority queues of a top-down sweep only contain arcs between non-terminal nodes of its output. While their contents in general form a 2-level cut, the sweep also enumerates all 1-level cuts when it has finished processing one level, and is about to start processing the next. That is, the top-down algorithm that constructs the unreduced decision diagram $(V', A')$ for $f'$ can compute $C^{\emptyset}_{1:f'}$ in $O(|\mathcal{L}_{OBDD}(V')|)$ time by accumulating the maximum size of its own priority queue when switching from one level to another. The number of I/O operations is not affected at all.

### 3.4.2 $i$-Level Cuts within the Bottom-up Reduce

To compute the 1-level and 2-level cuts of the output during the Reduce algorithm, the algorithms in the proofs of Prop. 3 and 4 need to be incorporated. Since the Reduce algorithm works bottom-up, it cannot compute these cuts exactly: the bottom-up nature only allows information to flow from lower levels upwards while an exact result also requires information to be passed downwards. Specifically, Fig. 9 shows an unreduced BDD whose maximum 1 and 2-level cut is increased due to the reduction removing nodes above the cut. Both over-approximation algorithms below are tight since for the input in Fig. 9 they compute the exact result.

**Over-approximating the 1-level Cut.** Starting from the bottom, when processing a level $k \in \mathcal{L}_{OBDD}(V)$ we may over-approximate the 1-level cut $C^{B}_{1:f}$ for $B \subseteq \{\bot, \top\}$ at $j = k$ by summing the following four disjoint contributions.

1. After having obtained all outgoing arcs for unreduced nodes for level $k$, the priority queue only contains outgoing arcs from a level $\ell < k$ to a level

$\ell' > k$. All of these arcs (may) contribute to the cut.

2. After having obtained all outgoing arcs for level $k$, all yet unread arcs to terminals $b \in B$ are from some level $\ell < k$ and (may) contribute to the cut.

3. BDD nodes $v$ removed by the first reduction rule in favor of its reduced child $v'$ and $w_B(\_ \to v') = 1$ (may) contribute up to $|in(v')|$ arcs to the cut.

4. BDD nodes $v'$ that are output on level $k$ after merging duplicates (definitely) contribute with $w_B(v'.low) + w_B(v'.high)$ arcs to the cut.

1 and 2 can be obtained with some bookkeeping on the priority queue and the contents of the file containing arcs to terminals. 4 can be resolved when reduced nodes are pushed to the output. Yet, 3 cannot just use the immediate indegree of the removed node $v$ since, as in Fig. 9, it may be part of a longer chain of redundant nodes. Here, the actual contribution to the cut at level $j = k$ is the indegree to the entire chain ending in $v$. Due to the single bottom-up sweep style of the Reduce algorithm, the best we can do is to assume the worst and always count reduced arcs $s' \to t'$ where a node $v$ has been removed between $s'$ and $t'$ as part of the maximum cut.

**Over-approximating the 2-level Cut.** The above over-approximation of the 1-level cut can be extended to recreate the greedy algorithm from the proof of Prop. 4. Notice, the 1-level $(S, T)$ cut mentioned before places all nodes of level $j$ in $S$, whereas these nodes are free to be moved to $T$ in the 2-level cut for $j - 1$. Specifically, Part 4 should be changed such that $v'$ contributes with

$$\max(w_B(v'.low) + w_B(v'.high), |in(v')|) \ .$$

This requires knowing $|in(v')|$. The Reduce algorithm in [41] reads from a file containing the parents of an unreduced node $v$, so information about the reduced result $v'$ can be forwarded to its unreduced parents. Hence, one can accumulate the number of parents, $|in(v)|$. If $|in(v')|$ is not affected by the first reduction rule then this is an upper bound of $|in(v')|$. Otherwise, it still is sound in combination with the above over-counting to solve the 3$^{\text{rd}}$ type of contribution.

# 4 Experimental Evaluation

We have extended Adiar to incorporate the ideas presented in Section 3 to address the issues highlighted in Section 1. Each algorithm has been extended to compute sound upper bounds for the next phase. Based on these, each algorithm chooses during initialisation between running with TPIE's internal or external memory data structures. This choice is encapsulated within C++ templates,

which avoids introducing any costly indirection when using the auxiliary data structures since in both cases their type is already known to the compiler.

Section 3.4 motivates the following three levels of granularity:

- **#nodes:** Thm. 6 is used based on knowing the number of internal nodes in the input and deriving the trivial worst-case size of the output.
- **1-level:** Extends #nodes with Thm. 7. The $i$-level cuts are given by computing the 1-level cut with the proof of Prop. 3 as described in Section 3.4.2 and then applying Lem. 10 to obtain a bound on the 2-level cut.
- **2-level:** Extends the 1-level variant by computing 2-level cuts directly with the algorithm based on the proof of Prop. 4 in Section 3.4.2.

All three variants include the computation of 1-level cuts – even the #nodes one. This reduces the number of variables in our measurements. We have separately measured the slowdown introduced by computing 1-level cuts to be 1.0%.

## 4.1 Benchmarks

We have evaluated the quality of our modifications on the four benchmarks below that are publicly available at [36]. These were also used to measure the performance of Adiar 1.0 (BDDs) and 1.1 (ZDDs) in [38, 41]. The first benchmark is a circuit verification problem and the others are combinatorial problems.

- ***EPFL* Combinational Benchmark Suite [2].** The task is to check equivalence between an original hardware circuit (specification) and an optimised circuit (implementation). We construct BDDs for all output gates in both circuits, and check if they are equivalent. We focus on the 23 out of the 46 optimised circuits that Adiar could verify in [41]

  Input gates are encoded as a single variable, $x_i$, with a maximum 2-level cut of size 2.

- **Knight's Tour.** On an $N_r \times N_c$ chessboard, the set of all paths of a Knight is created by intersecting the valid transitions for each of the $N_r N_c$ time steps. The cut of each such ZDD constraint is $\sim 8 N_r N_c$. Then, each Hamiltonian constraint with cut size 4 is imposed onto this set [38].

- **$N$-Queens.** On an $N \times N$ chessboard, the constraints on placing queens are combined per row, based on a base case for each cell. Each row constraint is finally accumulated into the complete solution [21].

  For BDDs, each basic cell constraint has a cut size of $\sim 3N$, while for ZDDs it is only 3.

- **Tic-Tac-Toe.** Initially, a BDD or ZDD with cut size $\sim N$ is created to represent that $N$ crosses have been set within a $4 \times 4 \times 4$ cube. Then

for each of the 76 lines, a constraint is added to exclude any *non-draw* states [21].

Each such line constraint has a cut size of 4 with BDDs and 6 with ZDDs.

## 4.2 Tradeoff between Precision and Running Time

We have run all benchmarks on a consumer-grade laptop with one 2.6 GHz Intel i7-4720HQ processor, 8 GiB of RAM, 230 GiB of available SSD disk, running Fedora 36, and compiling code with GCC 12.2.1. For each of these 71 benchmark instances, Adiar has been given 128 MiB or 4 GiB of internal memory.

All combinatorial benchmarks use a unary operation at the end to count the number of solutions. Table 1 shows the average ratio between the predicted and actual maximum size of this operation's priority queue. As instances grow larger, the quality of the #nodes heuristic deteriorates for BDDs. On the other hand, the 1 and 2-level cut heuristics are at most off by a factor of 2. Hence, since the priority queue's maximum size is some 2-level cut, the algorithms in Section 3.4.2 are only over-approximating the actual maximum 2-level cut by a factor of 2. The result of this is that $i$-level cuts can safely identify that a BDD with $5.2 \cdot 10^7$ nodes (1.1 GiB) can be processed purely within 128 MiB of internal memory available. The precision of $i$-level cuts are worse for ZDDs, but still allow processing a ZDD with $4.3 \cdot 10^7$ nodes (978 MiB) with 128 MiB of memory.

This difference in precision affects the product construction algorithms, e.g. the Apply operation. Fig. 10 shows the amount of product constructions that each heuristic enables to run with internal memory data structures. Even when the average BDD was $10^7$ nodes (229 MiB) or larger, with $i$-level cuts at least 59.5% of all algorithms were run purely in 128 MiB of memory, whereas with #nodes sometimes none of them were. Yet, while there is a major difference between #nodes and 1-level cuts, going further to 2-level cuts only has a minor effect.

How often internal memory could be used is also reflected in Adiar's performance. Fig. 11 shows the difference in the running time between using $i$-level

Table 1: Geometric mean of the ratio between the predicted and the actual maximum size of the unary Count operation's priority queue. This average is also weighed by the input size to gauge the predictions' quality for larger BDDs.

| | BDD | | |
|---|---|---|---|
| | #nodes | 1-level | 2-level |
| Unweighted Avg. | 2.1% | 69.2% | 86.3% |
| Weighted Avg. | 0.1% | 76.5% | 77.4% |
| | ZDD | | |
| Unweighted Avg. | 15.2% | 47.8% | 67.0% |
| Weighted Avg. | 25.0% | 50.7% | 61.8% |

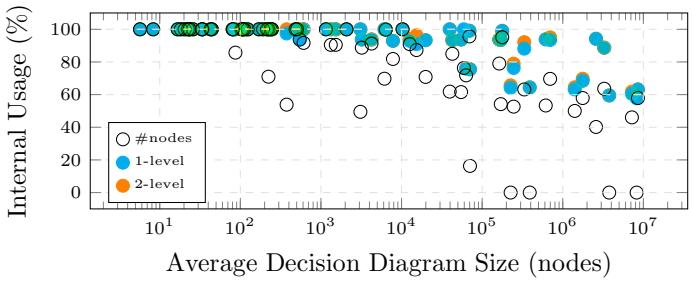
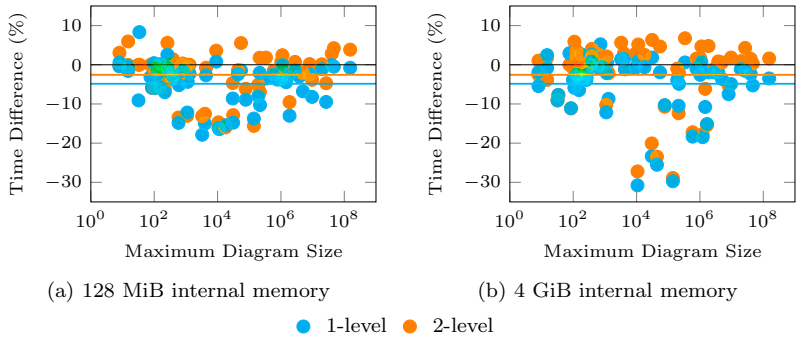



Figure 10: *Internal* vs. *external* memory usage for product constructions (128 MiB).




(a) 128 MiB internal memory (b) 4 GiB internal memory

Figure 11: Adiar with $i$-level cuts compared to #nodes (lower is better). Horizontal lines show the average difference in performance.

cuts and only using #nodes. All benchmarks runs were interleaved and repeated at least 8 times. The minimum measured running time is reported as it minimises any noise due to hardware and the operating system [12]. Since the #nodes version also includes the computation for the 1-level cuts but does not use them, any performance decrease in Fig. 11 for 1-level cuts is due to noise.

Using the geometric mean, 1-level cuts provide a 4.9% improvement over #nodes. Considering the 1.0% overhead for computing the 1-level cuts, this is a net improvement of 3.9%. More importantly, in a considerable amount of benchmarks, using $i$-level cuts improves the performance by more than 10%, sometimes by 30%. These are the benchmark instances where only $i$-level cuts can guarantee that all auxiliary data structures can fit within internal memory, yet the instances are still so small that there is a major overhead in initialising TPIE's external memory data structures.

The improvement in precision obtained by using 2-level cuts does not pay off in comparison to using 1-level cuts. On average, using 2-level cuts only improves the performance of using #nodes with 2.6%. That is, the additional cost of computing 2-level cuts outweighs the benefits of its added precision.

Adiar with $i$-level cuts did not slow down as internal memory was increased from 128 MiB to 4 GiB. That is, the precision of both these bounds – unlike #nodes – ensures that external memory data structures are only used when their initialisation cost is negligible. Hence, Adiar with 1-level cuts covers all our needs at the minimal computational cost and so is included in Adiar 1.2.

## 4.3 Impact of Introducing Cuts on Adiar's Running Time

In [38, 41] we measured the performance of Adiar 1.0 and 1.1 against the conventional BDD packages CUDD 3.0 [34] and Sylvan 1.5 [16]. In those experiments [37, 39], Sylvan was not using multi-threading and all experiments were run on machines with 384 GiB of RAM of which 300 GiB was given to the BDD package. To gauge the impact of using cuts, we now compare our previous measurements without cuts to new ones with cuts on the exact same hardware and settings. The results of our new measurements are available at [40].

With 300 GiB internal memory available, all three modified versions of Adiar essentially behave the same. Hence, in Fig. 1 (cf. Section 1) we show the best performance for all three versions on top of the data reported in [41]. Even on the largest benchmarks we see a performance increase by exploiting cuts. Most important is the increase in performance for the moderate-size instances where the initialisation of TPIE's external memory data structures are costly, e.g. $N$-Queens with $N < 11$ and Tic-Tac-Toe with $N < 19$. Based on the data in [38, 41] these instances of the combinatorial benchmarks are the ones where the largest constructed BDD or ZDD is smaller than $4.9 \cdot 10^6$ nodes (113 MiB).

Using the geometric mean, the time spent solving both the combinatorial and verification benchmarks decreased with Adiar 1.2 on average by 86.1% (with median 89.7%) in comparison to previous versions. For some instances this difference is even 99.9%. In fact, Adiar 1.2 is in some specific instances of the Tic-Tac-Toe benchmarks faster than CUDD. These are the very instances that are large enough for CUDD's first – and comparatively expensive – garbage collection to kick in and dominate its running time.

Verifying the EPFL benchmarks involves constructing a few BDDs that are larger than the 113 MiB bound mentioned above, but most BDDs are much smaller. As shown in Table 2, for the 16 EPFL circuits[1] that only generate BDDs smaller than 113 MiB, using cuts decreases the computation time on average by 92% (with median 92%). While Adiar v1.0 still took 56.5 hours to verify these 16 circuits, now with Adiar 1.2 it only takes 4.0 hours to do the same. These 52.5 hours are primarily saved within one of the 16 circuits. Specifically, using cuts has decreased the time to verify the `sin` circuit optimised for depth by 52.1 hours. Here, the average BDD size is 2.9 KiB, the largest BDD constructed is 25.5 MiB in size, and up to 42,462 BDDs are in use concurrently.

Despite this massive performance improvement with Adiar 1.2 due to our new technique, there is still a significant gap of 3.7 hours with CUDD and Sylvan on these 16 circuits. We attribute this to the fact that these benchmarks

[1] In [33] we incorrectly reported this as being 15 rather than 16 circuits of this size.

Table 2: Minimum Running Time to construct the EPFL benchmark circuits [2] optimized for depth (d) or size (s) together with its respective specification circuit. The variable order $\pi$ was either set to be based on a level/depth-first (LD) traversal of the circuit or the given input-order (I). Some timings are not provided due to a Memory Out (MO), a Time Out (TO), or the measurement has not been made (–). All timings for Adiar v1.0, CUDD, and Sylvan are from [41], except for `mem_ctrl` and `voter` with LD variable ordering.

| Circuit | | | BDD Size (MiB) | | Adiar v1.0 | Adiar v1.2 | CUDD | Sylvan |
|---|---|---|---|---|---|---|---|---|
| name | opt. | $\pi$ | Avg. | Max | Time (ms) | Time (ms) | Time (ms) | Time (ms) |
| adder | d | LD | 0.0028 | 0.0182 | 193552 | 8022 | 790 | 170 |
| | s | LD | 0.0030 | 0.0088 | 138116 | 5300 | 772 | 91 |
| arbiter | d | I | 0.0125 | 63.64 | 472784 | 73638 | 7664 | 24769 |
| cavlc | d+s | I | 0.0001 | 0.0022 | 29550 | 1943 | 2 | 8 |
| ctrl | d+s | I | 0.0000 | 0.0003 | 5173 | 461 | 0 | 2 |
| dec | d+s | I | 0.0001 | 0.0002 | 21305 | 1544 | 0 | 4 |
| i2c | d | I | 0.0001 | 0.0060 | 36637 | 2942 | 3 | 9 |
| | s | I | 0.0001 | 0.0060 | 36192 | 3290 | 3 | 9 |
| int2float | d | I | 0.0001 | 0.0035 | 8166 | 783 | 0 | 3 |
| | s | I | 0.0001 | 0.0035 | 15205 | 1224 | 0 | 4 |
| mem_ctrl | d | I | 3.9550 | 16571 | 400464754 | 357042302 | MO | TO |
| | s | I | 3.9226 | 16571 | 398777513 | 356500951 | MO | TO |
| | d | LD | 0.0713 | 34.94 | – | 199999 | 57728 | – |
| | s | LD | 0.1264 | 264.1 | – | 298615 | 97441 | – |
| priority | d | I | 0.0001 | 0.0029 | 30864 | 1861 | 2 | 10 |
| | s | I | 0.0003 | 0.0049 | 33321 | 2035 | 3 | 11 |
| router | d | I | 0.0001 | 0.0073 | 7526 | 545 | 0 | 3 |
| | s | I | 0.0001 | 0.0029 | 5644 | 569 | 0 | 2 |
| sin | d | LD | 0.0021 | 25.50 | 199739354 | 12268821 | 299403 | 226713 |
| | s | LD | 0.8787 | 25.43 | 2585946 | 1840623 | 465770 | 394675 |
| voter | d | I | 2.190 | 8241 | 25078751 | 16357661 | MO | 11191333 |
| | s | I | 0.4801 | 8241 | 8520173 | 5197230 | 2307858 | 2775903 |
| | d | LD | 1.044 | 4348 | – | 32637944 | 3950294 | – |
| | s | LD | 0.2477 | 249 | – | 1391096 | 295991 | – |

also include many computations on really tiny BDDs. Although we keep the auxiliary data structures in internal memory, the resulting BDDs are still stored on disk, even when they consist of only a few nodes.

# 5 Conclusion

We introduce the idea of a maximum $i$-level cut for DAGs that restricts the cut to be within a certain window. For $i \in \{1, 2\}$ the problem of computing the maximum $i$-level cut is polynomial-time computable. But, we have been able to piggyback a slight over-approximation with only a 1% linear overhead onto Adiar's I/O-efficient bottom-up Reduce operation.

An $i$-level cut captures the shape of Adiar's auxiliary data structures during the execution of its I/O-efficient time-forward processing algorithms. Hence, similar to how conventional recursive BDD algorithms have the size of their call stack linearly dependent on the depth of the input, the maximum 2-level cuts provide a sound upper bound on the memory used during Adiar's computation. Using this, Adiar 1.2 can deduce soundly whether using exclusively internal memory is possible, increasing its performance in those cases. Doing so decreases computation time for moderate-size instances up to 99.9% and on average by 86.1% (with median 89.7%).

## 5.1 Related and Future Work

Many approaches tried to achieve large-scale BDD manipulation with distributed memory algorithms, some based on breadth-first algorithms, e.g. [21, 26, 35, 43]. Yet, none of these approaches obtained a satisfactory performance. The speedup obtained by a multicore implementation [16] relies on parallel depth-first algorithms using concurrent hash tables, which doesn't scale to external memory.

CAL [32] (based on a breadth-first approach [6, 30]) is to the best of our knowledge the only other BDD package designed to process large BDDs on a single machine. CAL is I/O efficient, assuming that a single BDD level fits into main memory; the I/O efficiency of Adiar does not depend on this assumption. Similar to Adiar, CAL suffers from bad performance for small instances. To deal with this, CAL switches to the classical recursive depth-first algorithms when all the given input BDDs contain fewer than $2^{19}$ nodes (15 MiB). As far as we can tell, CAL's threshold is purely based on experimental results of performance and without any guarantees of soundness. That is, the output may potentially exceed main memory despite all inputs being smaller than $2^{19}$ nodes, which would slow it down significantly due to random-access. For BDDs smaller than CAL's threshold of $2^{19}$ nodes, Adiar 1.2 with $i$-level cuts could run almost all of our experiments with auxiliary data structures purely in internal memory.

Yet, as is evident in Fig. 1, when dealing with decision diagrams smaller than 44.000 nodes (1 MiB), there is still a considerable gap between Adiar's performance and conventional depth-first based BDD packages (see also end of Sec. 4.3). Apparently, we have reached a lower bound on the BDD size for which time-forward processing on external memory is efficient. Solving this would require an entirely different approach: one that can efficiently and seamlessly combine BDDs stored in internal memory with BDDs stored in external memory.

## 5.2 Applicability Beyond Decision Diagrams

Our idea is generalisable to all time-forward processing algorithms: the contents of the priority queues are at any point in time a 2-level cut with respect to the input and/or output DAG. Hence, one can bound the algorithm's memory usage if one can compute a levelisation function and the 1-level cuts of the inputs.

A levelisation function is derivable with the preprocessing step in [19] and the cut sizes can be computed with an I/O-efficient version of the greedy algorithm presented in this paper. Yet for our approach to be useful in practice, one has to identify a levelisation function that best captures the structure of the DAG in relation to the succeeding algorithms and where both the computation of the levelisation and the 1-level cut can be computed with only a negligible overhead – preferably within the other algorithms.

## Acknowledgements

We want to thank Anna Blume Jakobsen for her help implementing the use of $i$-level cuts in Adiar and Kristoffer Arnsfelt Hansen for his input on the computational complexity of these cuts. Finally, thanks to the Centre for Scientific Computing, Aarhus, (phys.au.dk/forskning/cscaa/) for running our benchmarks.

## References

[1] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL combinational benchmark suite. In *24th International Workshop on Logic & Synthesis*, 2015.

[3] Elvio Gilberto Amparore, Susanna Donatelli, and Francesco Gallà. starMC: an automata based CTL* model checker. *PeerJ Comput. Sci.*, 8:e823, 2022.

[4] Lars Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *6th International Symposium on Algorithms and Computations (ISAAC)*, volume 1004 of *Lecture Notes in Computer Science*, pages 82–91, 1995.

[5] Lars Arge. The I/O-complexity of ordered binary-decision diagram. In *BRICS RS preprint series*, volume 29. Department of Computer Science, University of Aarhus, 1996.

[6] Pranav Ashar and Matthew Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 622–627. IEEE Computer Society Press, 1994.

[7] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th Design Automation Conference (DAC)*, pages 40–45. Association for Computing Machinery, 1990.

[8] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[9] Randal E. Bryant, Armin Biere, and Marijn J. H. Heule. Clausal proofs for pseudo-Boolean reasoning. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–461. Springer, 2022.

[10] Randal E. Bryant and Marijn J. H. Heule. Dual proof generation for quantified Boolean formulas with a BDD-based solver. In *Automated Deduction – CADE 28*, pages 433–449. Springer, 2021.

[11] Randal E. Bryant and Marijn J. H. Heule. Generating extended resolution proofs with a BDD-based sat solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12651 of *Lecture Notes in Computer Science*, pages 76–93. Springer, 2021.

[12] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. arXiv, 2016.

[13] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pages 139—-149. Society for Industrial and Applied Mathematics, 1995.

[14] Gianfranco Ciardo, Andrew S. Miner, and Min Wan. Advanced features in SMART: the stochastic model checking analyzer for reliability and timing. *SIGMETRICS Perform. Evaluation Rev.*, 36(4):58–63, 2009.

[15] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:410–425, 2000.

[16] Tom Van Dijk and Jaco Van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19:675–696, 2016.

[17] Peter Gammie and Ron Van der Meyden. MCK: Model checking the logic of knowledge. In *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 479–483, Berlin, Heidelberg, 2004. Springer.

[18] Leifeng He and Guanjun Liu. Petri net based symbolic model checking for computation tree logic of knowledge. arXiv, 2020.

[19] Jelle Hellings, George H.L. Fletcher, and Herman Haverkort. Efficient external-memory bisimulation on DAGs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 553—-564. Association for Computing Machinery, 2012.

[20] Gijs Kant, Alfons Laarman, Jeroenn Meijer, Jaco Van de Pol, Stefan Blom, and Tom Van Dijk. LTSmin: High-performance language-independent model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *Lecture Notes in Computer Science*, pages 692–707, Berlin, Heidelberg, 2015. Springer.

[21] Daniel Kunkle, Vlad Slavici, and Gene Cooperman. Parallel disk-based computation for large, monolithic binary decision diagrams. In *4th International Workshop on Parallel Symbolic Computation (PASCO)*, pages 63–72, 2010.

[22] Michael Lampis, Georgia Kaouri, and Valia Mitsou. On the algorithmic effectiveness of digraph decompositions and complexity measures. *Discrete Optimization*, 8(1):129–138, 2011. Parameterized Complexity of Discrete Optimization.

[23] Yi Lin, Lucas M. Tabajara, and Moshe Y. Vardi. ZDD Boolean synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 64–83. Springer, 2022.

[24] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: an open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, 19:9–30, 2017.

[25] Ulrich Meyer, Peter Sanders, and Jop Sibeyn. *Algorithms for Memory Hierarchies: Advanced Lectures*. Springer, Berlin, Heidelberg, 2003.

[26] Kim Milvang-Jensen and Alan J. Hu. BDDNOW: a parallel BDD package. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 501–507, Berlin, Heidelberg, 1998. Springer.

[27] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th Design Automation Conference (DAC)*, pages 272–277. Association for Computing Machinery, 1993.

[28] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *27th Design Automation Conference (DAC)*, pages 52–57. Association for Computing Machinery, 1990.

[29] Thomas Mølhave. Using TPIE for processing massive data sets in C++. Technical report, Duke University, Durham, NC, 2012.

[30] Hiroyuki Ochi, Koichi Yasuoka, and Shuzo Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *International Conference on Computer Aided Design (ICCAD)*, pages 48–55. IEEE Computer Society Press, 1993.

[31] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991.

[32] Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierarchy. In *33rd Design Automation Conference (DAC)*, pages 635–640. Association for Computing Machinery, 1996.

[33] Steffan Christ Sølvsten and Jaco van de Pol. Predicting memory demands of BDD operations using maximum graph cuts. In *Automated Technology for Verification and Analysis*, volume 14216 of *Lecture Notes in Computer Science*, pages 72–92. Springer, 2023.

[34] Fabio Somenzi. CUDD: CU decision diagram package, 3.0. Technical report, University of Colorado at Boulder, 2015.

[35] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *Design Automation Conference Proceedings*, volume 33, pages 641–644, 1996.

[36] Steffan Christ Sølvsten. BDD Benchmark. Zenodo, 2022.

[37] Steffan Christ Sølvsten and Jaco van de Pol. Adiar 1.0.1 : Experiment data, 11 2021.

[38] Steffan Christ Sølvsten and Jaco van de Pol. Adiar 1.1: Zero-suppressed Decision Diagrams in External Memory. In *NASA Formal Methods Symposium*, LNCS 13903, Berlin, Heidelberg, 2023. Springer.

[39] Steffan Christ Sølvsten and Jaco van de Pol. Adiar 1.1.0 : Experiment data, 03 2023.

[40] Steffan Christ Sølvsten and Jaco van de Pol. Adiar 1.2.0 : Experiment data, 07 2023.

[41] Steffan Christ Sølvsten, Jaco van de Pol, Anna Blume Jakobsen, and Mathias Weller Berg Thomasen. Adiar: Binary Decision Diagrams in External Memory. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13244 of *Lecture Notes in Computer Science*, pages 295–313, Berlin, Heidelberg, 2022. Springer.

[42] Darren Erik Vengroff. A transparent parallel I/O environment. In *In Proc. 1994 DAGS Symposium on Parallel Computation*, pages 117–134, 1994.

[43] Bwolen Yang and David R. O'Hallaron. Parallel breadth-first BDD construction. *SIGPLAN Not.*, 32(7):145–156, 06 1997.


# Random Access on Narrow Decision Diagrams in External Memory

Steffan Christ Sølvsten(✉), Casper Moldrup Rysgaard, and Jaco Van de Pol

Aarhus University, Aarhus, Denmark
{soelvsten,rysgaard,jaco}@cs.au.dk

**Abstract.** The external memory BDD package Adiar can manipulate Binary Decision Diagrams (BDDs) larger than the RAM of the machine. To do so, it uses one or more priority queues to defer processing each recursion until the relevant nodes are encountered in a sequential scan. We outline how to improve the performance of Adiar's algorithms if the BDD width of one of its inputs is small enough to fit into main memory. In this case, one of the algorithms' priority queues can entirely be replaced with (levelised) random access to the nodes of the narrow BDD. This preserves the I/O efficiency of the original algorithm, is applicable to other types of decision diagrams, and significantly improves performance for many larger BDD computations.

**Keywords:** Binary Decision Diagrams · External Memory Algorithms

## 1 Introduction

Based on the work of Lars Arge [4,5], Adiar[1] [24] is an implementation of Binary Decision Diagrams (BDD) [7] capable of handling BDDs larger than the machine's random access memory (RAM). To achieve this, it uses time-forward processing [3,8,15] to replace the conventional depth-first recursion stack with one (or more) priority queue(s) that are synchronised with a sequential iteration through the input BDD(s).

The high performance of conventional BDD implementations is the result of several decades of research. Especially the unique node table and its layout has been vital [12,14,16,19]. Yet, these and other ideas are not applicable to time-forward processing. Hence, new ideas are needed to make Adiar achieve a satisfactory performance. This has motivated the introduction of its levelised priority queue [23], its equality checking algorithm [24], and the concept of levelised cuts [21]. Common to all these optimisations is the use of some meta information about the BDD graph to substantially improve performance.

Adiar's performance was evaluated [22,24] on various combinatorial benchmarks. Each of these benchmarks accumulates a set of constraints, each of which

[1] github.com/ssoelvsten/adiar.


T. Neele and A. Wijs (Eds.): SPIN 2024, LNCS 14624, pp. 137–145, 2025.
https://doi.org/10.1007/978-3-031-66149-5_7

is a very narrow BDD, into one BDD whose size quickly grows large. Certain instances of symbolic model checking or symbolic SCC computation are quite similar. Here, the BDDs that represent transition relations in deterministic finite automata [9,11] or in asynchronous models of concurrency [10], e.g. Petri Nets [13,18], are narrow while the one for the accumulated state space is large.

### 1.1 Contributions

In the same vein as the prior optimisations in [21,24], we show in Sect. 3 how Adiar can exploit the width of the input BDDs (defined in Sect. 2). In this case, the product construction algorithm in [24] can omit the use of one of its priority queues in favour of per-level using random access directly on the narrow BDD. Our experiments in Sect. 4 show that this considerably improves performance for the larger instances of both the motivating use case, i.e. when computing on at least one narrow BDD, and also average use cases.

### 1.2 Related Work

Prior to this work, levelised cuts [21] improves Adiar's performance by soundly upper bounding the size of its priority queues. If it is smaller than main memory, then the external memory priority queue is replaced by a simpler and faster priority queue that only works in internal memory. This has been vital for Adiar's performance on BDDs that do fit into the RAM. In this work, we instead improve Adiar's performance by changing the algorithms' logic. Hence, in constrast to the prior work, this optimisation targets the entire spectrum of BDDs. It especially is of benefit to some larger instances.

CAL [20] (based on [6,17]) is to the best of our knowledge the only other BDD package to compute on BDDs that exceed main memory. To do so, it stores all BDD nodes in a unique node table and uses queues to execute its algorithms in the breadth-first manner. These node tables and queues can be offloaded to the disk via the operating system's swap memory. Yet, this is only efficient, if every level fits into memory. In general, CAL is not I/O-efficient [5], whereas Adiar is [24].

## 2 Preliminaries

### 2.1 I/O Model

Aggarwal and Vitter [1] designed the I/O-model to analyse the data transfers between two levels of a memory hierarchy. Here, the internal memory, e.g. the RAM, has a finite size of $M$ and data exceeding its capacity needs to be transferred in blocks of size $B$ to/from the external memory, e.g. the Disk.

The number of block data transfers (I/Os) needed to sequentially read and write $N$ amounts of data is $\text{scan}(N) \triangleq N/B$. To sort $N$ amounts of data one

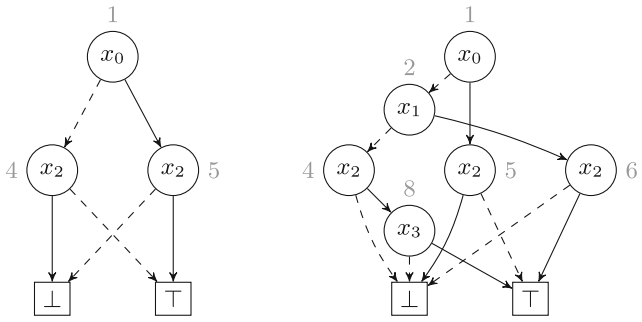



**Fig. 1.** Examples of Reduced and Ordered Binary Decision Diagrams. Terminals are drawn as boxes while internal nodes are drawn as circles with its decision variable. The *then* and *else* edges are respectively drawn solid and dashed.

needs to use $\mathrm{sort}(N) \triangleq N/B \cdot \log_{M/B}(N/B)$ I/Os. Furthermore, one can also design a priority queue capable of inserting and extracting $N$ elements in the optimal $\Theta(\mathrm{sort}(N))$ number of I/Os [3]. For all realistic values of $N$, $M$, and $B$, both $\mathrm{scan}(N)$ and $\mathrm{sort}(N)$ are several magnitudes smaller than $N$ itself.

### 2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDD) [7] provide a concise representation of Boolean functions $\mathbb{B}^n \to \mathbb{B}$ as a singly-rooted directed acyclic graph (DAG). As shown in Fig. 1, a BDD has two terminals with the Boolean values $\mathbb{B} = \{\bot, \top\}$ as the function's output whereas each internal BDD node provides an if-then-else decision on one of the $n$ input variables, $x_i$.

What are colloquially referred to as BDDs are in fact *Reduced* and *Ordered* BDDs (ROBDDs). A BDD is ordered if the decision variables only occur once on each path from the root to a terminal and always following the same order. This induces a *levelisation* with level $x_i$ only containing nodes with the said variable. The *width* of a BDD is the size of its largest level. A BDD is reduced, if there are (1) no *duplicate* nodes and (2) no *redundant* nodes. A node is a duplicate if it represents the same if-then-else. In conventional BDDs, a node is redundant if it has two identical children.

Fundamental to BDDs is the *Apply* operation, which, given BDDs $f$ and $g$ and a binary operator $\odot$, constructs the BDD for $f \odot g$. This is done via a product construction of both input BDDs and applying $\odot$ when arriving at a pair of terminals. As an example, Fig. 2 shows the product of Fig. 1a and 1b.

Here, we only provide a high-level description of Adiar's Apply algorithm that includes the details needed for Sect. 3; we refer to [24] for a detailed explanation. To make it I/O-efficient, Adiar imposes a total order on its BDD nodes such that the BDD is sorted level by level. Specifically, each node is associated with a numeric *time point* that they are encountered in the input (grey indices in

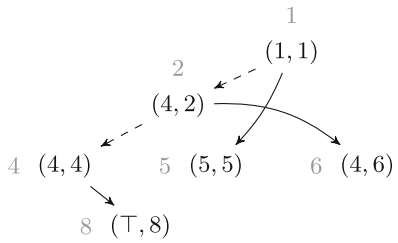



**Fig. 2.** Product Graph for BDDs in Fig. 1a and Fig. 1b.

Fig. 1). This ensures that each BDD node will always come after its parent during a sequential scan. To identify the pair of children of $(n_f, n_g) \in f \times g$ in a product construction, the ordering implies one needs the first to-be seen node, $\min(n_f, n_g)$, and possibly also the second one, $\max(n_f, n_g)$. Hence, a priority queue can make the recursion to target $(n_f, n_g)$ match the sequential scan of the inputs by sorting it on the time point $\min(n_f, n_g)$. If $\max(n_f, n_g)$ is also needed then the BDD node $\min(n_f, n_g)$ is further forwarded to $\max(n_f, n_g)$ with a second priority queue. To do so, the second priority queue sorts its elements on the time point $\max(n_f, n_g)$. To guarantee a polynomial running time, recursions to the same target are grouped together. This is done by resolving ties in both priority queues' ordering via a lexicographical ordering of the recursion targets. For example, the product graph of the BDDs in Fig. 1a and Fig. 1b is resolved in [24] in the order depicted in Fig. 2.

This Apply algorithm only uses $\mathcal{O}(\text{scan}(N_f) + \text{scan}(N_g) + \text{sort}(T))$ I/Os unlike the $\mathcal{O}(T)$ I/Os used by conventional recursive implementations [5, 12], where $N_f, N_g$ are the number of internal BDD nodes in $f$ and $g$, respectively, and $T$ is the number of BDD nodes in the output.

## 3 Using Random Access for Narrow Decision Diagrams

Without loss of generality, assume that the second input, $g$, to the Apply operation is *narrow*, i.e. the width of $g$ is smaller than some threshold $\theta < M/2$. In this case, we can completely omit the second priority queue.

To do so, when processing level $x_i$, we load all BDD nodes of $g$ at level $x_i$ from external memory. This provides immediate random access to the entire level $x_i$ of $g$. Hence, the second priority queue can be omitted if the ordering of the first priority queue is changed accordingly. Specifically, the first priority queue now solely has to respect the levels of the recursive calls and synchronise them with the sequential scan through $f$. Hence, the recursion target $(n_f, n_g)$ should first be sorted on its level to not miss any requests where the level of $n_f$ is below the one of $n_g$. Secondly, it is sorted lexicographically to respect the sequential scan through $f$. Futhermore, lexicographical sorting also groups recursive calls for the same target together, which preserves the polynomial running time and I/Os.

Doing so only affects the order in which all recursions are resolved; the output is still isomorphic to what is produced by the algorithm in [24]. For example in Fig. 2, if random access is used on the BDD from Fig. 1b then $(4, 6)$ is resolved prior to $(5, 5)$. In the previous algorithm [24], both the node at time point 4 in Fig. 1a and the one at 6 in Fig. 1b had to be visited in-order to resolve the product $(4, 6)$; hence, this product was resolved after $(5, 5)$. Instead, with random access the node at time point 6 in Fig. 1b is immediately available when reading the one at 4 in Fig. 1a; hence, $(4, 6)$ is resolved before $(5, 5)$.

**Proposition 1.** *The Apply algorithm with random access on a narrow BDD saves $\mathcal{O}(sort(T))$ I/Os in comparison to the prior algorithm from [24].*

*Proof.* Since the input BDDs are already sorted based on their level, loading the levels of $g$ top-down is possible in a single sequential scan. Yet, the algorithm in [24] also needs to scan through $g$. Hence, loading the nodes of $g$ for random access does not cost any additional I/Os. Yet, it completely removes the $\mathcal{O}(\mathrm{sort}(T))$ I/Os incured by the second priority queue.

Note that this is only a constant improvement over the algorithm in [24]. Furthermore, it is only an $\mathcal{O}(\mathrm{sort}(T))$ rather than a $\Theta(\mathrm{sort}(T))$ improvement, since recursion requests do not necessarily need to be moved into the second priority queue.

## 4 Experimental Evaluation

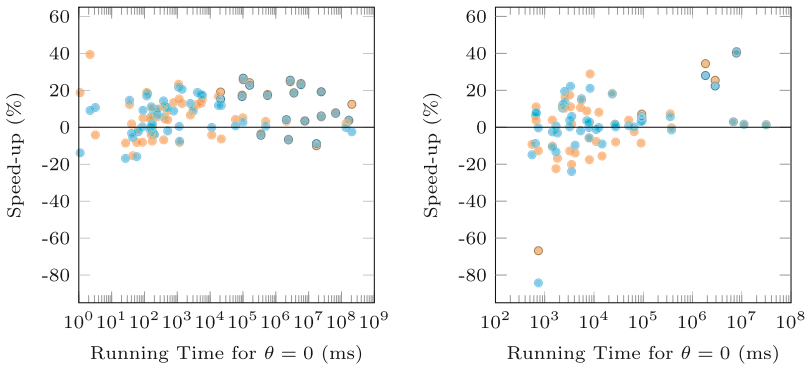
We have implemented the modified algorithm of Sect. 3 and run the benchmarks from [22,24] with threshold $\theta = 0$, $B$ (2 MiB), and $\infty$. Using $\theta = 0$ essentially turns the random access optimisation off and provides a baseline. On the other hand, using $\theta = \infty$ entirely replaces the previous Apply algorithm. Finally, $\theta = B$ provides a small value which covers the motivating use cases while also leaving more of the internal memory to the remaining priority queue.

The benchmarks of [22,24] consist of two categories. First, the *Combinatorial Counting* problems, e.g. the Queens problem, primarily involve the accumulation of lots of narrow decision diagrams. On the other hand, the *EPFL* [2] *Circuit Verification* provides a typical use case for decision diagrams.

As in [21,22,24], we have run all experiments on the CSCAA *Grendel* cluster where Adiar is initialised with $M = 300$ GiB. For each value of $\theta$ and each benchmark instance, the running time has been measured between 3 and 15 times (11.2 times on average) depending on its expected running time. We consider a measurement to be significant if the difference between the mean running time of $\theta = 0$ and $\theta = B, \infty$ is larger than twice their largest standard deviation. Figure 3 shows the speed-up in the mean running time for all 113 benchmark instances. Table 1 provides a summary of all significant instances.

Both $\theta = B$ and $\theta = \infty$ provide a significant performance increase in performance for the larger combinatorial benchmarks compared to the $\theta = 0$ baseline. Of the 64 combinatorial instances, 14 (21.9% of all of these instances) had

142 S. C. Sølvsten et al.




(a) Combinatorial Counting (b) EPFL [2] Circuit Verification

**Fig. 3.** Speed-up in running time with • $\theta = B$ and • $\theta = \infty$ relative to $\theta = 0$ (higher is better). Statistically significant measurements have a black border.

a significant improvement of 15.4% on average for $\theta = B$ and 16.3% for $\theta = \infty$. These 14 instances all require 10 s or more to solve with $\theta = 0$. In total, 26 out of the total 64 instances required this amount of time to solve. That is, performance improved for 53.8% of these larger instances.

Similarly to the combinatorial benchmarks, EPFL Verification also gains significant improvements for many of its larger instances. Only one circuit, `int2float`, requires significantly more time to verify. Yet, while a decrease in performance of 67% seems worrisome, it is only an increase in the computation time from 0.74 s to 1.12 s.

Finally, the optimisation presented in this work further closes the gap between Adiar and conventional BDD packages. For example, CUDD [26] can solve up to the 15-Queens problem with BDDs. In [24] the gap between Adiar and CUDD for this problem's instance was a factor of 1.42. In [21], this was improved to 1.26. With $\theta = B, \infty$, the gap is now further decreased down to 1.07. Simultaneously, this also improves performance for instances not solvable with CUDD. For example, it improves the solving time of 16-Queens by 19.2%.

**Table 1.** Speed-up ($\triangle$) and slowdown ($\triangledown$) of $\theta = B$ and $\theta = \infty$ for Combinatorial Counting (CC) and EPFL [2] Circuit Verification (EPFL) benchmarks.

| | | | # Significant Instances | | | | Average Relative Difference | |
|---|---|---|---|---|---|---|---|---|
| | | | $\triangle$ | | $\triangledown$ | | $\triangle$ | $\triangledown$ |
| CC | • | $\theta = B$ | 14 | (21.9%) | 3 | (4.7%) | 15.4% | −6.7% |
| | • | $\theta = \infty$ | 14 | (21.9%) | 3 | (4.7%) | 16.3% | −6.7% |
| EPFL | • | $\theta = B$ | 6 | (12.2%) | 0 | (0.0%) | 25.1% | – |
| | • | $\theta = \infty$ | 6 | (12.2%) | 1 | (2.0%) | 26.9% | −66.8% |

## 5 Conclusion

Let a BDD be *narrow* if its width is smaller than some threshold $\theta < M/2$, where $M$ is the amount of internal memory. Specifically, each individual level of a narrow BDD fits into internal memory. Hence, one can load each of its levels in their entirety and do random access on it. If some input to the Apply algorithm in [24] is narrow then one does not need to synchronise its computation with the sequential order of the narrow one. In this work, we have sketched how this algorithm can be adapted to this case, to save on computation time and I/Os.

For $\theta = B$ ($B$ is the block transfer size), our experiments show a significant improvement in performance for larger instances. In the motivating use case with lots of narrow BDDs, performance increased significantly by 11.8% on average. In the average use case, performance even increased significantly by 25.1%.

Relative to $\theta = B$, our results with an unlimited $\theta$ further improves performance significantly for 3 larger combinatorial instances. No instances slowed down significantly. Hence, we have implemented levelised random access as part of Adiar 2.0 and based on the observation above, we use a large $\theta$ of $M/8$.

### 5.1 Future Work

While using levelised cuts [21] improves Adiar's performance for algorithms whose priority queues fit into RAM, it still leaves a gap between the running time of Adiar and conventional BDD packages for the smallest problems.

To efficiently solve the smallest of BDD instances (15 MiB or smaller), the BDD package CAL [20] switches from its breadth-first approach to the conventional depth-first algorithms.

The work presented in this paper can be extended to compute on input BDDs that are stored in an (internal memory) unique node table. Its (unreduced) output, which may exceed main memory, is still stored on the disk. Conversely, one can change Adiar's algorithms to place the final (reduced) BDD back in the node table. Hence, this work provides the basis for an efficient and seamless transition between a conventional depth-first approach and Adiar's external memory algorithms. This will be the final step to make Adiar competitive across the entire spectrum of BDDs.

Furthermore, this work applies to all of Adiar's product construction operations, such as variable quantification. Hence, this work, together with [21], is vital for the design of an I/O-efficient relational product that is usable in practice.

**Acknowledgements.** Thanks to the Centre for Scientific Computing, Aarhus, (phys.au.dk/forskning/cscaa/) for access to the Grendel cluster.

**Data Availability Statement.** The data presented in Sect. 4 is available at [25] while the code to run the benchmarks can be found at [27].

## References

1. Aggarwal, A., Vitter, Jeffrey, S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988). https://doi.org/10.1145/48529.48535
2. Amarú, L., Gaillardon, P.E., De Micheli, G.: The EPFL combinational benchmark suite. In: 24th International Workshop on Logic and Synthesis (2015)
3. Arge, L.: The buffer tree: a new technique for optimal I/O-algorithms. In: Workshop on Algorithms and Data Structures (WADS). LNCS, vol. 955, pp. 334–345. Springer, Heidelberg (1995).https://doi.org/10.1007/3-540-60220-8_74
4. Arge, L.: The I/O-complexity of ordered binary-decision diagram manipulation. In: 6th International Symposium on Algorithms and Computations (ISAAC). LNCS, vol. 1004, pp. 82–91 (1995). https://doi.org/10.1007/BFb0015411
5. Arge, L.: The I/O-complexity of ordered binary-decision diagram. In: BRICS RS Preprint Series, vol. 29. Department of Computer Science, University of Aarhus (1996). https://doi.org/10.7146/brics.v3i29.20010
6. Ashar, P., Cheong, M.: Efficient breadth-first manipulation of binary decision diagrams. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 622–627. IEEE Computer Society Press (1994). https://doi.org/10.1109/ICCAD.1994.629886
7. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
8. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1995), pp. 139—149. Society for Industrial and Applied Mathematics (1995)
9. Elgaard, J., Klarlund, N., Møller, A.: MONA 1.x: new techniques for WS1S and WS2S. In: Proceedings of the 10th International Conference on Computer-Aided Verification, CAV 1998. LNCS, vol. 1427, pp. 516–520. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-61648-9_56
10. Kant, G., Laarman, A., Meijer, J., Van de Pol, J., Blom, S., Van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
11. Klarlund, N.: Mona & Fido: the logic-automaton connection in practice. In: Computer Science Logic. LNCS, vol. 1414, pp. 311–326. Springer, Cham (1998). https://doi.org/10.1007/BFb0028022
12. Klarlund, N., Rauhe, T.: BDD algorithms and cache misses. In: BRICS Report Series, vol. 26 (1996). https://doi.org/10.7146/brics.v3i26.20007
13. Larsen, C.A., Schmidt, S.M., Steensgaard, J., Jakobsen, A.B., van de Pol, J., Pavlogiannis, A.: A truly symbolic linear-time algorithm for SCC decomposition. In: Tools and Algorithms for the Construction and Analysis of Systems (2). LNCS, vol. 13994, pp. 353–371. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30820-8_22
14. Long, D.E.: The design of a cache-friendly BDD library. In: Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 639–645. Association for Computing Machinery (1998)
15. Meyer, U., Sanders, P., Sibeyn, J.: Algorithms for Memory Hierarchies: Advanced Lectures. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36574-5

16. Minato, S.I., Ishiura, N., Yajima, S.: Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In: 27th Design Automation Conference (DAC), pp. 52–57. Association for Computing Machinery (1990). https://doi.org/10.1145/123186.123225
17. Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: International Conference on Computer Aided Design (ICCAD), pp. 48–55. IEEE Computer Society Press (1993). https://doi.org/10.1109/ICCAD.1993.580030
18. Pastor, E., Roig, O., Cortadella, J., Badia, R.M.: Petri net analysis using boolean manipulation. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 416–435. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58152-9_23
19. Pastva, S., Henzinger, T.: Binary decision diagrams on modern hardware. In: Conference on Formal Methods in Computer-Aided Design, pp. 122–131 (2023)
20. Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.: High performance BDD package by exploiting memory hierarchy. In: 33rd Design Automation Conference (DAC), pp. 635–640. Association for Computing Machinery (1996). https://doi.org/10.1145/240518.240638
21. Sølvsten, S.C., Van de Pol, J.: Predicting memory demands of BDD operations using maximum graph cuts. In: André, É., Sun, J. (eds.) Automated Technology for Verification and Analysis. LNCS, vol. 14216, pp. 72–92. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-45332-8_4
22. Sølvsten, S.C., Van de Pol, J.: Adiar 1.1: zero-suppressed decision diagrams in external memory. In: NASA Formal Methods Symposium, LNCS, vol. 13903, Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-33170-1_28
23. Sølvsten, S.C., Van de Pol, J., Jakobsen, A.B., Thomasen, M.W.B.: Efficient binary decision diagram manipulation in external memory. arXiv preprint arXiv:2104.12101 (2021)
24. Sølvsten, S.C., Van de Pol, J., Jakobsen, A.B., Thomasen, M.W.B.: Adiar: binary decision diagrams in external memory. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 13244, pp. 295–313. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-030-99527-0_16
25. Sølvsten, S.C., Rysgaard, C.M., van de Pol, J.: Adiar 2.0.0-beta.3 : Experiment Data (2024). https://doi.org/10.5281/zenodo.10493770
26. Somenzi, F.: CUDD: CU decision diagram package, 3.0. Tech. rep., University of Colorado at Boulder (2015)
27. Slvsten, S.C.: BDD Benchmark. Zenodo (2024).https://doi.org/10.5281/zenodo.10803154

# Multi-variable Quantification of BDDs in External Memory using Nested Sweeping
## (Extended Version)

Steffan Christ Sølvsten and Jaco van de Pol

Aarhus University, Denmark {soelvsten,jaco}@cs.au.dk



**Abstract.** Previous research on the Adiar BDD package has been successful at designing algorithms capable of handling large Binary Decision Diagrams (BDDs) stored in external memory. To do so, it uses consecutive sweeps through the BDDs to resolve computations. Yet, this approach has kept algorithms for multi-variable quantification, the relational product, and variable reordering out of its scope.
In this work, we address this by introducing the *nested sweeping* framework. Here, multiple concurrent sweeps pass information between each other to compute the result. We have implemented the framework in Adiar and used it to create a new external memory multi-variable quantification algorithm. In practice, this improves Adiar's running time by a factor of 1.7. In turn, this work extends the previous research results on Adiar to also apply to its quantification operation: compared to conventional depth-first implementations, Adiar with nested sweeping is able to solve more problems and/or solve them faster.

## 1 Introduction

The ability of Binary Decision Diagrams (BDDs) to represent Boolean formulae as small directed acyclic graphs (DAGs) have made them an invaluable tool to solve many complex problems. For example, recently they have been used to check type-and-effect systems [35, 36], to generate proofs for SAT and QBF solvers [14–16], for circuit synthesis [22, 32], to solve games [37, 45, 53], and for symbolic model checking [3, 19, 20, 23, 26, 28, 34].

Implementations of decision diagrams conventionally make use of recursive depth-first algorithms and a unique node table [10,21,29,33,40,52]. Both of these introduce random access, which pauses the entire computation while missing data is fetched [30,39,44]. For large enough instances, data has to reside on disk and the resulting I/O-operations that ensue become the bottle-neck.

Adiar [49] is a BDD package written in C++ based on the ideas of Lars Arge [5]: the depth-first recursive algorithms are replaced with iterative algorithms. Here, one or more priority queues reorder the execution of recursive calls such that they are synchronised with a level-by-level traversal of the inputs. This makes Adiar's algorithms, unlike the conventional recursive implementations, optimal in the I/O-model [1] of Aggarwal and Vitter [5, 6]. In turn, this enables

it to manipulate BDDs beyond the reach of conventional BDD packages at a negligible cost to its running time [49].

Yet, the ideas in [5,6,49] only provide a translation of the simplest BDD algorithms. Specifically, it only provides a translation for operations without any data-dependency on earlier recursive calls in each node in the BDD's graph. This does not provide a way to translate the more complex BDD algorithms that recurse on intermediate recursion results, e.g. multi-variable quantification. Hence, until this work, Adiar could not easily be used for solving Quantified Boolean formulæ (QBF). Furthermore, game solving and symbolic model checking has until now been out of reach for Adiar.

### 1.1 Contributions

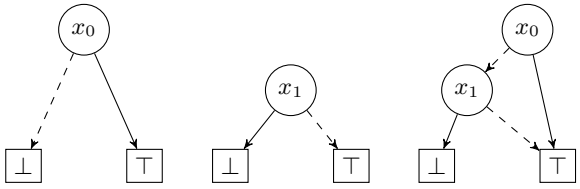
In Section 3, we introduce the notion of *nested sweeping* to provide a framework on which these more complex BDD operations can be implemented. Here, an *outer* bottom-up sweep accumulates the results from multiple nested *inner* sweeps. With this framework in hand, we implement an I/O-efficient multi-variable quantification akin to the one in conventional BDD packages. Furthermore, we identify in Section 3.2 optimisations for the nested sweeping framework in general and in Section 3.3 for the quantification operation in particular. Section 4 provides an overview of the implementation while Section 5 shows that nested sweeping improves the running time in practice by a factor of 1.7 when solving QBF-encodings of two-player games and when reasoning about the transition system in Conway's Game of Life [24]. We compare our approach to related work in Section 6 and finally provide our conclusions and future work in Section 7.

## 2 Preliminaries

### 2.1 The I/O-Model

Aggarwal and Vitter introduced the I/O-model [1] to analyse the cost of transferring data to and from a slow storage device. Here, computations can only operate on data that resides in *internal* memory, e.g. the RAM, with a finite size of $M$. Hence, if the input of size $N$ (or some intermediate result) exceeds $M$ then it needs to be transferred to and from *external* memory, e.g. the disk. Yet, each such data transfer (I/O) moves an entire consecutive block of $B$ elements; an algorithm's I/O-complexity is the number of I/Os it uses.

One needs $\mathrm{scan}(N) \triangleq N/B$ I/Os to linearly scan through a consecutive list of $N$ elements in external memory [1]. Assuming $N > M$, one needs to use $\Theta(\mathrm{sort}(N))$ I/Os to sort $N$ elements, where $\mathrm{sort}(N) \triangleq N/B \cdot \log_{M/B}(N/B)$ [1]. Furthermore, one can design an I/O-efficient priority queue capable of doing $N$ insertions and deletions in $\Theta(\mathrm{sort}(N))$ I/Os [4]. For simplicity, we overload $\mathrm{scan}(N)$ to be $N$ and $\mathrm{sort}(N)$ to be $N \log N$ when referring to an algorithm's time complexity rather than its I/O complexity.




(a) $x_0$ (b) $\neg x_1$ (c) $x_0 \vee \neg x_1$

Fig. 1: Examples of Reduced Ordered Binary Decision Diagrams. Terminals are drawn as boxes surrounding their Boolean value. Internal nodes are drawn as circles and contain their decision variable. Arcs to the *high* and *low* child are respectively drawn solid and dashed.

Intuitively, an algorithm is I/O-inefficient if it uses an entire I/O to retrieve a block but does not make use of a significant portion of the $B$ elements within. That is, random access can result in $N$ I/Os. For all realistic values of $N$, $M$, and $B$, this is several magnitudes larger than both $\mathrm{scan}(N)$ and $\mathrm{sort}(N)$.

### 2.2 Binary Decision Diagrams

As shown in Fig. 1, a Binary Decision Diagram [13] (BDD) (based on [2, 31]) represents an $n$-ary Boolean function as a singly-rooted directed acyclic graph (DAG). Each of its two sinks, refered to as *terminals*, contain one of the two Boolean values, $\mathbb{B} = \{\top, \bot\}$. These represent the function's output values. An internal BDD node, $v$, is associated in $v.\texttt{var}$ with a Boolean input variable $x_i$. Furthermore, it has two BDD nodes as children, $v.\texttt{low}$ and $v.\texttt{high}$. These three values in $f$ encode the ternary if-then-else

$$v.\texttt{var}\ ?\ v.\texttt{high} : v.\texttt{low}\ .$$

What are colloquially referred to as BDDs are in fact *Reduced Ordered* Binary Decision Diagrams (ROBDDs). An Ordered BDD (OBDD) restricts each variable to occur at most once on each path from the root to a terminal and to occur according to a certain order, $\pi$. This gives rise to a levelisation of the OBDD where each level, $\ell$, is associated with an input variable, $x_i$. For sake of simplicity, we assume that $\pi$ is the identity order. A *Reduced* OBDD further restricts the DAG such that (1) no nodes are duplicates of another and (2) no node is redundant, i.e. $v.\texttt{high} = v.\texttt{low}$. Assuming the variable ordering, $\pi$, is fixed, ROBDDs are a unique canonical form of the Boolean function it represents.

**Quantification Algorithm** The levelisation of OBDDs allows the recursive BDD algorithms to both be efficient and elegant. For example, the `or` operation works by a product construction of the two input BDDs. Here, each node of the output BDD simulates, according to $\pi$, the decision(s) taken on the shallowest BDD node(s) in the product of nodes from the input.

```
1  exists(v, X)
2    if v = ⊥ ∨ v = ⊤
3        return v
4    exi0 ← exists(v.low, X)
5    exi1 ← exists(v.high, X)
6    if v.var ∉ X
7        return Node { v.var, exi0, exi1 }
8    return or(exi0, exi1)
```

Fig. 2: A recursive multi-variable **exists** operation.

Since $(\exists x : \phi) \equiv \phi[\top/x] \vee \phi[\bot/x]$, the `or` operation can be used as the basis for an existential quantification ($\exists$) for a set of input variables, $X = \{x_i, x_j, \ldots, x_k\}$. As shown in Fig. 2, if $v$ is a terminal then this (sub)BDD depends on none of the to be quantified variables. Otherwise, both its children are resolved recursively into intermediate results, `exi0` and `exi1`. If the decision variable of the root, $v$.`var`, should not be quantified, a new node with variable $v$.`var` is created from the two recursive results. Otherwise, `exi0` and `exi1` are instead combined (recursively once more) with a nested `or` operation.

Similarly, one can implement a universal quantification ($\forall$) by use of a nested `and` operation. For clarity, our contributions in Section 3 are only phrased with respect to the `exists` operation. But, everything that follows also applies to `forall` by replacing `or` with `and`.

**Relational Product** The *relational product* computes the set of states after taking a step in a transition system with the formula $\exists \vec{x} : S(\vec{x}) \wedge R(\vec{x}, \vec{x'})$. Hence, the support for a multi-variable quantification operation is key for the application of BDDs in the context of symbolic model checking.

### 2.3 I/O-efficient BDD Manipulation

The Adiar [49] BDD package builds on top of Lars Arge's ideas [5, 6] on how to improve the I/O complexity of BDD manipulation. To not introduce random access, Adiar does not use any hash tables nor recursion for its BDD manipulation. As a result, different BDD objects do not share common subtrees in Adiar. For the same reason, it neither uses pointers to traverse its BDDs. Instead, every BDD node $v$ is uniquely identified by a pair ($v$.`var`, $v$.`id`) where $v$.`id` is $v$'s index on level $v$.`var`. Lexicographically, this *unique identifier* (`uid`) imposes a total ordering of all BDD nodes into a levelised sequence of nodes. Here, the `uid` does not specify exactly where to find a BDD node in the input but when to expect it relative to the one currently read. For example, the BDD for $x_0 \vee \neg x_1$ in Fig. 1c is represented in Adiar as the list of nodes in Fig. 3a: every node is a 3-tuple with its `uid` followed by the unique identifier of its low and its high children.

As depicted in Fig. 4, the previous BDD operations in Adiar, such as `or`, process a BDD with two sweeps. Both sweeps use *time-forward processing* [4, 18]

$$[\ \{(0,0),(1,0),\top\};\ \{(1,0),\top,\bot\}\ ]$$

(a) Node-based Representation

$$[\ (0,0)\longrightarrow(1,0);\ (1,0)\longrightarrow\bot;\ (0,0)\dashrightarrow\top;\ (1,0)\dashrightarrow\top;\ ]$$

(b) Arc-based Representation

Fig. 3: Representation of the $x_0 \vee \neg x_1$ BDD (Fig. 1c) in Adiar.

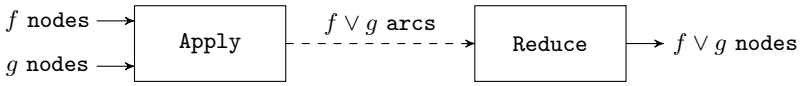



Fig. 4: The Apply–Reduce pipeline of `or` in Adiar.

to achieve their I/O-efficiency: computation is deferred with one or more priority queues until all relevant data has been read. During the first sweep, the *Apply*, the entire recursion tree is unfolded top-down. Here, the priority queues also double as a computation cache [10,40] by merging separate paths to the same recursion target. Hence, the resulting output is in fact not a tree but a DAG. Yet, it is only an OBDD and needs to be reduced. To do so, Adiar uses an I/O-efficient variant of the original bottom-up Reduce algorithm by Bryant [4,13]. Here, a priority queue is used to forward the `uid` of reduced nodes $t'$ in the final ROBDD to their to be reduced parents $s$ in the intermediate OBDD. Yet, to know the parents $s$, the Reduce needs the intermediate OBDD to be transposed, i.e. the DAG's edges to be reversed. Luckily, the Apply sweep outputs its OBDD transposed and so no extra work is needed [5,49]. For example, the `or` of Fig. 1a and Fig. 1b creates the arc-based representation in Fig. 3b. Here, the arcs (directed edges) end up sorted by their target. For all intents and purposes, this is a transposition of the DAG. This can then be reduced into the node-based representation in Fig. 3a.

The I/O and time complexity of this Apply–Reduce tandem is

$$\mathcal{O}(\mathrm{sort}(N+T))\ ,$$

where $N$ is the size of the input(s) and $T$ is the size of the unreduced output of the Apply sweep [49].

To catch up with conventional implementation's performance, major efforts have been dedicated to improve on this foundational design.

**Levelised Cuts [51]** The arcs placed in the above-mentioned priority queues correspond to cuts in the (R)OBDDs. These cuts have a particular shape that follows its levelisation. Hence, the maximum size of the priority queues is bounded by (heuristic over-approximations of) the maximum levelised cut in the input.

These sound upper bounds on the priority queues' size can in turn be used to determine a priori whether one can use a priority queue that is much faster but only works in internal memory.

In practice, this improves performance for smaller and moderate instances.

**Levelised Random Access [50]** Orthogonally, a product construction's Apply sweep, e.g. an `or`, can be simplified if one of its inputs is narrow, i.e. each level fits into internal memory. In this case, one can load each level in its entirety into internal memory. Doing so, provides random access to all of its nodes on said level making one of the Apply's two priority queues in [49] obsolete.

In practice, this improves performance for larger instances.

## 3 I/O-efficient Multi-variable Quantification

The work in [49] only covers simple BDD operations without any data-dependencies in its recursion, e.g. the `or`. Yet, this does not cover the `exists` in Fig. 2, where the nested call to `or` on line 8 depends on the recursions from lines 4 and 5.

To address this, we introduce the *nested sweeping* framework. As shown in Figs. 5 and 6, this wraps the algorithm(s) depicted in Fig. 4: after transposing the input in an initial Apply sweep, a single *outer* Reduce sweep accumulates the result of multiple *inner* Apply–Reduce sweeps. More precisely, nested sweeping consists of the following four phases.

*Outer Apply:* As shown in Fig. 7, inputs are combined (and possibly manipulated) in an Apply sweep into a single file, $F_{outer}$. This transposes and merges the inputs such that they are of the form needed by the Reduce of [49].

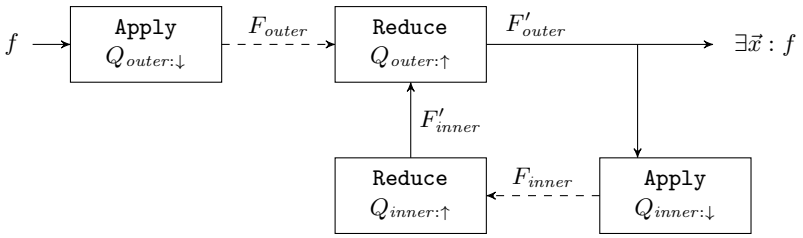



Fig. 5: The Apply–Reduce pipeline of `exists` with Nested Sweeping.

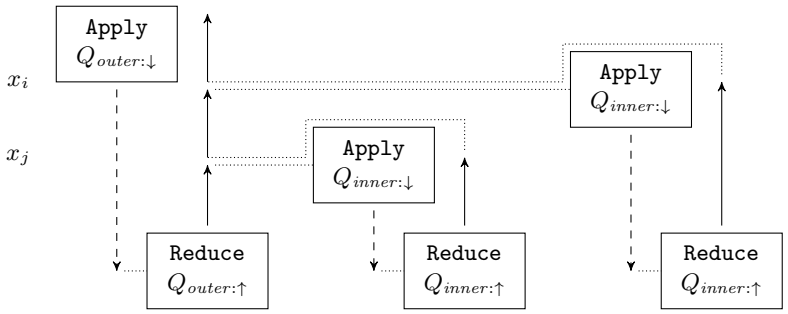



Fig. 6: Sweep direction (solid/dashed) and control-flow (dotted) of Nested Sweeping. The y-axis corresponds to the levels within the BDD.

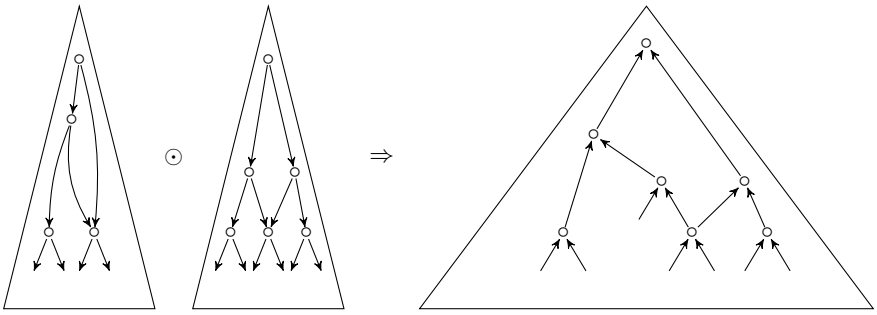


Fig. 7: Outer Apply one (or more) input BDD(s) are processed in a top-down sweep to create a single transposed BDD.

In the case of `exists`, this could be a simple transposition of $f$. In Section 3.3, we describe how this phase can do double duty to process some of the quantifications. In the case of the relational product, the conjunction of states and relation can be computed as part of this phase.

*Outer Reduce:* As in [49], each level of $F_{outer}$ is reduced bottom-up by having a priority queue, $Q_{outer:\uparrow}$, forward the information about reduced nodes, $t'$, to their unreduced parents, $s$. The reduced output is pushed into a new file, $F'_{outer}$.

Let $x_j$ be the next level that needs a nested sweep. For `exists`, $x_j$ is the largest still to be quantified variable in $X$. As visualised in Fig. 8, the logic of [49] is extended as follows:

1. If the current level is $x_j$, each arc $s \longrightarrow t'$ to a reduced node $t'$ at this level is turned into a request and placed in a second priority queue, $Q_{inner:\downarrow}$.
   For `exists`, the requests are of the form $s \longrightarrow (t'.\texttt{low}, t'.\texttt{high})$.
2. If the current level is deeper than $x_j$, nodes are reduced as in [49] with a caveat: whether the arc $s \longrightarrow t'$ to the reduced node $t'$ is placed in $Q_{outer:\uparrow}$ or in $Q_{inner:\downarrow}$ depends on the level of the unreduced parent $s$ as follows:
   (a) If $x_j \leq s.\texttt{var}$, i.e. $s$ is as deep or deeper than level $x_j$, then $s \longrightarrow t'$ is placed in $Q_{outer:\uparrow}$ as normal.
   (b) Otherwise, i.e. if $s.\texttt{var} < x_j$, $s \longrightarrow t'$ is placed in $Q_{inner:\downarrow}$ instead.

For `exists`, Case 1 matches the invocation of `or` on line 8 of Fig. 2 whereas 2 is the return with an unquantified variable on line 7.

When level $x_j$ has finished processing, $Q_{inner:\downarrow}$ is populated with all requests that span across level $x_j$. Now, the inner Apply sweep is invoked.

*Inner Apply:* As depicted in Fig. 9a, starting with the requests in $Q_{inner:\downarrow}$, the reduced nodes, $t'$, placed in $F'_{outer}$ by the outer Reduce sweep, are processed with an Apply sweep from [49]. The intermediate unreduced result is placed in a new file, $F_{inner}$.

For `exists`, this sweep is the execution of the `or` on line 8 of Fig. 2. Here, one can use the previous top-down algorithms from [49].

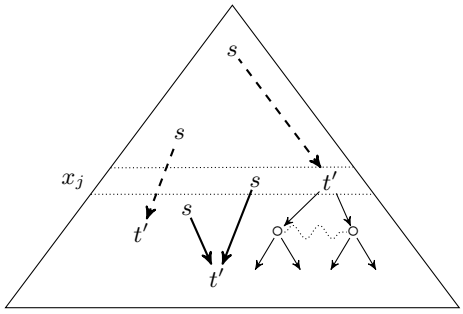
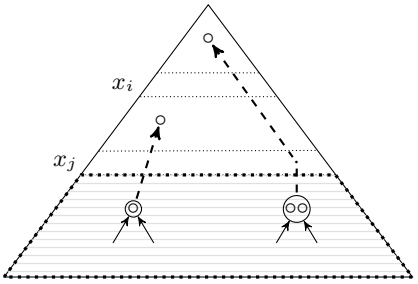
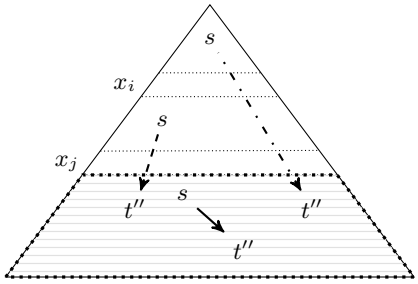



Fig. 8: Outer Reduce: solid arcs stay in $Q_{outer:\uparrow}$ (Case 2a) while dashed arcs are turned into requests for $Q_{inner:\downarrow}$ (Cases 2b on the left and 1 on the right).




(a) Inner Apply: starting with root requests (dashed), $F_{inner}$ is constructed with the to-be preserved subtrees (left) together with new nodes that are products of previous ones (right).




(b) Inner Reduce: arcs below $x_j$ stay within the inner sweep (Case 1, solid). Arcs that cross $x_j$ are given back to the outer (Case 2, dashed) or to the next inner sweep (Case 3, dash dotted).

Fig. 9: Visualization of the Inner Apply and the Inner Reduce.

*Inner Reduce:* After the inner Apply sweep, $F_{inner}$ is reduced in another bottom-up Reduce sweep of [49]. This creates the reduced nodes $t''$ placed in $F'_{inner}$. Let $x_i$ be the next level above $x_j$ that also needs a nested sweep. For `exists`, $x_i$ is the largest variable smaller than $x_j$ that also needs to be quantified. The arc $s \longrightarrow t''$ is placed in a priority queue, $Q_{inner:\uparrow}$, as follows.

1. If $x_j < s.\texttt{var}$, i.e. the parent $s$ is below level $x_j$, then $s \longrightarrow t''$ is forwarded within this inner Reduce sweep's priority queue, $Q_{inner:\uparrow}$.
2. If $s.\texttt{var} \in [x_i, x_j]$, i.e. the parent $s$ is between level $x_i$ and $x_j$ then $s \longrightarrow t''$ is given back to the outer sweep, $Q_{outer:\uparrow}$. This matches Case 2a in the Outer Reduce.
3. If $s.\texttt{var} < x_i$, i.e. the parent $s$ is above $x_i$, then $s \longrightarrow t''$ is placed into $Q_{inner:\downarrow}$ to prepare the next invocation of an inner Apply sweep. This matches Case 2b in the Outer Reduce.

The three cases above are depicted in Fig. 9b. For `exists`, Cases 2 and 3 are equivalent to the return from `or` back to `exists`. Case 1 is equivalent to a return statement within the `or`'s own recursion. Case 3 is needed to match 2b with $x_j$ replaced with $x_i$.

Finally, $F'_{inner}$ replaces $F'_{outer}$ and control returns to the outer Reduce sweep to proceed with the levels above $x_j$.

$F_{inner}$ from the inner Apply sweep can be thought of as overlayed on top of $F_{outer}$ from the outer Apply sweep; together they produce a valid (but unreduced) OBDD. The Outer and inner Reduce sweeps work together to reduce this into a single file, $F'_{outer}$. When no more levels, $x_j$, need to be processed and the outer Reduce sweep has finished processing, then $F'_{outer}$ contains the final reduced BDD of all nested operations.

Whereas the Apply–Reduce algorithms in [49] only operate on a singly-rooted DAG, the inner Apply and Reduce sweeps have to operate on a multi-rooted one. Yet, these previous algorithms need not be changed since $Q_{inner:\downarrow}$ is prepopulated with all relevant roots in Cases 1 and 2b in the outer and Case 3 in the inner Reduce sweeps.

Since the result of the inner Apply and Reduce sweeps replaces the entire set of nodes in $F'_{outer}$, the priority queue $Q_{inner:\downarrow}$ not only needs to be populated with requests for the nodes that need to be changed but also with requests for the nodes one wishes to keep (see also Fig. 9a). This makes the inner Apply sweep not only compute the desired result but also act as a mark-and-sweep garbage collection. On the first glance, these additional non-modifying requests may seem too costly – especially if most requests do not modify subtrees. In practice, 33.3% of all requests created throughout our benchmarks (see Section 5 for a detailed presentation thereof) are subtree modifying. For each benchmark instance, 23.0% of all requests modify subtrees on average (median 35.6%). That is, a reasonable number of all requests (and hence BDD nodes processed) change the subgraph in $F'_{outer}$.

### 3.1 Complexity of Nested Sweeping

As mentioned in the description of the outer Apply sweep, nested sweeping works for multiple inputs. In this work, it suffices to assume it only has to deal with a single BDD $f$ of $N$ nodes as also depicted in Fig. 5.

**Lemma 1.** *A single BDD $f$ with $N$ nodes can be transposed in $\Theta(sort(N))$ I/Os and time and $\Theta(N)$ space.*

*Proof.* In $\Theta(\text{scan}(N))$ I/Os and time iterate over and split all nodes $v$ in-order into the two arcs $v.\texttt{uid} \dashrightarrow v.\texttt{low}$ and $v.\texttt{uid} \longrightarrow v.\texttt{high}$. Sort these $2N$ arcs on their target using $\Theta(\text{sort}(N))$ I/Os and time and linear space transposes them.

In Section 3.3, we propose to embed valuable computations inside of the outer Apply sweep. This comes at the cost of potentially changing the BDD size. To encapsulate such cases too, let $N'$ be the output size of the outer Apply sweep which may exceed $\mathcal{O}(N)$. Yet, this step is not the bottle-neck of the algorithm.

**Lemma 2.** *Ignoring the work done within the inner sweeps, the outer Reduce sweep costs $\Theta(sort(N'))$ I/Os and time and requires $\mathcal{O}(N')$ space.*

*Proof.* This follows from the complexity of the Reduce algorithm in [49] (based on [6]) and the constant extra time spent for each of the $N'$ nodes to resolve the additional logic in Cases 1 and 2 of the outer Reduce sweep.

By combining Lemmas 1 and 2 together with the fact that the last invocation of the inner Apply and Reduce sweeps constructs, together with the outer Reduce sweep, the output of size $T$, we obtain the following lower bound on the complexity of nested sweeping.

**Corollary 1.** *Nested Sweeping uses $\Omega(N + T)$ space and $\Omega(sort(N + T))$ time and I/Os where $N$ and $T$ are the size of the input and output, respectively.*

In particular for the `exists` BDD operation, let $T_j$ be the size of $F'_{outer}$ when the inner Apply sweep is invoked at level $x_j$.

**Lemma 3.** *A single invocation of the inner Apply and Reduce sweeps at $x_j$ costs $\Theta(sort(N' + T_j^2))$ I/Os and time and uses $O(N' + T_j^2)$ space.*

*Proof.* As in [49], the algorithm's complexity depends on the number of elements placed in the priority queues [4]. In particular, a single nested `or` sweep deals with up to $2N'$ arcs from $F_{outer}$. On top of these, it also processes up to $2T_j^2$ arcs created during the product construction of $F'_{outer}$.

Since nested sweeping closely simulates the (parallelised) recursive BDD algorithm in Fig. 2, one should expect it achieves, similar to the algorithms in [49], major improvements in the number of I/Os at the cost of a log-factor in the running time when compared to the conventional recursive algorithms. This is indeed the case.

**Proposition 1.** *Quantification of a set of variables, $X$, is computable with nested sweeping in $O(sort(N^{2^{|X|}}))$ I/Os and time and $O(N^{2^{|X|}})$ space.*

*Proof.* Due to Lemma 1, $F_{outer}$ from the outer Apply sweep has up to $2N$ arcs. This is also the size of $F'_{outer}$ without any inner sweeps. Each inner Apply and Reduce sweep may increase the size of $F'_{outer}$ quadratically. The result follows from Lemmas 1 to 3.

Asymptotically, this is not an improvement over just quantifying each variable one-by-one using the algorithm already proposed in the full version of [49]. Yet, doing so would involve $2|X|$ sweeps over *all* levels of the input whereas, as highlighted in Fig. 6, nested sweeping only processes levels below each quantified variable. This difference is also evident in practice: throughout our benchmarks (see Section 5 for details), when quantifying with nested sweeping rather than each variable independently, the total number of requests processed with the `or` operation decreases by 13.9% while the share of 2-ary product constructions increases from 57.3% to 66.6%.

### 3.2 Optimisations for Nested Sweeping

While nested sweeping as described above is an improvement over previous work in [49], there are multiple avenues to further improve its performance in practice.

**Terminal Arcs:** No inner Apply sweep changes the value of terminals. Hence, requests of the form $s \longrightarrow \top$ and $s \longrightarrow \bot$ can be forwarded to $s$ regardless of any nesting levels, $x_j$, in-between. Furthermore, the request based on $t'$ in the outer Reduce sweep may trivially resolve into a terminal. In this case, the resulting terminal can be forwarded to its parents (regardless of their level). For `exists`, this would be if both $t'$.`low` and $t'$.`high` are terminals or either of them is the $\top$ terminal.

This decreases the size of $Q_{inner:\downarrow}$. Furthermore, it makes the requests placed in $Q_{inner:\downarrow}$ compatible with an implicit invariant of the Apply sweep's in [49].

In practice, the number of requests skipped this way depends on the use-case and the scale. 3.7% of all requests processed as part of our benchmarks (see Section 5 for a description) are for terminals. On average, 6.9% of the requests (with a median of 7.0%) are for terminals in each benchmark. For the Garden of Eden (GoE) benchmark specifically, 15.3% of all requests are terminals on average (median 17.7%). On the other hand for the Quantified Boolean Formula (QBF) benchmark, only 5.0% (median 5.9%) of them were.

**Bail-out of Inner Sweep:** There is no need for the outer Reduce sweep to invoke the inner sweeps if $Q_{inner:\downarrow}$ only contains requests that preserve subtrees, i.e. if Case 1 in the Outer Reduce did not create any requests that manipulate the accumulated OBDD in $F'_{outer}$. On level $x_j$, such requests can stem from a redundant node $t'$ being suppressed. For `exists`, this may also occur due to either $t'$.`low` or $t'$.`high` being the $\bot$ terminal, which is neutral for the `or` operation, or being $\top$, which is short-circuiting it.

In this case, the entire content of $Q_{inner:\downarrow}$ can be redistributed between $Q_{outer:\uparrow}$ and $Q_{inner:\downarrow}$ for the next deepest to be quantified level, $x_i$. After doing so, the outer Reduce sweep can immediately proceed processing the next level.

For `exists`, any short-circuiting by the `or` operation in Case 1 of the outer Reduce sweep can kill off some subtrees in $F'_{outer}$. In this case, one cannot skip the last invocation of the inner sweeps. Otherwise, the final result $F'_{outer}$ could include dead nodes. Yet, even so, one can instead of the expensive top-down algorithm, e.g. `or` for `exists`, invoke the inner Apply sweep with a much simpler (and therefore faster) mark-and-sweep algorithm.

In practice, 75.6% of all nested sweeps in our benchmarks (see Section 5 for their presentation) are skippable. For each benchmark, between 6.8% and 93.5% of all nested computations were skipped with an average of 59.2% (median of 81.0%). The number of nested levels depends on the problem domain and its instance. For the Garden of Eden (GoE) benchmark, only 26.8% of all nested computations were skipped on average (median 29.0%), whereas 82.7% (median 84.3%) of all levels of the Quantified Boolean Formulas (QBFs) could be skipped.

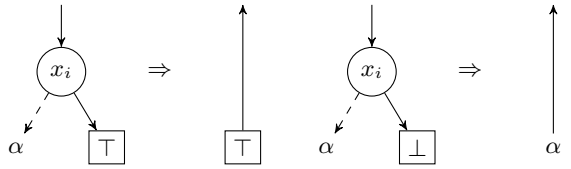
**Root Requests Sorter:** Instead of making the outer Reduce sweep push requests directly into $Q_{inner:\downarrow}$, it can push it into an intermediate list of requests, $L_{outer:\downarrow}$. The content of $L_{outer:\downarrow}$ is sorted using the same ordering as $Q_{inner:\downarrow}$ as the inner Apply sweep is invoked, to then merge it on the fly with theinner Apply sweep's priority queue. This allows one to postpone initialising this priority queu until the inner Apply sweep is invoked. This has multiple benefits:

- $Q_{inner:\downarrow}$, resp. $Q_{inner:\uparrow}$, only exists and uses internal memory during the inner Apply sweep, resp. the inner Reduce sweep. Hence, the memory otherwise dedicated to $Q_{inner:\downarrow}$ can be used in the outer Reduce sweep for the Reduce's per-level data structures in [49]. Furthermore, this also increases the amount of space available to the inner Reduce sweep. Hence, this ought to improve the running time of both the inner and the outer Reduce sweeps.
- In practice, sorting a list of elements once is significantly faster than maintaining an order in a priority queue [41]. Merging $L_{outer:\downarrow}$ on the fly with $Q_{inner:\downarrow}$ is faster than passing requests to the inner sweep's priority queue.
- If $Q_{inner:\downarrow}$, resp. $Q_{inner:\uparrow}$, is initialised for each inner Apply sweep, resp. inner Reduce sweep, then the monotonic and faster *levelised priority queue* in the full version of [49] can be used instead of a regular non-monotonic priority queue.
- Levelised cuts [51] bound the size of each individual inner Apply and Reduce sweep. Hence, for each nested sweep, one can, if it is safe to do so, replace $Q_{inner:\downarrow}$ and/or $Q_{inner:\uparrow}$ with a faster internal memory variant.
- Levelised random access [50] may need to change the sorting predicate in $Q_{inner:\downarrow}$. Hence, $L_{outer:\downarrow}$ allows this optimisation to be applied for each invocation of the inner Apply sweep depending on the width of $F'_{outer}$.

Furthermore, levelised cuts not only bound the size of $Q_{outer:\uparrow}$ in the outer Reduce sweep but also the size of $L_{outer:\downarrow}$. Hence, while deciding whether $Q_{outer:\uparrow}$ fits into memory, one can also decide whether $L_{outer:\downarrow}$ does.

All in all, this allows the optimisations in [50, 51] to be applied on a sweep-by-sweep basis. In practice, if one neither uses faster internal memory variants of $Q_{inner:\downarrow}$ and $Q_{inner:\uparrow}$ nor levelised random access, then Adiar needs a total of 32.1 h to solve 145 out of the 147 benchmarks in Section 5. Using these two optimisations shaves 13.0 h off the total computation time (speedup of 1.68). For each individual instance, this improves Adiar's performance between a factor of 1.07 and 5.05 (1.77 on average)[1]. Furthermore, without $L_{outer:\downarrow}$, the exponential blow-up in Proposition 1 implies $Q_{inner:\downarrow}$ would almost always have to use external memory. As the optimisations in [49–51] would then not be applicable, one would expect a slowdown of several orders of magnitude similar to [50, 51].

[1] Compared to [51], the *external* memory sorters in this comparison still use the levelised cuts to circumvent wasting time with initialising too much internal memory. This is why, there is not a speedup of several orders of magnitude. If this use of levelised cuts is also reverted to obtain its state back in [49], then preliminary experiments on a machine with 8 GiB of memory exhibits a speedup of 2.71 on average. As memory increases, one should expect a difference similar to the one reported in [51]




(a) Pruning due to $\top$. (b) Skipping node due $\bot$.

Fig. 10: Example of pruning quantification of a to-be quantified level $x_i$.

### 3.3 Optimisations for Quantification

As presented above, the outer Apply sweep merely transposes the input BDD $f$. Yet, doing so may not make the most out of having to touch the entire BDD graph; as long as the result is a transposed BDD for, one can incorporate additional computations inside of this sweep. Hence, we now explore possible top-down sweeps that can be used instead of the algorithm in Lemma 1.

**Pruning $\top$ Siblings:** Corollary 1 and Proposition 1 show a possibly wide gap in the potential performance of the nested `exists` algorithm. Lemma 3 shows this stems from the possibility of some partially quantified result explodes exponentially in size. Yet, $T_j$ can only be larger than $T$ if it contains subtrees that will be pruned or merged later when another variable is quantified. This can only happen due to the $\top$ terminal shortcutting an `or`. Hence, to be closer to the lower bound in Corollary 1, we need to identify redundant computation by pushing information about the $\top$ terminal down through the BDD of $f$.

As shown in Fig. 10a, one can collapse to be quantified nodes at a level $x_i$ if one of their children is the $\top$ terminal. Similarly, as shown in Fig. 10b, one can skip over nodes with a $\bot$ terminal as its child. This can be done as part of a simple top-down sweep similar to the Restrict in the full version of [49].

In the worst-case, this does not apply to any node in $f$ and so the output is similar to the algorithm in Lemma 1. Our preliminary experiments indicate this approach introduces an overhead of up to 2%. Yet, if nodes are prunable, making $N' < N$, then total performance can improve with up to 21%.

**Deepest Variable Quantification:** The single-variable quantification in the full version of [49] can also be used to transpose $f$. This removes one of the to be quantified variables $x_i \in X$ in $\mathcal{O}(\mathrm{sort}(N^2))$ I/Os and time and $\mathcal{O}(N^2)$ space. The resulting transposed graph, $F_{outer}$, has size $N' \leq N^2$. To not change the overall memory usage, one can choose $x_i$ to be the largest to be quantified variable. Doing so makes the levels at $x_i$ and below equivalent to $F_{inner}$ after the first inner Apply and Reduce sweep. That is, $N' \leq N + T_i$ and one saves an entire nested sweep at no cost to memory usage.

Our preliminary experiments indicate this only slows down computation time on average by 4.7%. We hypothesise this is due to the deepest variable $x_i$ is often

close to the bottom of the BDD and so, this sweep is primarily transposing the graph with more complex logic.

One can also incorporate the above pruning of $\top$ siblings inside this quantification sweep. This improves performance for applicable cases. But, it does not offset the additional overhead in the remaining cases.

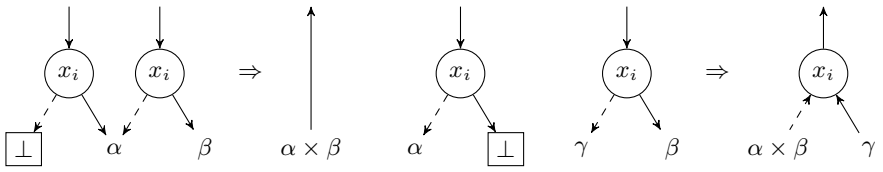
**Partial Quantification:** The single-variable quantification in the full version of [49] can be generalised to partially resolve all $x_i \in X$ in a single sweep. The requests in [49] are for pairs of nodes $(t_1, t_2) \in f \times f$.

Without loss of generality, assume $t_1.\texttt{var} = t_2.\texttt{var} = x_i \in X$. In this case, the 2-ary product construction should turn into $(t_1.\texttt{low}, t_1.\texttt{high}, t_2.\texttt{low}, t_2.\texttt{high})$. If any of these four `uid`s are the $\top$ terminal then the entire request can immediately be resolved to $\top$. Furthermore, any $\bot$ terminal is neutral to the `or` operation and can be pruned from the 4-tuple. Similarly, any duplicate `uid`s can be merged. Since `or` is commutative, one can sort the 4-tuple to quickly identify these cases. If the resulting tuple has 2 or fewer entries remaining, the product construction can proceed as in the full version of [49] (Fig. 11a). Otherwise, a new node with $x_i$ is created with the result of each half of the 4-tuple as its children (Fig. 11b). Inductively, this is correct (after later quantification of the new $x_i$ node and its subtrees) as the `or` operation is associative.

The resulting DAG is a 2-ary product construction of $f$, and so $F_{outer}$ has size $N' \leq N^2$. As per [49], this single sweep is computable in $\mathcal{O}(\text{sort}(N^2))$ I/Os and time and $\mathcal{O}(N^2)$ space.

Similar to the two $\top$ terminal pruning above, partial quantification prunes shortcutted subtrees across all levels of $f$. Furthermore, similar to deepest quantification, it leaves at least one fewer levels of to be quantified variables to be processed later.

Our preliminary experiments indicate, partial quantification can in practice improve performance up to 61%. Yet, many other intances slow down just as much (up to 115%, i.e. also a bit more than a factor of two). We hypothesise this is due to partial quantification pairing nodes with a conflicting assignment. For example, in Fig. 11b $\alpha$ is paired with $\beta$ rather than $\gamma$. Oddly enough, in our preliminary experiments, the instances that were improved by $\top$ pruning




(a) Fully quantified pair of nodes (b) Partially quantified pair of nodes

Fig. 11: Example of partial quantification of a level $x_i$.

are disjoint from the ones improved by partial quantification. Further research is needed to investigate why and when partial quantification is useful.

**Repeated Partial Quantification:** The top-down sweep aboves produces a transposed and unreduced OBDD. Yet, it is in practice possible that $N'$ is smaller than $(1+\epsilon)\cdot N$ for some $\epsilon \in \mathbb{Q}$, i.e. its size has not grown considerable. In this case, the resulting product construction has very few new BDD nodes that are potentially reducible. Hence, it may be more beneficial to untranspose the OBDD and then immediately rerun another transposing top-down sweep. Doing so with pruning or partial quantification can propagate the $\top$ terminal further and so prune more subtrees. Yet, it is unlikely that pruning $\top$ terminals in Fig. 10a makes said terminal available for another to be quantified variable. That is, it is unlikely in this case that a second sweep would further prune subtrees. Hence, this is most promising to do with partial quantification.

Since there are very few new BDD nodes, it is unlikely that the Reduce sweep of [49] will do much more than just untranspose the DAG. Hence, one would want to untranspose it with a simpler and faster algorithm. As can be seen in Fig. 3, one can instead merely sort all arcs on their source and then merge them on the fly into nodes. Asymptotically, this is still a $\Theta(\mathrm{sort}(N'))$ operation. But, the constant involved is smaller than the Reduce of [49].

Hence, one can repeat the above partial quantification operation until $N'$ exceeds $(1+\epsilon)\cdot N$, it has run $\delta$ times, or there are no more to be quantified variables left in $F_{outer}$. In practice, we have not yet found any instance where more than a single quantification sweep further improves performance. Hence, as further research hopefully uncovers when it is beneficial to use partial quantification, we can extrapolate this into a value of $\delta$.

## 4 Implementation of Nested Sweeping in Adiar

Most of the logic in Section 3 can be implemented by wrapping the priority queues $Q_{outer:\uparrow}$, $Q_{inner:\downarrow}$, and $Q_{inner:\uparrow}$ and $L_{outer:\downarrow}$ with additional logic on how to merge and whereto split requests.

- The logic of whether to push to $Q_{outer:\uparrow}$ or $L_{outer:\downarrow}$ in the outer Reduce sweep, resp. Cases 2 and 3 in the inner Reduce sweep, is a conditional on level $x_j$, resp. $x_i$.
- During the inner Apply sweep, requests from $L_{outer:\downarrow}$ are merged on the fly with the ones pushed to $Q_{inner:\downarrow}$.
- When placing requests in $L_{outer:\downarrow}$, they are marked as originating from the outer Reduce sweep. During the inner Reduce sweep, requests are forwarded to $Q_{outer:\uparrow}$ or $Q_{inner:\uparrow}$ depending on whether they are marked to be from the outer sweep or not.

This has been implemented in Adiar v2.0 with (compile-time known) *decorators*: a class with the same interface as the priority queues runs the above logic before

passing it onto the wrapped priority queues and sorters. This makes the logic of each sweep agnostic to and reusable in the context of nested sweeping.

The nested sweeping framework, i.e. the decorators, $L_{outer:\downarrow}$, and the algorithm and its optimisations, has been implemented with 1287 lines of templated C++ classes and functions. Similar to [48, 51], the use of templates completely remove any indirection and abstraction introduced for the sake of code quality. The entire framework has been tested separately from the remaining codebase with 104 unit tests. The quantification algorithms themselves grew from 548 lines of code and 84 unit tests to 1152 lines of code and 152 unit tests (without any of the optimisations in Section 3.3).

## 5 Experimental Evaluation

To evaluate the impact of using nested sweeping, we have run experiments aiming at answering the following three research questions:

1. How does nested sweeping compare to the repeated use of the single-variable quantification from the full version of [49]?
2. How does Adiar with nested sweeping compare to the external memory BDD package, CAL [46]?
3. How does Adiar with nested sweeping compare to conventional BDD packages that use depth-first recursion and memoisation [9, 21, 27, 33, 52]?

### 5.1 Benchmarks

For this evaluation, we have implemented the following two benchmarks that rely on multi-variable quantification. Similar to [48, 49], all benchmarks have been implemented on top of C++-templated adapters for each BDD package. This makes each BDD package run the exact same set of operations without introducing any indirection. The source code for all benchmarks can be found at the following url:

github.com/ssoelvsten/bdd-benchmark

**QBF Solving:** Given a Quantified Boolean Formula (QBF) in the `QCIR` [54] format, each gate of the given circuit is recursively transformed into a BDD. For inputs, we use the 102 encodings from [47] of 2-player games on a grid. In our experience, the symbolic style of these inputs makes them well suited to be solved with BDDs. Hence, they provide a typical use-case of quantification in BDDs. Furthermore, though these inputs are not in CNF they are in prenex form. In practice, resolving these prenex quantifications at the end is computationally much more expensive than computing the to be quantified circuit, i.e. the *matrix*.

Based on preliminary experiments, we use a variable order based on a depth-first traversal of the given circuit. In the prenex, we merge adjacent blocks with the same quantifier to increase the number of concurrently quantified variables.

**Garden-of-Eden:** In a cellular automaton, a *Garden-of-Eden* [42] (GoE) is any configuration without a predecessor. In Conway's Game of Life [24], recent results show there exists no GoE of size $8 \times 8$ or smaller [8]. Hence, the BDD for an $n_r \times n_c \leq 8 \times 8$ sized transition relation will collapse to $\top$ when all of its previous state variables are existentially quantified. Yet, a row-major encoding of the transition relation requires only a polynomially sized BDD. Hence, the complexity of this problem manifests as an explosion of the BDD's size during the existential quantification.

The transition relation on a grid of size $n_r \times n_c$ is encoded with $(n_r+2)\cdot(n_c+2)$ *previous* state variables, $\vec{x}$, and (up to) $n_r \cdot n_c$ *next* state variables, $\vec{x'}$. By reusing *next* state variables for multiple cells, one can restrict the search for symmetric GoEs. Post state variables, $x'_i \in \vec{x'}$, follow a row-major order. Previous state variables, $x_i \in \vec{x}$, are interleaved to directly preceede their respective post state variable, $x'_i$.

### 5.2 Hardware and Settings

As in [48–51], we have run our experiments on the *Grendel* cluster at the Centre for Scientific Computing Aarhus. In particular, we ran both benchmarks on machines with 48-core 3.0 GHz Intel Xeon Gold 6248R processors, 384 GiB of RAM, 3.5 TiB of SSD disk, and run Rocky Linux (Kernel 4.18.0-513). All code was compiled with GCC 10.1.0 or `rustc` 1.72.1. Each BDD package was given $\frac{9}{10}$th of the available RAM, i.e. 345 GiB, leaving $\frac{1}{10}$th to other data structures and the operating system. Next to that, the BDD packages use a single thread and their default/recommended settings.

Note that these machines have vasts amounts of memory. This is to ensure that depth-first implementations are not slowed down by external factors. If less memory is available, then depth-first implementations would have to run multiple garbage collections to stay within the memory limits (cf. the largest instances solved by BuDDy [33] in Fig. 14a). This, in turn, clears their memoisation tables and forces them to recompute previous results. Furthermore, this large amount of memory ensures they can solve larger problems without using the swap partition. If they had to use it then they would slow down by about two orders of magnitude (see the full paper of [49] for an example). Hence, machines of this scale allow us to measure the algorithms' running time without the noise otherwise introduced by their execution environment. Finally, this biases the running time in favour of the depth-first implementations, which in turn makes the numbers we report on Adiar's relative performance close to the worst-case.

### 5.3 Experimental Results

The computing cluster's scheduler does not let many long-running jobs run concurrently. To obtain all 1176 data points reported below within only a few months, we had to place each of the 147 instances in buckets of instances with a common timeout. In particular, an instance is placed in the bucket with the

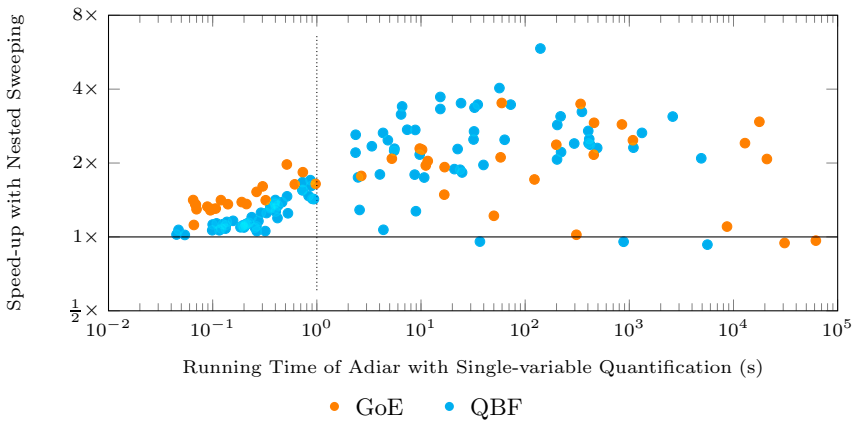



Fig. 12: Relative time ($T_{\text{old}}/T_{\text{new}}$) of quantification with nested sweeping ($T_{\text{new}}$) compared to the previous repeated single-variable quantification ($T_{\text{old}}$).

smallest timeout that is four times larger than Adiar needed during preliminary experiments. That is, a BDD package timing out should only be understood as it (possibly) being considerably slower than Adiar.

Depending on an instance bucket placement, running time measurements were made 1 to 3 times. Due to node failures on the cluster, Adiar with nested sweeping was run once more, resulting in its measurements being repeated on many instances 4 times. On average, all data points had 3.0 measurements. Similar to [48,49,51], we report for each benchmark the minimum time recorded as it is the measurement with the least noise [17]. Average ratios are aggregated using the geometric mean.

Adiar needs less than 1 s to solve 27 out of the 45 GoE instances, resp. 50 out of the 102 QBF instances. As will become evident later with Figs. 13 and 14, this makes them so small that they are not within the current scope of Adiar. For completeness, we still show and discuss these results.

**RQ 1: Improvement by Nested Sweeping** Figure 12 shows the speed-up of using Adiar with nested sweeping (without any optimisations in Section 3.3) relative to quantifying each variable individually. Across all instance sizes, nested sweeping is in general an improvement in performance. We have recorded a slowdown of up a factor of 1.05 for 5 instances. Yet, we also recorded speed-ups up to a factor of 5.88 for the 142 remaining instances. On average, performance improves by a factor of 1.7 for both QBF and GoE. The total computation time was decreased by 21% from 49.4 h to 39.1 h.

**RQ 2: Comparison to CAL** To the best of our knowledge, CAL [46] (based on [7,43]) is the only other BDD package also designed to manipulate BDDs larger than main memory. To do so, it uses breadth-first algorithms that should work

Table 1: Time taken and the average ratio between Adiar and CAL for the 124 commonly solved instances. The average (geometric mean) pertains only to instances where Adiar needed at least 1 s to solve them. Ratios larger than 1.00 means Adiar is faster.

| | Time | | # Solved | | Avg. Ratio (1+s) | |
|---|---|---|---|---|---|---|
| | • GoE | • QBF | • GoE | • QBF | • GoE | • QBF |
| Adiar | 7431.9s | 688.0s | 45 | 102 | – | – |
| CAL | 184688.3s | 295660.0s | 38 | 86 | 5.0 | 25.2 |

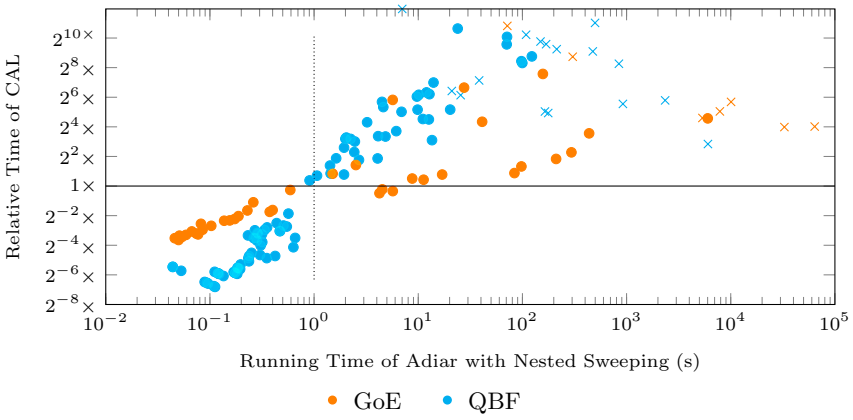



Fig. 13: Relative performance ($T_{\text{CAL}}/T_{\text{Adiar}}$) of CAL ($T_{\text{CAL}}$) compared to Adiar with Nested Sweeping ($T_{\text{Adiar}}$). Time-/Memouts are marked as crosses.

well with BDDs stored on disk via the operating system's swap memory [46]. CAL also includes algorithms to support multi-variable quantification [46]. For more details, see [46] and Section 6. The machines for our experiments provide a 48 GiB swap partition, i.e. a 12.5% increase in available space.

Preliminary experiments indicated CAL's breadth-first algorithms are much slower than Adiar's time-forward processing. Hence, we multiplied the timeout for CAL by a factor of 3. But as is evident in Fig. 13, this increase turned out to still overestimate CAL's performance on larger instances. Hence, the running times and averages in Fig. 13 and Table 1 pertain only to the 124 instances which CAL can solve within the given RAM, SWAP, and the time limits.

Even though this discards the instances where CAL struggles, i.e. the data points that remain are in CAL's favour, Fig. 13 shows Adiar heavily outperforms CAL for instances where Adiar takes 1 s or longer to solve. Where CAL uses 133.4 h to solve 124 instances, Adiar, by solving them in only 2.3 h, is 59.1 times faster. On these larger instances, CAL is on average 14.7 times slower than Adiar. As is evident in Fig. 13 and Table 1, Adiar especially outperforms CAL on the QBF benchmark. For example, the largest difference was measured

for the `hex/hein_15_5x5-13` QBF instance, where CAL is 1081 times slower than the 71.2 s Adiar needs to solve it.

On the other hand, CAL is considerably faster for the instances where Adiar takes less than 1 s to solve. At this smaller scale, CAL primarily uses conventional depth-first algorithms; doing the same for Adiar is still left as future work [50,51].

**RQ 3: Comparison to Depth-First Implementations** For this comparison, we have compared performance with BuDDy 2.4 [33], CUDD 3.0.0 [52], LibBDD 0.5 [9], OxiDD 0.6 [27], and Sylvan 1.8.1 [21]. Their individual performance relative to Adiar is shown in Fig. 14. Out of the 147 instances, 140 are solved by all depth-first BDD packages, i.e. the remaining 7 instances have at least one BDD package running out of memory (MO) or time (TO). Adiar solves all of them. Running out of time is most likely due to repeated need for garbage collection, which essentially is equivalent to an MO. Yet for fairness, Table 2 shows the total time for these 140 commonly solved instances. The average ratio, on the other hand, pertains to all instances solved by the respective BDD package.

As shown in Table 2, Adiar solves the 40 common GoE instances in 2.7 h. This makes it 1.13 times faster than CUDD and 2.20 times faster than OxiDD at solving all benchmarks. Adiar further solves the 100 common QBF instances in 1.25 h. This makes it as fast as CUDD and 1.3 times faster than Sylvan at solving these QBF instances.

The relative running time of BuDDy in Fig. 14a and OxiDD in Fig. 14d shows that Adiar's performance can be divided into three categories: the *small* instances that takes Adiar less than 1 s to solve but is out of its (current) scope, the *medium* instances where Adiar needs between 1 and $10^3$ s to solve and it is up to a constant factor of 4 slower than other BDD packages, and the *large* instances beyond $10^3$ s where other BDD packages slow down compared to Adiar due to limited internal memory and repeated garbage collection. While the distinction is not as clear for CUDD in Fig. 14b and Sylvan in Fig. 14e, they also follow the

Table 2: Total time needed by Adiar and conventional depth-first implementations to solve the 140 commonly solved instances. The average (geometric mean) covers all instances that were commonly solved by all BDD packages and where Adiar needed at least 1 s to solve. Ratios larger than 1.00 means Adiar is faster.

| | Time | | # Solved | | Avg. Ratio (1+s) | |
|---|---|---|---|---|---|---|
| | • GoE | • QBF | • GoE | • QBF | • GoE | • QBF |
| Adiar | 9655.7s | 4499.4s | 45 | 102 | – | – |
| BuDDy | 4725.5s | 3793.1s | 40 | 100 | 0.30 | 0.25 |
| CUDD | 10892.8s | 4591.9s | 40 | 101 | 0.61 | 0.75 |
| Lib-BDD | 4365.7s | 2687.3s | 43 | 101 | 0.54 | 0.45 |
| OxiDD | 21223.9s | 2379.6s | 41 | 101 | 0.48 | 0.39 |
| Sylvan | 2925.4s | 5841.4s | 44 | 102 | 0.46 | 0.70 |

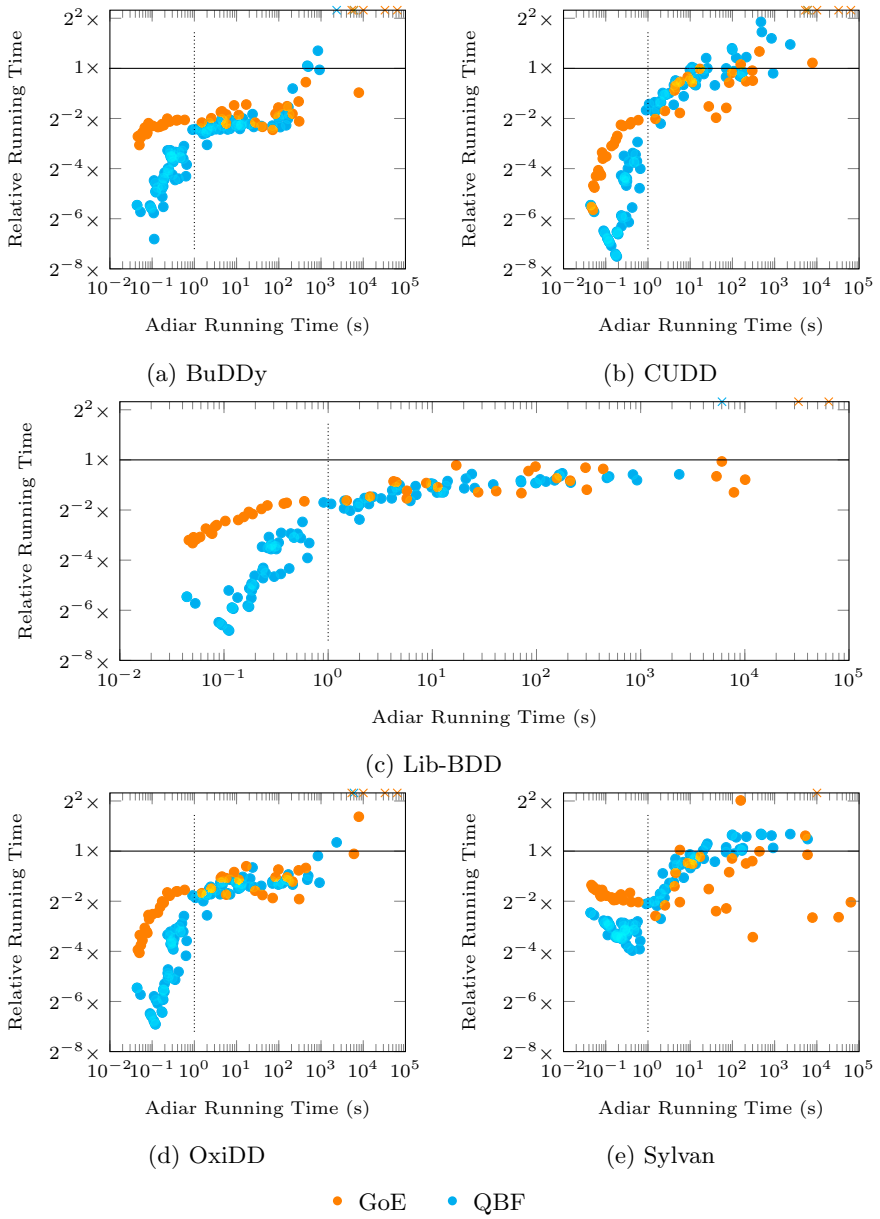



Fig. 14: Relative performance of depth-first implementations compared to Adiar with nested sweeping. Time-/Memouts are marked as crosses.

same trend. This relative performance is similar to the results in [48–51]. That is, nested sweeping allows Adiar to compute quantifications at no additional cost to previous work.

LibBDD in Fig. 14c is the only BDD package consistently faster than Adiar (ignoring its three MOs). Most likely, this is due to its lack of a shared unique node table. This makes the expensive garbage collection steps obsolete; instead, the memory is merely freed. Hence, either LibBDD can fit its BDD computations into the internal memory (and it is faster than Adiar) or it aborts.

As is evident from Fig. 14e, Sylvan is comparatively good at some of the larger GoE instances, thereby beating all other BDD packages in the total time to solve the GoE benchmarks. As the BDD collapses to $\top$, one may expect this is due to Sylvan skipping the second recursive calls to `exists` if the first recursion resulted in $\top$. Yet, CUDD also includes this optimisation without exhibiting the same behaviour. Further investigation is needed to identify how Sylvan excels on these instances. Sylvan is also the only other BDD package able to solve all QBF instances within the given time limit, in parts thanks to its small memory footprint per BDD node [21]. Yet, Sylvan requires a total of 5.0 h to solve all 102 QBF instances whereas Adiar only needed 3.6 h, making Sylvan 1.4 slower than Adiar.

## 6 Related Work

Many other implementations of BDDs also support quantification of multiple variables. All these are based on a nested (inner) operation being accumulated in an (outer) traversal of the input; the nested sweeping framework achieves the same within the time-forward processing paradigm [4, 18] of Adiar's algorithms.

**CAL:** The CAL [46] BDD package (based on [7, 43]) is to the best of our knowledge the only implementation of BDDs also designed to compute on BDDs whose size exceed main memory. To do so, it uses breadth-first algorithms that are resolved level by level. For each level it still follows the conventional approach: a unique node table is used to manage BDD nodes while a polynomial running time is guaranteed by use of a memoisation table. These per-level hash tables, both in theory and in practice, put an upper bound on the maximum BDD width that CAL can support with a certain amount of internal memory [6].

Its quantification operation also required additional ideas particular to the design of CAL. Since it uses a single breadth-first queue for each level, each queue contains requests for both the outer and the nested inner traversals. Hence, both can be – and are – processed simultaneously [46]. Furthermore, it switches between breadth- and depth-first evaluation of subtrees to improve performance: the outer traversal is depth-first for the BDD nodes with to be quantified variables and breadth-first otherwise. If the first subtree's quantification makes computing the other ones redundant, then all computation of the second is skipped. These depth-first steps are also placed in the very same queues as the breadth-first steps; doing so ensures no additional random access is introduced.

By the nature of nested sweeping, our proposed algorithm is, unlike CAL, not easily able to skip redundant computations. In Section 3.3 we investigate multiple promising avenues to achieve similar pruning of redundant computation. Furthermore, the lack of a unique node table in Adiar requires our algorithms to retraverse and copy the subtrees that are unchanged. Even so, as evident in Section 5, Adiar with nested sweeping outperforms CAL by up to several orders of magnitude. Moreover, the I/O-efficient approach in [6, 49], and by extension the ones in this work, are, unlike CAL, I/O efficient despite a BDD's level is wider than main memory.

**Distribution Sweeping:** In the context of computational geometry, *distribution sweeping* [25] is an I/O-efficient translations of internal memory sweepline algorithms. Here, the recursion is turned on its head: the recursive but I/O inefficient data structure is replaced with an I/O-efficient list and the iterative algorithm is instead turned into a recursive one. Specifically, all the points in the plane are sorted on the $x$-axis and distributed into $M/B$ vertical *strips* (see Section 2.1 on the I/O-model). After these strips have been solved recursively, an $M/B$-way merge procedure both merges and prunes all strips into one while simultaneously recreating a vertical sweepline moving across all strips [12, 25].

In our case of translating the `exists` algorithm (see Fig. 2), we also intend to move the recursion out of a data structure, namely out of the BDD. Unlike for distribution sweeping, we do not intend to divide-and-conquer the input but instead recurse through the dependencies of the algorithm's recursion, e.g. between the independent calls to `exists` and the nested `or` operation that depends on their result. Independent recursions are resolved simultaneously with regular time-forward processing sweeps as in [49]. Dependencies are handled by moving requests from the priority queue of one time-forward processing sweep to the priority queue of another. When all dependencies have been moved, the current sweep is paused to then start a *nested sweep* – the results of which are in turn parsed to its dependencies.

## 7 Conclusions and Future Work

Each sweep in [49] is independent of the others. Using only this approach, one can only quantify a single variable a time but not multiple at once. In this work, we enable multi-variable quantification with the *nested sweeping* framework. Here, multiple sweeps work together: each sweep forwards information within priority queues to itself, its parent, or its child in a recursion stack.

In practice, nested sweeping has improved the total time that Adiar needs to solve our quantification benchmarks by 21%. On average, it improves each instance's running time by a factor of 1.7. This allows us to extend the results in [48–51] to Adiar's quantification operations: ignoring small instances, Adiar is at most 4 times slower than conventional depth-first implementations. Adiar even outperforms depth-first implementations as they get closer to the limits of internal memory. As Adiar's nested sweeping algorithms are implemented on-top

of the I/O-efficient data structures that were also used in [48–51], its performance is unaffected by a limited internal memory [49]. For example, whereas CUDD [52] could solve 141 out of our 147 benchmark instances in 5.6 h, Adiar needed only 4.6 h to do the same; Adiar could also solve the remaining 6 instances. On average, Adiar is only 1.3 times slower than CUDD for the instances that CUDD could solve.

Adiar is also faster, often by one or more orders of magnitude, than the only other existing external memory BDD package, CAL [46].

The nested sweeping framework has already been generalised to pave the way for the implementation of other multi-recursive BDD operations. We intend to use it for the relational product and functional composition which are both used in model checking such as [28]; in particular, the I/O-efficient relational product still requires optimisations for its variable relabelling and its combined `and-exists`. Furthermore, we hope to also use nested sweeping as the foundation for novel I/O-efficient variable reordering procedures. Finally, nested sweeping opens up the possiblity to create an I/O-efficient implementation of other types of decision diagrams. For example, both Quantum Multiple-valued Decision Diagrams [38] and Polymial Boolean Rings [11] require nested sweeps to implement their multiplication operations.

### Acknowledgements

Thanks to the Centre for Scientific Computing, Aarhus, for running our benchmarks on the Grendel cluster and thanks to Marijn Heule and Randal E. Bryant for suggesting the Garden of Eden problem and their ideas on how to encode it.

## References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM **31**(9), 1116–1127 (1988). https://doi.org/10.1145/48529.48535
2. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers **C-27**(6), 509–516 (1978). https://doi.org/10.1109/TC.1978.1675141
3. Amparore, E.G., Donatelli, S., Gallà, F.: starMC: an automata based CTL* model checker. PeerJ Comput. Sci. **8**, e823 (2022)
4. Arge, L.: The buffer tree: A new technique for optimal I/O-algorithms. In: Algorithms and Data Structures. pp. 334–345. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
5. Arge, L.: The I/O-complexity of ordered binary-decision diagram manipulation. In: Proceedings of International Symposium on Algorithms and Computations, ISAAC'95. pp. 82 – 91 (1995)
6. Arge, L.: The I/O-complexity of ordered binary-decision diagram manipulation. In: Efficient External-Memory Data Structures and Applications (PhD Thesis), pp. 123 – 145. Aarhus Universitet, Datalogisk Institut, Denmark (08 1996)
7. Ashar, P., Cheong, M.: Efficient breadth-first manipulation of binary decision diagrams. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 622–627. IEEE Computer Society Press (1994). https://doi.org/10.1109/ICCAD.1994.629886

8. Beer, R.D.: Cultivating the garden of Eden. arXiv (2022). https://doi.org/10.48550/arXiv.2210.07837
9. Beneš, N., Brim, L., Kadlecaj, J., Pastva, S., Šafránek, D.: AEON: Attractor bifurcation analysis of parametrised Boolean networks. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 12224, pp. 569 – 581. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_28
10. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: 27th Design Automation Conference (DAC). pp. 40–45. Association for Computing Machinery (1990). https://doi.org/10.1109/DAC.1990.114826
11. Brickenstein, M., Dreyer, A.: PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. Journal of Symbolic Computation **44**(9), 1326–1345 (2009). https://doi.org/10.1016/j.jsc.2008.02.017
12. Brodal, G.S., Fagerberg, R.: Cache oblivious distribution sweeping. In: Automata, Languages and Programming. pp. 426–438. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
13. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677 – 691 (1986)
14. Bryant, R.E., Biere, A., Heule, M.J.H.: Clausal proofs for pseudo-Boolean reasoning. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 443–461. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_25
15. Bryant, R.E., Heule, M.J.H.: Dual proof generation for quantified Boolean formulas with a BDD-based solver. In: Automated Deduction – CADE 28. pp. 433–449. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_25
16. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 12651, pp. 76–93. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_5
17. Chen, J., Revels, J.: Robust benchmarking in noisy environments. arXiv (2016), https://arxiv.org/abs/1608.04295
18. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 139—-149. SODA '95, Society for Industrial and Applied Mathematics (1995)
19. Ciardo, G., Miner, A.S., Wan, M.: Advanced features in SMART: the stochastic model checking analyzer for reliability and timing. SIGMETRICS Perform. Evaluation Rev. **36**(4), 58–63 (2009)
20. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer **2**, 410–425 (2000). https://doi.org/10.1007/s100090050046
21. Van Dijk, T., Van de Pol, J.: Sylvan: multi-core framework for decision diagrams. International Journal on Software Tools for Technology Transfer **19**, 675–696 (2016). https://doi.org/10.1007/s10009-016-0433-2
22. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based boolean functional synthesis. In: Computer Aided Verification. pp. 402–421. Springer International Publishing (2016)
23. Gammie, P., Van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 3114, pp. 479–483. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_41

24. Gardner, M.: The fantastic combinations of John Conway's new solitaire game "life". Scientific American **223 No. 4**, 120–123 (10 1970). https://doi.org/10.1038/scientificamerican1070-120
25. Goodrich, M., Tsay, J.J., Vengroff, D., Vitter, J.: External-memory computational geometry. In: Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science. pp. 714–723 (1993). https://doi.org/10.1109/SFCS.1993.366816
26. He, L., Liu, G.: Petri net based symbolic model checking for computation tree logic of knowledge. arXiv (2020), https://arxiv.org/abs/2012.10126
27. Husung, N., Dubslaff, C., Hermanns, H., Köhl, M.A.: OxiDD: A safe, concurrent, modular, and performant decision diagram framework in Rust. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24). Lecture Notes in Computer Science, vol. 14572. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_13
28. Kant, G., Laarman, A., Meijer, J., Van de Pol, J., Blom, S., Van Dijk, T.: LTSmin: High-performance language-independent model checking. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
29. Karplus, K.: Representing boolean functions with if-then-else DAGs. Tech. rep., University of California at Santa Cruz, USA (1988)
30. Klarlund, N., Rauhe, T.: BDD algorithms and cache misses. In: BRICS Report Series. vol. 26 (1996). https://doi.org/10.7146/brics.v3i26.20007
31. Lee, C.Y.: Representation of switching circuits by binary-decision programs. The Bell System Technical Journal **38**(4), 985 – 999 (1959). https://doi.org/10.1002/j.1538-7305.1959.tb01585.x
32. Lin, Y., Tabajara, L.M., Vardi, M.Y.: ZDD Boolean synthesis. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 64–83. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_4
33. Lind-Nielsen, J.: BuDDy: A binary decision diagram package. Tech. rep., Department of Information Technology, Technical University of Denmark (1999)
34. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. International Journal on Software Tools for Technology Transfer **19**, 9–30 (2017). https://doi.org/10.1007/s10009-015-0378-x
35. Madsen, M., Van de Pol, Jaco: Polymorphic types and effects with Boolean unification. Proc. ACM Program. Lang. **4**(OOPSLA) (11 2020). https://doi.org/10.1145/3428222
36. Madsen, M., Van de Pol, Jaco, Henriksen, T.: Fast and efficient boolean unification for Hindley-Milner-style type and effect systems. Proceedings of the ACM on Programing Languages **7**(OOPSLA 2) (10 2023). https://doi.org/10.1145/3622816, https://doi.org/10.1145/3622816
37. Michaud, T., Colange, M.: Reactive synthesis from LTL specification with Spot. In: 7th Workshop on Synthesis, SYNT@ CAV. vol. 5 (2018)
38. Miller, D., Thornton, M.: QMDD: A decision diagram structure for reversible and quantum circuits. In: 36th International Symposium on Multiple-Valued Logic. pp. 30–36 (2006). https://doi.org/10.1109/ISMVL.2006.35
39. Minato, S.i., Ishihara, S.: Streaming BDD manipulation for large-scale combinatorial problems. In: Design, Automation and Test in Europe Conference and Exhibition. pp. 702–707 (2001). https://doi.org/10.1109/DATE.2001.915104
40. Minato, S.i., Ishiura, N., Yajima, S.: Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In: 27th Design Au-

tomation Conference. pp. 52–57. Association for Computing Machinery (1990). https://doi.org/10.1145/123186.123225
41. Mølhave, T.: Using TPIE for Processing Massive Data Sets in C++. SIGSPATIAL Special **4**(2), 24–27 (2012). https://doi.org/10.1145/2367574.2367579
42. Moore, E.F.: Machine models of self-reproduction. In: Proceedings of Symposia in Applied Mathematics. vol. 14, pp. 17–33 (1962). https://doi.org/10.1090/psapm/014
43. Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: International Conference on Computer Aided Design (ICCAD). pp. 48–55. IEEE Computer Society Press (1993). https://doi.org/10.1109/ICCAD.1993.580030
44. Pastva, S., Henzinger, T.: Binary decision diagrams on modern hardware. In: Conference on Formal Methods in Computer-Aided Design. pp. 122–131 (2023)
45. Renkin, F., Schlehuber-Caissier, P., Duret-Lutz, A., Pommellet, A.: Dissecting ltlsynt. Formal Methods in System Design **61**, 248–289 (2022). https://doi.org/10.1007/s10703-022-00407-6
46. Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.: High performance BDD package by exploiting memory hierarchy. In: 33rd Design Automation Conference (DAC). pp. 635–640. Association for Computing Machinery (1996). https://doi.org/10.1145/240518.240638
47. Shaik, I., Van de Pol, J.: Concise QBF encodings for games on a grid (extended version). arXiv (2023). https://doi.org/10.48550/arXiv.2303.16949
48. Sølvsten, S.C., Van de Pol, J.: Adiar 1.1: Zero-suppressed decision diagrams in external memory. In: NASA Formal Methods Symposium. LNCS 13903, Springer, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-33170-1_28
49. Sølvsten, S.C., Van de Pol, J., Jakobsen, A.B., Thomasen, M.W.B.: Adiar: Binary decision diagrams in external memory. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 13244, pp. 295–313. Springer, Berlin, Heidelberg (2022). https://doi.org/10.1007/978-3-030-99527-0_16
50. Sølvsten, S.C., Rysgaard, C.M., Van de Pol, J.: Random access on narrow decision diagrams in external memory. In: Model Checking Software. pp. 137–145. Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_7
51. Sølvsten, S.C., Van de Pol, J.: Predicting memory demands of BDD Operations using maximum graph cuts. In: André, É., Sun, J. (eds.) Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 14216, pp. 72–92. Springer (2023). https://doi.org/10.1007/978-3-031-45332-8_4
52. Somenzi, F.: CUDD: CU decision diagram package, 3.0. Tech. rep., University of Colorado at Boulder (2015)
53. Van Dijk, T., Van Abbema, F., Tomov, N.: Knor: reactive synthesis using oink. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 103–122. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_7
54. QBF Gallery 2014: QCIR-G14: A non-prenex non-CNF format for quantified Boolean formulas (04 2014)

# Symbolic Model Checking in External Memory

Steffan Christ Sølvsten and Jaco van de Pol

Aarhus University, Denmark {soelvsten,jaco}@cs.au.dk



**Abstract.** We extend the external memory BDD package Adiar with support for monotone variable substitution. Doing so, it now supports the relational product operation at the heart of symbolic model checking. We also identify additional avenues for merging variable substitution fully and the conjunction operation partially inside the relational product's existential quantification step. For smaller BDDs, these additional ideas improve the running of Adiar for model checking tasks up to 47%. For larger instances, the computation time is mostly unaffected as it is dominated by the existential quantification.
Adiar's relational product is about one order of magnitude slower than conventional depth-first BDD implementations. Yet, its I/O-efficiency allows its running time to be virtually independent of the amount of internal memory. This allows it to compute on BDDs with much less internal memory and potentially to solve model checking tasks beyond the reach of conventional implementations.
Compared to the only other external memory BDD package, CAL, Adiar is several orders of magnitude faster when computing on larger instances.

**Keywords:** Time-forward Processing · External Memory Algorithms · Binary Decision Diagrams · Symbolic Model Checking

## 1 Introduction

Binary Decision Diagrams [13] (BDDs) are a concise and canonical representation of $n$-ary Boolean functions as directed acyclic graphs. Starting with [11, 17, 21], BDDs have become a popular tool for symbolic model checking. To this day, they are still used for model checking probabilistic [25, 27, 40] and multi-agent systems [25, 26, 44, 60] and CTL* formulas [5]. Furthermore, they are also still used for verifying network configurations [3, 4, 12, 45], circuits [30–32], and feature models [22, 24]. They also have found recent use for the generation of extended resolution proofs for SAT and QBF problems [14–16]. Furthermore, recent research efforts have made progress on fundamental BDD-based procedures, e.g. [29, 41], and the very implementation of BDDs, e.g. [10, 28, 54]

The conciseness of BDDs mitigate the state space explosion problem of model checking. Yet, there is an inherent lower bound to how much space (or computation time) is needed to meaningfully encapsulate the state space. Hence, the size of the BDDs grows together with the size and the complexity of the

model under verification. Inevitably, BDDs must outgrow the machine's RAM for some models. Yet, most implementations use recursion and hash tables for memoisation [10, 23, 28, 43, 59]. Both recursion and the hash tables introduce cache misses [34, 48]. As the BDDs outgrow the RAM, these cache misses turn into memory swaps which slows computation down by several orders of magnitude [54]. This puts an upper limit on what BDDs can solve in practice.

Unlike conventional BDD implementations, Adiar [54] is designed to handle BDDs that are too large for the RAM. To this end, it replaces the conventional approach of memoised recursion with time-forward processing [6, 20] algorithms. Unlike depth-first recursion [7], this technique is efficient in the I/O-model [1] of Aggarwal and Vitter. This I/O-efficiency, in turn, allows Adiar in practice to process large BDDs without being affected by the disk's speed. Using this technique is only at the cost of a small overhead to its running time [54].

### 1.1 Contributions

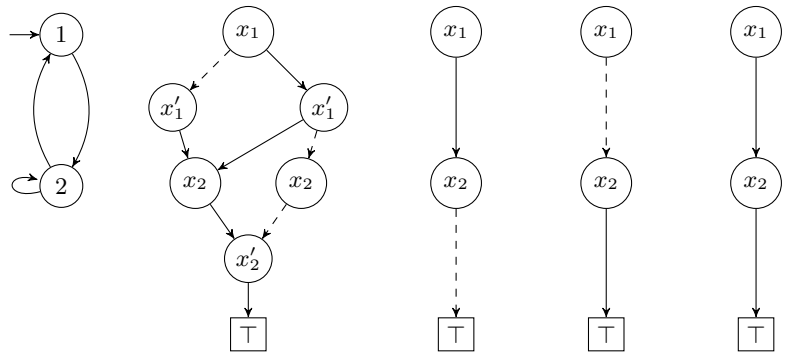
In [57], we extended the external memory BDD package Adiar with efficient multi-variable quantification, based on the notion of nested sweeps. The framework in [57] was evaluated on quantifiers that occur in Quantified Boolean Formulas. In this paper, we evaluate the effectiveness of nested sweeps in the context of symbolic model checking. This requires some extensions to Adiar in order to improve the *relational product*. This operation is needed for the computation of symbolic successors and predecessors of a set of states. We show in Section 3.1 how monotone variable substitutions can be piggy-backed "for free" onto Adiar's other algorithms. Furthermore, we show in Section 3.2 how to combine the conjunction and quantification operations to further improve the running time of the relational product. Finally, Section 5 identifies future work and provides recommendations based on our experimental results in Section 4.

## 2 Preliminaries

### 2.1 I/O Model

To make algorithmic analysis tractable, Aggarwal and Vitter's I/O-model [1] is an abstraction of the machine's complex memory hierarchy. This model consists of two levels of memory: the *internal memory*, e.g. the RAM, and the *external memory*, e.g. the disk. To compute on some data, it has to reside in internal memory. Yet, whereas the external memory is unlimited, the internal memory has only space for $M$ elements. Hence, if the input of size $N$ or some auxiliary data structure exceeds $M$ then data has to be offloaded to external memory. Yet, each data transfer between the two levels, i.e. each read and write, consists of $B$ sized *blocks* of consecutive data. Intuitively, an algorithm is I/O-efficient if it makes sure to use a substantial portion of the $B$ elements in each block that has been read.

An algorithm's I/O-complexity is the number of data transfers, I/Os, it uses. For example, reading an input sequentially requires $N/B$ I/Os whereas random




(a) TS (b) BDD (c) BDD (d) BDD (e) BDD

Fig. 1: The transition system in 1a can be represented as the BDD for its relation relation 1b and the BDD for its initial state in 1c via a *unary* encoding and an *interleaved* variable ordering. One (More) relational product of these two BDDs creates the one in 1d (1e).
For readability, we suppress the $\bot$ terminal. The $\top$ terminal is drawn as a box while BDD nodes are drawn as circles surrounding their decision variable. The *then*, resp. *else*, child is drawn solid, resp. dashed.

access costs up to $N$ I/Os. Furthermore, one can sort $N > M$ elements in $\Theta(\text{sort}(N)) \triangleq \Theta(N/B \cdot \log_{M/B}(N/B))$ I/Os [1]. For most realistic values of $N$, $M$, and $B$, $N/B < \text{sort}(N) \ll N$.

## 2.2 Binary Decision Diagrams

Binary Decision Diagrams [13] (BDDs), based on [2, 42], represent Boolean formulæ as singly-rooted binary directed acyclic graphs. These consist of two sink nodes (*terminals*) with the Boolean values $\top$ and $\bot$. Each non-sink node (*BDD node*) contains a decision variable, $x_i$, together with two children, $v_\top$ and $v_\bot$. Together, these three values represent the if-then-else decision $x_i \ ? \ v_\top \ : \ v_\bot$. Hence, the BDD represents an $n$-ary Boolean formula. In particular, each path from the root to the $\top$ terminal represents one (or more) assignments for which the function outputs $\top$. For example, Fig. 1c represents the formula $x_1 \wedge \neg x_2$.

What are colloquially called BDDs are in particular *ordered* and *reduced* BDDs. A BDD is ordered, if the decision variables occur only once on each path and always according to the same ordering [13]. An ordered BDD is furthermore reduced if it neither contains duplicate subgraphs nor any BDD nodes have their two children being the same [13]. Assuming the variable ordering is fixed, reduced and ordered BDDs are a canonical representation of a Boolean formulæ [13].

**Relational Product** In the case of symbolic model checking, one or more BDDs, $R_{\vec{x},\vec{x}'}$, represent relations between unprimed Boolean variables, $\vec{x}$, that

encode the *current* and primed variables, $\vec{x}'$, that encode the *next* state. These relations are applied to sets of states, $S_{\vec{x}}$, which is identified using only the unprimed variables. For example, the transition system in Fig. 1a, can be represented via a unary encoding with the two variables $x_1$ and $x_2$ which represent the states 1 and 2, respectively. The transitions would then be the relation in Fig. 1b. In particular, Fig. 1b represents each of the three transitions in Fig. 1a as the disjunction of the following three formulas.

$$x_1 \wedge \neg x_1' \wedge \neg x_2 \wedge x_2' \qquad \neg x_1 \wedge x_1' \wedge x_2 \wedge \neg x_2' \qquad \neg x_1 \wedge \neg x_1' \wedge x_2 \wedge x_2'$$

As Fig. 1b is a disjunction of all three transitions, it is a joint [18] relation[1]. Instead, one could also keep Fig. 1b as three separate BDDs for a disjoint [18] relation. This has the benefit of representing the transition system symbolically via smaller BDDs at the cost of having to apply the transitions one-by-one [18]. The initial state of the transition system would be the BDD in Fig. 1c, i.e. the formula $x_1 \wedge \neg x_2$.

These BDDs are manipulated using the following two operations (the *relational product*) to obtain the next or the previous set of states.

$$\mathit{Next}(S_{\vec{x}}, R_{\vec{x},\vec{x}'}) \triangleq (\exists \vec{x} : S_{\vec{x}} \wedge R_{\vec{x},\vec{x}'})[\vec{x}'/\vec{x}] \tag{1}$$

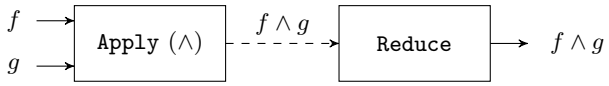
$$\mathit{Prev}(S_{\vec{x}}, R_{\vec{x},\vec{x}'}) \triangleq \exists \vec{x}' : S_{\vec{x}}[\vec{x}/\vec{x}'] \wedge R_{\vec{x},\vec{x}'} \tag{2}$$

For example, *Next* of Fig. 1c and Fig. 1b is the BDD shown in Fig. 1d. The transitive closure of applying *Next*, i.e. the result of $\mathit{Next}^*$ of Fig. 1c and Fig. 1b, is shown in Fig. 1e.

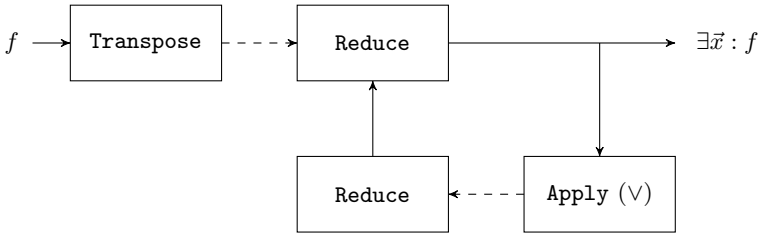
### 2.3 I/O-efficient BDD Manipulation

The Adiar [54] BDD package (based on [7]) aims to compute efficiently on BDDs that are so large they have to be stored on the disk. To do so, rather than using depth-first recursion, it processes the BDDs level by level with time-forward processing [6, 20]. This makes it optimal in the I/O-model [7]. As shown in Fig. 2a, the basic BDD operations, such as the `And` ($\wedge$), are computed in two phases. First, the resulting, but not necessarily reduced, BDD is computed in a top-down `Apply` sweep [54]. Secondly, this BDD is made canonical in a bottom-up `Reduce` sweep [54]. As shown in Fig. 2b, more complex BDD operations, such as `Exists` ($\exists$), are computed by accumulating the result of multiple nested `Apply`–`Reduce` sweeps bottom-up in an outer `Reduce` sweep [57]. Here, each set of nested `Apply`–`Reduce` sweeps computes the `Or` ($\vee$) of a to-be quantified variable's cofactors [57].

[1] This entire example only pertains to a unary encoding of a transition system with asynchronous semantics. If the transition system is synchronous, then the conjunction would be used instead and one has to join all transitions together into a single relation.




(a) The `And` operation.




(b) The `Exists` operation.

Fig. 2: The `Apply`–`Reduce` pipelines in Adiar.

## 3 I/O-efficient Relational Product

### 3.1 I/O-efficient Variable Substitution

The work in [54, 57] covers all BDD operations in Eqs. (1) and (2) but the variable substitution ($[\vec{x}'/\vec{x}]$) between primed and unprimed variables. Hence, this operation's design is the last step towards an I/O-efficient relational product.

In general, the substitution algorithm is equivalent to changing the variable ordering. Yet, this operation is notorious for being hard to compute since it greatly affects the structure and the size of the BDD. But, in the context of symbolic model checking, it merely suffices to consider variable substitutions, $\pi$, that are *monotone* with respect to the variable ordering. That is, if a variable $x_i$ is prior to $x_j$ in the given BDD then $\pi(x_i)$ is also prior to $\pi(x_j)$ in the substituted BDD. For example, the BDDs in Fig. 1 use a variable ordering where primed and unprimed variables are interleaved and so the variable substitutions in Eqs. (1) and (2) are monotone. This monotonicity is useful, since the BDDs before and after such a substitution are isomorphic.

**Proposition 1.** *A monotone variable substitution can be applied to a BDD of $N$ nodes in $\mathcal{O}(N)$ time and using $2 \cdot \frac{N}{B}$ I/Os.*

*Proof.* Using $\frac{N}{B}$ I/Os, one can stream through the $N$ BDD nodes in external memory and apply $\pi$ onto all BDD nodes requiring $\mathcal{O}(N)$ time. The result is simultaneously streamed back to external memory using another $\frac{N}{B}$ I/Os. □

This is optimal if the input is reduced and the output needs to exist separately in external memory. Yet, if the input still needs to be reduced, then substitution can be integrated into the `Reduce` algorithm from [54] as follows: when reducing the level for variable $x_i$, output its new nodes with variable $\pi(x_i)$.

**Proposition 2.** *A monotone variable substitution can be applied during the* `Reduce` *sweep to an unreduced BDD with $n$ levels in $\mathcal{O}(n)$ time, using no additional space in internal memory, and not using any additional I/Os.*

Particular to Eq. (1), variable substitution can be integrated into the quantification operation's outer `Reduce` sweep which preceedes it.

Similarly, substitution in Eq. (2) can become part of the conjunction operation that succeeds it. Here, each BDD node of $S_{\vec{x}}$ would be mapped on-the-fly as they are streamed from external memory. The implementation of such an idea can be greatly simplified with one additional restriction on $\pi$. Note, in the context of model checking, variable substitutions are not only monotone but also *affine*, i.e. $\pi(x_i) = x_{\alpha \cdot i + \beta}$ for some $\alpha, \beta \in \mathbb{N}$ and $\alpha \geq 1$. Hence, by storing $\alpha$ and $\beta$ in internal memory, one can defer applying $\pi$ until they are read as part of the succeeding BDD operation. This makes variable substitution a constant-time operation by adding an overhead to all other BDD operations.

**Proposition 3.** *A monotone and affine variable substitution can be applied to a BDD in $\mathcal{O}(1)$ time, using $\mathcal{O}(1)$ additional space in internal memory, and using no additional I/Os.*

In practice, $\alpha$ is always 1. Hence, only $\beta$ needs to be stored.

### 3.2 An I/O-efficient `AndExists`

Just as Propositions 2 and 3 move the variable substitution inside another operation, the relational product's performance can be further improved by merging the conjunction and existential quantification into a single `AndExists` [62].

The quantification operation's outer `Reduce` sweep requires the input to be transposed [57]. Hence, Fig. 2b includes an initial transposition step. The `Apply` step of the `And` in Fig. 2a produces an unreduced output which already is transposed [54]. This means that naively combining Figs. 2a and 2b into an `AndExists` will result in some computational steps having no effect: the reduced BDD from the `And` is immediately transposed and reduced once more as part of the `Exists`. Hence, the `Reduce` step of the `And` and the `Transpose` step of the `Exists` can be skipped.

**Optimisation 1.** *Use the unreduced result of the conjunction operation as the input to the quantification operation's outer* `Reduce` *sweep.*

This makes the quantification operation's outer `Reduce` sweep also do double duty as the conjunction operation's `Reduce` sweep. This saves the linearithmic time and I/Os otherwise needed to first reduce the result of the conjunction and to then transpose it.

Furthermore, experiments in [57] indicate, the quantification algorithm can become up to $\sim 21\%$ faster by pruning a BDD node (and possibly its subtrees) where quantification is trivial because one of or both its children are terminals.

**Optimisation 2.** *During the conjunction's* `Apply` *sweep, prune the resulting BDD nodes that have to-be quantified variables.*

## 4 Experimental Evaluation

We have added the `bdd_replace` function to Adiar to provide support for variable substitution. For now, it only supports monotone substitutions with the three propositions presented in Section 3.1. Building on this, we added the `bdd_relprod`, `bdd_relnext`, and `bdd_relprev` operations for model checking applications. This includes the ideas in Section 3.2. Optimisation 1 was added in its general form to both the `bdd_exists` and the `bdd_forall` functions: transposition is skipped if the input BDD is unreduced.

With this in hand, we have conducted multiple experiments to evaluate the impact and utility of this work. In particular, we have sought to answer the following research questions:

1. What is the impact of the proposed optimisations in Section 3?
2. How does Adiar's Relational Product operation compare to conventional depth-first implementations?
3. How does Adiar's Relational Product operation compare to the breadth-first algorithms of CAL?

### 4.1 Benchmarks

We have extended our BDD benchmarking suite[2] [54] with the foundations for a symbolic model checker for Petri Nets [49] and Asynchronous Boolean Networks [33,61]. This benchmark explores the given model symbolically as follows:

- *Reachability*:
  The set of reachable states, $S_{\text{reach}}$, is computed via the transitive closure $Next^*(S_I, R)$ on the initial state, $S_I$, and transition relation $R$. This uses `bdd_relnext` up to a polynomial number of times with respect to the model and its state space.
- *Deadlock*:
  The set of deadlocked states are identified via $S_{\text{reach}} \setminus Prev(S_{\text{reach}}, R)$. This requires a single use of `bdd_relprev` if a joint partitioning [18] is used. If a disjoint partitioning [18] is used then this operation is called once for each transition in the model.

Based on [46], the variable ordering is predetermined by analysing the given model[3] with Sloan's algorithm [51]. For each model, we have run them with both a joint and a disjoint [18] partitioning of the transition relation.

Based on preliminary experiments, we identified 75 model instances that were solvable with Adiar. In particular, these are 16 Petri nets from the 2021–2023

[2] github.com/SSoelvsten/bdd-benchmark

[3] Even though [46] suggests one runs the algorithm on a bipartite read/write graph derived from the model, we instead run it on an incidence graph derived from the model. Our preliminary experiments indicate this further decreases the size of the BDDs.

Model Checking Competitions [36–38], 41 Boolean networks distributed with AEON [9], and 18 Boolean networks distributed with PyBoolNet [35].

For all 150 of these instances, none of the BDDs generated during either benchmark grew larger than $2 \cdot 10^6$ BDD nodes (48 MiB). Furthermore, the BDDs encoding the respective initial states required only 294 or fewer BDD nodes (6.9 KiB). While computing the transition relation's transitive closure, the BDDs grew slowly. That is, most, if not all, BDD computations in either benchmark are too small to be within the current scope of Adiar [56]. Hence, inspired by the recent work of Pastva and Henzinger [48], we have also created the following two additional benchmarks.

- *Next*:
  Given a BDD with a set of states, $S_{\vec{x}}$, and another one with a relation, $R_{\vec{x},\vec{x}'}$, the next set of states, $\mathit{Next}(S_{\vec{x}}, R_{\vec{x},\vec{x}})$, is computed with a single `bdd_relnext`.
- *Prev*:
  Given a BDD with a set of states, $S_{\vec{x}}$, and another one with a relation, $R_{\vec{x},\vec{x}'}$, the previous set of states, $\mathit{Prev}(S_{\vec{x}}, R_{\vec{x},\vec{x}})$, is computed with a `bdd_relprev`.

For inputs, we have followed the approach in [48]. The above-described reachability analysis has been extended to save the (joint) transition relation BDD, $R_{\vec{x},\vec{x}'}$, together with the first state BDD, $S_{\vec{x}}$, constructed of each order of magnitude. Using this, we have generated BDDs by running reachability analysis on all models from the 2020–2023 Model Checking Competitions [36–39]. This was done using LibBDD [9] as the BDD backend and a time limit of 1 h on a Ubuntu 24.4 machine with a 12-core 3.6 GhZ Intel i7-12700 processor and 64 GiB of memory. This has resulted in serialised BDDs from 124 model instances. These BDDs have been made publically available at the following DOI:

https://doi.org/10.5281/zenodo.13928216

For our evaluation, we focus on the three models `GPUForwardProgress` 20a (2021), `SmartHome` 16 (2020), and `ShieldPPPs` 10a (2020) where we could generate a set of states with a magnitude of $2^{25}$ BDD nodes[4]. The relation sizes are 500 MiB, 270 MiB, and 0.1 MiB, respectively. For state size, we focus on the four largest orders of magnitude constructed, i.e. from $2^{22}$ (40 MiB) to $2^{25}$ BDD nodes (320 MiB).

### 4.2 Hardware, Settings, and Measurements

Similar to [52,54–57], we ran our experiments on machines at the Centre for Scientific Computing in Aarhus. These machines run Rocky Linux (Kernel 4.18.0-513) with 48-core 3.0 GHz Intel Xeon Gold 6248R processors, 384 GiB of RAM,

[4] A fourth model, `SmartHome` 17 (2020), also created a set of states this large. Yet, the serialized BDD for the relation turned out to be corrupted. The published set of BDDs above has fixed this and other data corruptions. Furthermore, the repository also includes large BDDs that required more than 1 h to be generated.

3.5 TiB of SSD disk (with 48 GiB of swap memory). The benchmark and BDD packages were compiled with GCC 10.1.0 and Rust 1.72.1. Due to an update to the cluster's machines, LibBDD [9] was compiled with GCC 13.2.0 and Rust 1.77.1 for the *Next* and *Prev* benchmarks.

Each BDD package was given 9/10th of these 384 GiB of internal memory[5]; this leaves the remaining space for the OS and the benchmark's other data structures. Otherwise, each BDD package was given a single CPU core and initialised with their respective default and/or recommended settings.

For *Reachability* and *Deadlock* (Table 2 and Fig. 5), we have measured the running time 3 times with a timeout of 48 h. For *Next*, and *Prev* (Table 2 and Fig. 6) we measured the running time of each instance 5 times and bounded the running time to 12 h. For RQ 1, we tried to minimise the noise due to hardware and the OS in the reported numbers. Similar to [54, 56, 57] (based on [19]), we intended to do so by reporting the smallest measured running time. But, some measurements seem to have been taken while the machines were in a particularly good state. Using the minimum in this case makes RQ 1 much harder to investigate. Hence, we instead resort to reporting the median.

For the data shown in Fig. 7 for RQ 2, we deemed that a single measurement of each data point would suffice.

### 4.3 RQ 1: Effect of the Optimisations

We have implemented the ideas from Section 3.1 and Section 3.2 in order of their complexity and expected benefit in practice: Proposition 1 (—), Optimisation 1 (◇), Optimisation 2 (◈), Proposition 2 (⧫), and finally Proposition 3 (♦). For evaluation, we have measured the running time of these five accumulated set of features. Figures 3 and 4 show the relative performance increase compared to only using Proposition 1.

Table 1 shows for each benchmark the total running time of each optimisation. The running time of the two model checking tasks, *Reachability* and *Deadlock*, are mainly affected by the optimisations. On the other hand, the running time of *Next* and *Prev* are comparatively unaffected. This suggest that, as the size of BDDs increases, the computational cost of variable substitution and

[5] CUDD and LibBDD ignore any given memory limit and hence may have used more.

Table 1: Total running time (seconds) of each version of Adiar. The # column indicates the number of instances that were solved by all five versions.

| | # | — Prop. 1 | ◇ Prop. 1 Opt. 1 | ◈ Prop. 1 Opt. 1+2 | ⧫ Prop. 1+2 Opt. 1+2 | ♦ Prop. 1+2+3 Opt. 1+2 |
|---|---|---|---|---|---|---|
| *Reachability* | 147 | 4134.2 | 3918.1 | 3738.6 | 3627.5 | 3659.7 |
| *Deadlock* | 147 | 416.4 | 269.3 | 246.5 | 248.1 | 223.8 |
| *Next* | 12 | 24921.8 | 24378.0 | 23994.6 | 23257.7 | 23628.1 |
| *Prev* | 10 | 77541.9 | 78068.2 | 76862.0 | 75693.8 | 76353.8 |

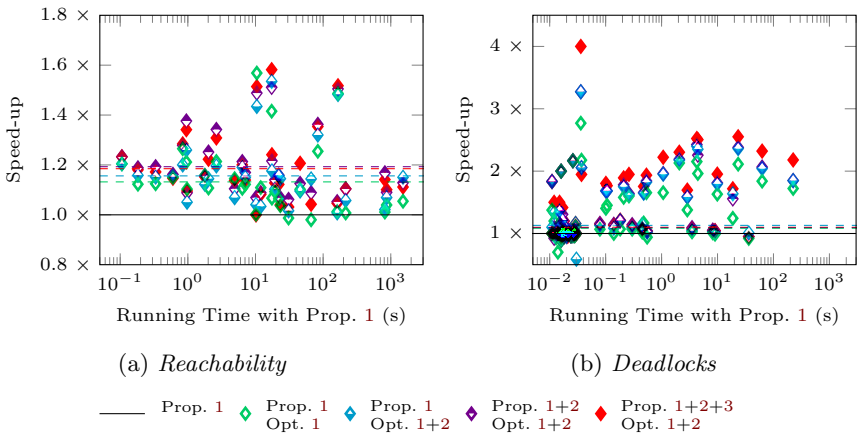
the conjunction, which the improvements pertain to, decreases in comparison to the cost of quantifying variables.

Optimisation 1 (◇) improves the total running time between −0.7% and 35.3% depending on the benchmark. In particular, it mainly improves the running for the smaller BDDs in *Reachability* and *Deadlock*.

Both the relation, $R_{\vec{x},\vec{x}'}$, and states, $S_{\vec{x}}$, have many edges to $\bot$ that shortcut the $\wedge$ operator and gives Optimisation 2 (◈) ample opportunity to prune subtrees. Hence, it improves the total running time between 4.6% and 15.7% depending on the benchmark. In particular, for the *Next* benchmark with `ShieldPPPs` of a magnitude $2^{24}$, this optimisation decreases the intermediate BDD size by 30%. Yet, the number of nodes processed during the quantification operation's nested inner `Apply`–`Reduce` sweeps is unaffected. Hence, this optimisation may only improve performance of the conjunction's `Apply` and `Reduce` sweeps (the latter of which is merged with the quantification operation's outer `Reduce` sweep due to Optimisation 1 (◇)) rather than save work during existential quantification.

Proposition 2 (◈) further improves the total running time for *Reachability* and *Next* by 3.0% and 3.1%, respectively.

Finally, Proposition 3 (◆) has no effect on the *Reachability* and the *Next* benchmarks. Table 1 shows it adds an overhead of up to 1.6%. This slowdown is due to mapping all BDD nodes on-the-fly with an $\alpha = 1$ and $\beta = 0$, i.e. without any actual changes. This overhead can be circumvented by only incorporating this remapping into the $\wedge$-operation in *Prev*. Yet, doing so would require a substantially higher implementation effort and it would also remove the ability to use this optimisation elsewhere. On the other hand, Table 1 also shows Proposition 3 (◆) improves *Deadlock* by 9.8% when compared to the version only including up to Proposition 2 (◈). For *Prev*, there is a slowdown of 0.8%. Yet, since Proposition 2 (◈) does not apply to this benchmark and the version in-




(a) *Reachability* (b) *Deadlocks*

Fig. 3: Impact of optimisations on model checking tasks running time. Averages are drawn as dashed lines.

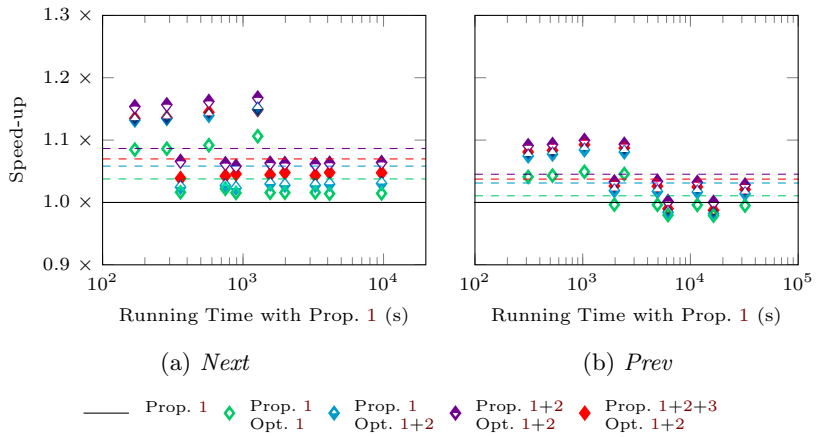



Fig. 4: Impact of optimisations on running time of a single relational product. Averages are drawn as dashed lines.

cluding Proposition 3 (♦) is 0.7% faster than the one only with Optimisations 1 and 2 (♦), this can be attributed to noise and the whims of the compiler.

### 4.4 RQ 2: Comparison to Depth-first Implementations

**Unlimited Memory** We have measured the running time for the same benchmarks with the depth-first BDD packages BuDDy 2.4 [43], CUDD 3.0 [59], LibBDD 0.5.22 [9], and Sylvan 1.8.1 [23]. Sylvan is not included for the *Next* and *Prev* measurements since the manual BDD reconstruction results in a stack overflow[6]. Figures 5 and 6 show scatter plots of the running time of Adiar (with all features from Section 3) in comparison to the other implementations.

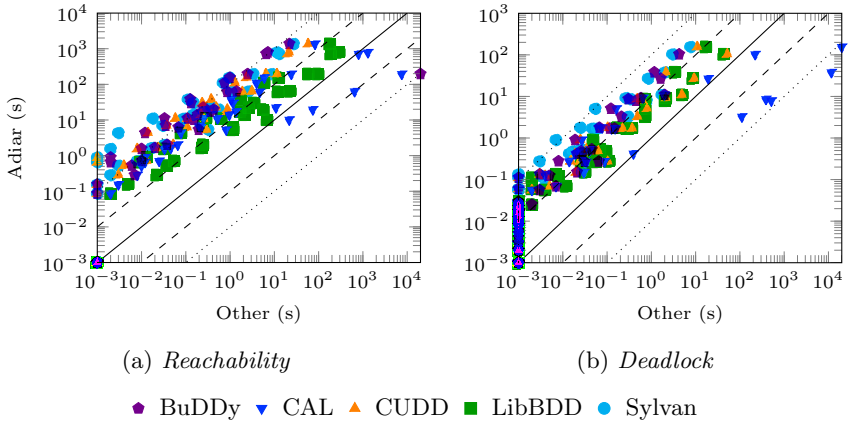
That the BDD size in *Reachability* and *Deadlock* are small is clearly evident in Table 2 and Fig. 5. Earlier, the disk-based algorithms of Adiar have proven to be several orders of magnitude slower than the conventional depth-first algorithms when computing on small BDDs [54, 56, 57]. This is also evident in Fig. 5 where the gap between Adiar and depth-first implementations becomes smaller as the computation time, and with it the BDDs, grow larger. Furthermore, the gap in Fig. 5b is smaller than in Fig. 5a since *Deadlock* only includes computation on the larger BDD that represents the entire state space.

As the BDD sizes are larger in *Next* and *Prev*, the gap between Adiar and depth-first implementations is expected to be smaller. Indeed, as Table 2 and Fig. 6 show, the gap is only of a single order of magnitude or less.

[6] More precisely, its garbage collection algorithm starts out by creating a task for each BDD root. This is to mark all nodes that are still alive. Yet, if the input BDD is big enough then the number of BDD nodes at a single level may outgrow the worker queues in Lace [63].

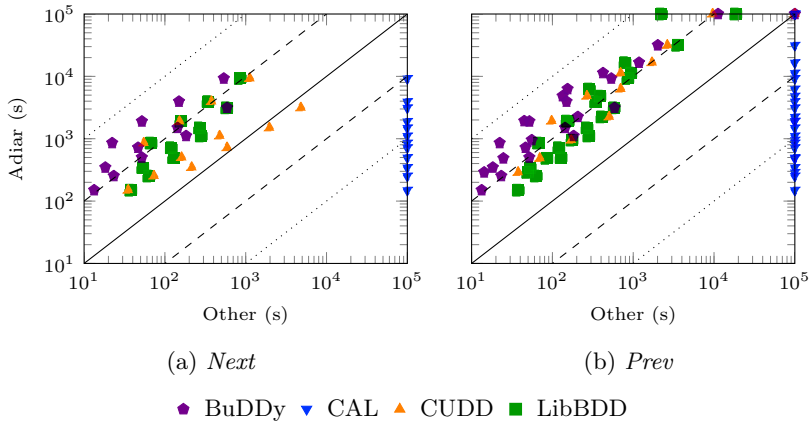
Table 2: Total Running time of Adiar (with Prop. 1, 2, and 3 and Opt. 1 and 2) and other implementations of BDDs to solve all benchmarks. The # column indicates the number of instances that were solved by all BDD packages.

| | # | Adiar | BuDDy | CAL | CUDD | LibBDD | Sylvan |
|---|---|---|---|---|---|---|---|
| *Reachability* | 149 | 3659.7 | 44.2 | 2989.7 | 118.4 | 797.9 | 62.4 |
| *Deadlock* | 149 | 223.8 | 11.3 | 12935.69 | 73.6 | 67.9 | 9.6 |
| *Next* | 12 | 23628.1 | 1827.43 | TO | 10021.34 | 2958.89 | – |
| *Prev* | 10 | 76353.8 | 4252.37 | TO | 6890.05 | 6815.761 | – |




(a) *Reachability* (b) *Deadlock*

Fig. 5: Running time of Adiar on model checking tasks compared to other implementations. Timeouts are shown as markers at the top and the right.




(a) *Next* (b) *Prev*

Fig. 6: Running time of Adiar on a single relational product compared to other implementations. Timeouts are shown as markers at the top and the right.

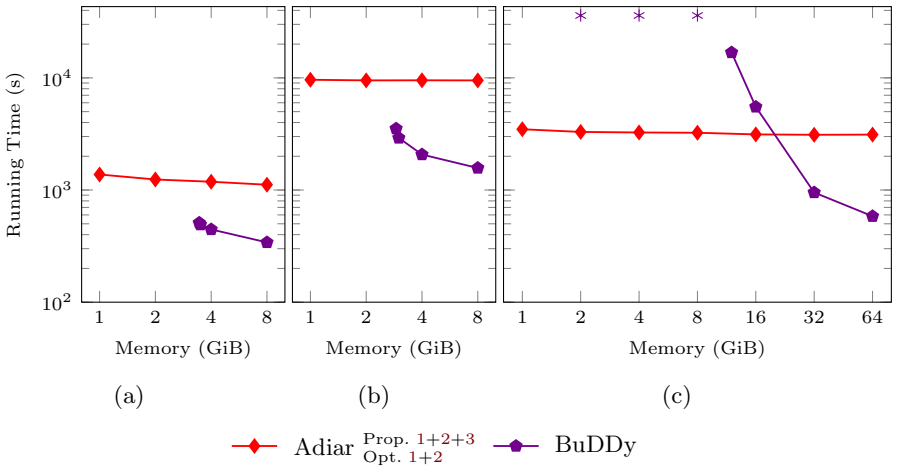



Fig. 7: Running time for *Next* depending on available internal memory. Timeouts are marked as stars. The models are `GPUForwardProgress` 20a (7a), `SmartHome` 16 (7b), and `ShieldPPPs` 10a (7c) with a state space BDD of magnitude $2^{25}$.

**Limited Memory** At time of writing, Adiar still mainly focuses on handling BDDs that are too large to fit into main memory. That is, the results from our experiments shown in Figs. 5 and 6 are heavily biased against Adiar. The benefit of doing so is for the experiments to provide a worst-case comparison between Adiar and other implementations.

To turn the tables, Fig. 7 shows for each of the three models of the *Next* benchmark the running time of Adiar and BuDDy when given different amounts of internal memory. Each of these data points were only measured once. In `GPUForwardProgress` 20a (Fig. 7a) and `SmartHome` 16 (Fig. 7b), BuDDy is unable to complete its computation with less than 3.45 and 2.9 GiB of memory, respectively. As the memory increases beyond this point, its performance quickly increases. For the `ShieldPPPs` 10a model (Fig. 7c), BuDDy needs more than 12 h to finish its computation when given less than 12 GiB of memory. This is due to repeated need to do garbage collection which in turn clears the computation cache and makes it repeat previous computations. Adiar, on the other hand, does not use any memoisation; its priority queues are implicitly doing double-duty as a computation cache [54]. It neither suffers from garbage collection: BDD nodes are stored in files on disk where the entire file is deleted when it is not needed anymore [54]. Yet, Adiar's use of I/O-efficient files and priority queues allows it to only slow down by 23% as its internal memory is decreased far below what BuDDy needs.

### 4.5 RQ 3: Comparison to CAL (Breadth-first Implementation)

Table 2 and Figs. 5 and 6 also include measurements of the breadth-first BDD package, CAL [50]. Similar to our previous results in [57], CAL's running time

is on-par with the depth-first implementations for smaller instances but then quickly deteriorates as the BDDs grow larger. This is due to its use of conventional depth-first recursion for the smaller instances [56].

The overhead of its breadth-first algorithms, on the other hand, makes it much slower than Adiar for the larger instances in Figs. 5a and 5b. This overhead is especially apparent in the *Next* and *Prev*. Here, CAL times out after 12 h for all instances.

## 5 Conclusions and Future Work

In this work, we have succesfully designed an I/O-efficient relational product in Adiar which enables BDD-based symbolic model checking beyond the limits of the machine's memory. These algorithms, as they exist today, are much faster at processing large BDDs than a conventional BDD package that either needs to repeatedly run garbage collection to stay within its memory limits or that needs to offload its BDDs to the disk by means of swap memory. In fact, Adiar's running time is virtually independent of its memory limits.

### Optimisations for Relational Product

Towards designing this I/O-efficient relational product, we have identified multiple optimisations particular to the design of Adiar's algorithms. Based on our results in Section 4, we recommend the following with respect to the optimisations we have proposed in Section 3.

***Recommendation.*** *Propositions 1 and 2 and Optimisations 1 and 2 should be included in an I/O-efficient relational product. A safer alternative to Proposition 3 may be worth the additional implementation effort.*

The gap in running time between Adiar and depth-first implementations is for larger instances about one order of magnitude. This gap is about twice the size than our previous results in [52, 54, 56, 57]: in those benchmarks, Adiar is only up to a factor 4 slower than the conventional approach. In [62], a combined `AndExists` BDD operation roughly halves the running time for depth-first implementations. The optimisations in Section 3.2 aim to also create a combined `AndExists` within the time-forward processing paradigm of Adiar. The gap in performance indicates more ideas are needed.

***Recommendation.*** *Future work on an I/O-efficient relational product should further integrate the* `Exists` *within the* `And`*. To this end, it may be worth investigating alternatives to Optimisation 2; for example, partial quantification in [57] may prove performant in the context of symbolic model checking with BDDs.*

Additional measurements have indicated that for larger instances, the `And` (even without the optimisations in Section 3.2) is responsible for less than 1/10th of the total computation time.

***Recommendation.*** *Future work on an I/O-efficient relational product should especially focus on improving the I/O-efficient* `Exists` *operation's performance.*

### Small-scale BDD Computation

The BDD library CAL [50] (based on [8, 47]) is to the best of our knowledge the only other implementation aiming at efficiently computing on BDDs stored on the disk. In practice, it switches from the conventional depth-first to the breadth-first algorithms described in [50] when the size of the BDDs exceed $2^{19}$ BDD nodes [56]. Yet, these breadth-first algorithms are in practice one or more orders of magnitude slower than Adiar's algorithms. This makes Adiar vastly outperform CAL at solving our benchmarks as the BDDs involved have grown large enough.

But, similar to our previous results in [54, 56, 57], the gap in performance between Adiar and depth-first implementations is still several orders of magnitude for smaller instances. As the BDDs in model checking tasks start out small and only grow slowly, this overhead bars the current version of Adiar from being applicable for model checking in practice.

***Recommendation.*** *Similar to CAL, one should combine Adiar's time-forward processing algorithms with the conventional depth-first approach. Both [56] and [55] have paved the way for getting Adiar's algorithms to work in tandem with a unique node table.*

Yet, doing such a vast engineering task is outside the scope of this paper. We leave the task of combining the strengths of depth-first recursion and time-forward processing as future work.

### Generic Variable Substitution

We have in this work only focused on monotone variable substitution. This still leaves a variable substitution that is I/O-efficient for the general case as an open problem, i.e. an algorithm capable of handling substitutions that change the order of the BDD's levels. If it is possible to design such an algorithm which is efficient with respect to time, space, and I/Os then it can be used as the backbone for novel external memory variable reordering algorithms.

## Acknowledgements

Thanks to the Centre for Scientific Computing, Aarhus, (www.cscaa.dk/) for running our benchmarks on their Grendel cluster. Furthermore, thanks to Nils Husung for previously having added LibBDD to the BDD Benchmarking Suite [53, 54] as part of [28]; this turned out to be vital for creating inputs for the *Next* and *Prev* benchmarks.

## Data Availability Statement

Our experiments are created based on the BDD Benchmarking Suite [53, 54]. The obtained data and its analysis are available in our accompanying artifact [58]. This artifact also includes the inputs, pre-compiled binaries, and scripts to recreate our experiments. Finally, it also provides the source code to read, change, and recompile said binaries.

## References

1. Aggarwal, A., Vitter, J.S.: The Input/Output complexity of sorting and related problems. Communications of the ACM **31**(9), 1116–1127 (1988). https://doi.org/10.1145/48529.48535
2. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers **C-27**(6), 509–516 (1978). https://doi.org/10.1109/TC.1978.1675141
3. Al-Shaer, E., Al-Haj, S.: Flowchecker: configuration analysis and verification of federated openflow infrastructures. In: Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration. pp. 37–44. Association for Computing Machinery (2010). https://doi.org/10.1145/1866898.1866905
4. Al-Shaer, E., Marrero, W., El-Atawy, A., ElBadawi, K.: Network configuration in a box: towards end-to-end verification of network reachability and security. In: International Conference on Network Protocols. vol. 17, pp. 123–132. IEEE (2009). https://doi.org/10.1109/ICNP.2009.5339690
5. Amparore, E., Donatelli, S., Gallà, F.: A CTL* model checker for Petri nets. In: Application and Theory of Petri Nets and Concurrency. Lecture Notes in Computer Science, vol. 12152, pp. 403–413. Springer (2020). https://doi.org/10.1007/978-3-030-51831-8_21
6. Arge, L.: The buffer tree: A new technique for optimal I/O-algorithms. In: Algorithms and Data Structures. pp. 334–345. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
7. Arge, L.: The I/O-complexity of ordered binary-decision diagram manipulation. In: Proceedings of International Symposium on Algorithms and Computations, ISAAC'95. pp. 82 – 91 (1995)
8. Ashar, P., Cheong, M.: Efficient breadth-first manipulation of binary decision diagrams. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 622–627. IEEE Computer Society Press (1994). https://doi.org/10.1109/ICCAD.1994.629886
9. Beneš, N., Brim, L., Kadlecaj, J., Pastva, S., Šafránek, D.: AEON: Attractor bifurcation analysis of parametrised Boolean networks. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 12224, pp. 569 – 581. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_28
10. Beyer, D., Friedberger, K., Holzner, S.: PJBDD: A BDD library for Java and multithreading. In: Automated Technology for Verification and Analysis. pp. 144 – 149. Springer (2021). https://doi.org/10.1007/978-3-030-88885-5_10
11. Bose, S., Fisher, A.L.: Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In: Proc. of the IMEC-IFIP Intl. Workshop Applied Formal Methods for Correct VLSI Design. pp. 759–767 (1989)

12. Brown, M., Fogel, A., Halperin, D., Heorhiadi, V., Mahajan, R., Millstein, T.: Lessons from the evolution of the batfish configuration analysis tool. In: Proceedings of the ACM SIGCOMM 2023 Conference. p. 122–135. ACM SIGCOMM '23, Association for Computing Machinery (2023). https://doi.org/10.1145/3603269.3604866
13. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677 – 691 (1986)
14. Bryant, R.E., Biere, A., Heule, M.J.H.: Clausal proofs for pseudo-Boolean reasoning. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 443–461. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_25
15. Bryant, R.E., Heule, M.J.H.: Dual proof generation for quantified Boolean formulas with a BDD-based solver. In: Automated Deduction – CADE 28. pp. 433–449. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_25
16. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 12651, pp. 76–93. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_5
17. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: $10^{20}$ states and beyond. Information and Computation **98**(2), 142–170 (1992). https://doi.org/10.1016/0890-5401(92)90017-A
18. Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L.: Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **13**(4), 401–424 (1994). https://doi.org/10.1109/43.275352
19. Chen, J., Revels, J.: Robust benchmarking in noisy environments. arXiv (2016), https://arxiv.org/abs/1608.04295
20. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 139—-149. SODA '95, Society for Industrial and Applied Mathematics (1995)
21. Coudert, O., Berthet, C., Madre, J.C.: Verification of synchronous sequential machines based on symbolic execution. In: Automatic Verification Methods for Finite State Systems. pp. 365–373. Springer Berlin Heidelberg, Berlin, Heidelberg (1990). https://doi.org/10.1007/3-540-52148-8_30
22. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: 11th International Software Product Line Conference (SPLC 2007). pp. 23–34 (2007). https://doi.org/10.1109/SPLINE.2007.24
23. Van Dijk, T., Van de Pol, J.: Sylvan: multi-core framework for decision diagrams. International Journal on Software Tools for Technology Transfer **19**, 675–696 (2016). https://doi.org/10.1007/s10009-016-0433-2
24. Dubslaff, C., Husung, N., Käfer, N.: Configuring bdd compilation techniques for feature models. In: International Systems and Software Product Line Conference. pp. 209–216. SPLC '24, Association for Computing Machinery (2024). https://doi.org/10.1145/3646548.3676538
25. Fu, C., Hahn, E.M., Li, Y., Schewe, S., Sun, M., Turrini, A., Zhang, L.: EPMC gets knowledge in multi-agent systems. In: Verification, Model Checking, and Abstract Interpretation. pp. 93–107. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_5
26. Gammie, P., Van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Computer Aided Verification. pp. 479–483. Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_41

27. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker `storm`. Software Tools for Technology Transfer **24**(4), 589–610 (2022). https://doi.org/10.1007/s10009-021-00633-z
28. Husung, N., Dubslaff, C., Hermanns, H., Köhl, M.A.: OxiDD: A safe, concurrent, modular, and performant decision diagram framework in Rust. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24). Lecture Notes in Computer Science, vol. 14572. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_13
29. Jakobsen, A.B., Jørgensen, R.S.M., Van de Pol, J., Pavlogiannis, A.: Fast symbolic computation of bottom SCCs. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 110–128. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_6
30. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E., Naik, A.: Replacing testing with formal verification in Intel® core™ i7 processor execution engine validation. In: Computer Aided Verification. pp. 414–429. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_32
31. Kaivola, R., Kama, N.B.: Timed causal fanin analysis for symbolic circuit simulation. In: Formal Methods in Computer-Aided Design. pp. 99–107 (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_16
32. Kaivola, R., O'Leary, J.: Verification of arithmetic and datapath circuits with symbolic simulation. In: Chattopadhyay, A. (ed.) Handbook of Computer Architecture, pp. 1–52. Springer (2022). https://doi.org/10.1007/978-981-15-6401-7_37-1
33. Kauffman, S.A.: Metabolic stability and epigenesis in randomly constructed genetic nets. Journal of Theoretical Biology **22**(3), 437–467 (1969). https://doi.org/10.1016/0022-5193(69)90015-0
34. Klarlund, N., Rauhe, T.: BDD algorithms and cache misses. In: BRICS Report Series. vol. 26 (1996). https://doi.org/10.7146/brics.v3i26.20007
35. Klarner, H., Streck, A., Siebert, H.: PyBoolNet: A python package for the generation, analysis and visualization of Boolean networks. Bioinformatics **33**(5), 770–772 (2016). https://doi.org/10.1093/bioinformatics/btw682
36. Kordon, F., Bouvier, P., Garavel, H., Hillah, L.M., F., H.H., Amat, N., Amparore, E., Berthomieu, B., Biswal, S., Donatelli, D., Galla, F., Dal Zilio, S., Jensen, P.G., Jezequel, L., He, C., Le Botlan, D., Li, S., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y., A., W., K., W.: Complete results for the 2021 edition of the model checking contest (2021), http://mcc.lip6.fr/2021/results.php
37. Kordon, F., Bouvier, P., Garavel, H., Hulin-Hubard, F., Amat, N., Amparore, E., Berthomieu, B., Donatelli, D., Dal Zilio, S., Jensen, P.G., Jezequel, L., He, C., Li, S., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y.: Complete results for the 2022 edition of the model checking contest (2022), http://mcc.lip6.fr/2022/results.php
38. Kordon, F., Bouvier, P., Garavel, H., Hulin-Hubard, F., Amat, N., Amparore, E., Berthomieu, B., Donatelli, D., Dal Zilio, S., Jensen, P.G., Jezequel, L., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y.: Complete results for the 2023 edition of the model checking contest (2023), https://mcc.lip6.fr/2023/results.php
39. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amparore, E., Berthomieu, B., Biswal, S., Donatelli, D., Galla, F., Ciardo, G., Dal Zilio, S., Jensen, P.G., Jezequel, L., Le Botlan, D., Li, S., Miner, A., Paviot-Adet, E., Srba, J., Thierry-Mieg, Y.: Complete results for the 2020 edition of the model checking contest (2020), http://mcc.lip6.fr/2020/results.php

40. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Computer Performance Evaluation: Modelling Techniques and Tools. pp. 200–204. Springer (2002). https://doi.org/10.1007/3-540-46029-2_13
41. Larsen, C.A., Schmidt, S.M., Steensgaard, J., Jakobsen, A.B., Van de Pol, J., Pavlogiannis, A.: A truly symbolic linear-time algorithm for SCC decomposition. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 353–371. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_22
42. Lee, C.Y.: Representation of switching circuits by binary-decision programs. The Bell System Technical Journal **38**(4), 985 – 999 (1959). https://doi.org/10.1002/j.1538-7305.1959.tb01585.x
43. Lind-Nielsen, J.: BuDDy: A binary decision diagram package. Tech. rep., Department of Information Technology, Technical University of Denmark (1999)
44. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. Software Tools for Technology Transfer **19**(1), 9–30 (2017). https://doi.org/10.1007/s10009-015-0378-x
45. Lopes, N.P., Bjørner, N., Godefroid, P., Varghese, G.: Network verification in the light of program verification. Tech. rep., Microsoft Research (2013), https://microsoft.com/en-us/research/publication/network-verification-in-the-light-of-program-verification/
46. Meijer, J., Van de Pol, J.: Bandwidth and wavefront reduction for static variable ordering in symbolic model checking. arXiv (2015), https://arxiv.org/abs/1511.08678
47. Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: International Conference on Computer Aided Design (ICCAD). pp. 48–55. IEEE Computer Society Press (1993). https://doi.org/10.1109/ICCAD.1993.580030
48. Pastva, S., Henzinger, T.: Binary decision diagrams on modern hardware. In: Conference on Formal Methods in Computer-Aided Design. pp. 122–131 (2023)
49. Petri, C.A.: Grundsätzliches zur beschreibung diskreter prozesse. In: 3. Colloquium über Automatentheorie, Hannover 1965. pp. 121–140. Birkhäuser-Verlag (1967)
50. Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.: High performance BDD package by exploiting memory hierarchy. In: 33rd Design Automation Conference (DAC). pp. 635–640. Association for Computing Machinery (1996). https://doi.org/10.1145/240518.240638
51. Sloan, S.W.: A FORTRAN program for profile and wavefront reduction. International Journal for Numerical Methods in Engineering **28**(11), 2651–2679 (1989). https://doi.org/10.1002/nme.1620281111
52. Sølvsten, S.C., Van de Pol, J.: Adiar 1.1: Zero-suppressed decision diagrams in external memory. In: NASA Formal Methods Symposium. LNCS 13903, Springer, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-33170-1_28
53. Sølvsten, S.C., Husung, N., Jakobsen, A.B., Van de Pol, J.: Bdd benchmarking suite. Zenodo (04 2021). https://doi.org/10.5281/zenodo.4718224
54. Sølvsten, S.C., Van de Pol, J., Jakobsen, A.B., Thomasen, M.W.B.: Adiar: Binary decision diagrams in external memory. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 13244, pp. 295–313. Springer, Berlin, Heidelberg (2022). https://doi.org/10.1007/978-3-030-99527-0_16
55. Sølvsten, S.C., Rysgaard, C.M., Van de Pol, J.: Random access on narrow decision diagrams in external memory. In: Model Checking Software. Lecture Notes in Computer Science, vol. 14624, pp. 137–145. Springer (2023). https://doi.org/10.1007/978-3-031-66149-5_7

56. Sølvsten, S.C., Van de Pol, J.: Predicting memory demands of BDD operations using maximum graph cuts. In: Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 14216, pp. 72–92. Springer (2023). https://doi.org/10.1007/978-3-031-45332-8_4
57. Sølvsten, S.C., Van de Pol, J.: Multi-variable quantification of BDDs in external memory using nested Sweeping (extended version). arXiv (2024). https://doi.org/10.48550/arXiv.2408.14216
58. Sølvsten, S.C., Van de Pol, J.: Artifact: Symbolic model checking in external memory. Zenodo (02 2025). https://doi.org/10.5281/zenodo.14833492
59. Somenzi, F.: CUDD: CU decision diagram package, 3.0. Tech. rep., University of Colorado at Boulder (2015)
60. Su, K., Sattar, A., Luo, X.: Model checking temporal logics of knowledge via OBDDs. The Computer Journal **50**(4), 403–420 (2007). https://doi.org/10.1093/comjnl/bxm009
61. Thomas, R.: Regulatory networks seen as asynchronous automata: A logical description. Journal of Theoretical Biology **153**(1), 1–23 (1991). https://doi.org/10.1016/S0022-5193(05)80350-9
62. Van Dijk, T., Hahn, E.M., Jansen, D.N., Li, Y., Neele, T., Stoelinga, M., Turrini, A., Zhang, L.: A comparative study of BDD packages for probabilistic symbolic model checking. In: Dependable Software Engineering: Theories, Tools, and Applications. pp. 35–51. Springer (2015). https://doi.org/10.1007/978-3-319-25942-0_3
63. Van Dijk, T., Van de Pol, J.: Lace: Non-blocking split deque for work-stealing. In: Euro-Par 2014: Parallel Processing Workshops. pp. 206–217. Springer (2014). https://doi.org/10.1007/978-3-319-14313-2_18

## A SCC Decomposition

We also implemented the following third foundational model checking operation for our experiments in Section 4.

– *SCC Decomposition*:
The reachable states are decomposed into their strongly connected components (SCCs) via the CHAIN algorithm [41]. This uses both `bdd_relnext` and `bdd_relprev` up to a polynomial number of times with respect to the model and its state space. As per [29], each deadlock state identified during the *Deadlock* stage is an SCC which can be skipped.

The 75 model instances were in particular the ones where Adiar seemed able to consecutively solve *Reachability*, *Deadlock*, and *SCC Decomposition* within 2 days. Likewise, the results in measurements in Table 2 and Fig. 5 were run with a timeout of 48 h.

This benchmark was left out of the above experiments, since they do not add new knowledge. In particular, the BDD size stayed too small to be within Adiar's current scope. If anything, the results shown below further cements the need for incorporating the conventional depth-first algorithms with Adiar's I/O-efficient time-forward processing.

### A.1 RQ 1: Effect of the Optimisations

Table 3 and Fig. 8 shows the effect of each optimisation on this particular benchmark. Similar to *Reachability* and *Deadlock*, the *SCC Decomposition* is positively affected by the optimisations due to the small BDD size.

Optimisation 1 (◇) improves the total running time for *SCC Decomposition* with 20.7%. Optimisation 2 (◆) further improves the running time by 14.0% and Proposition 2 (◆) by 4.8%. Finally, Proposition 3 (◆) improves the total running time by 1.5%

### A.2 RQ 2: Comparison to Depth-first Implementations

Table 4 and Fig. 9 shows the running time of Adiar and the depth-first BDD packages described in Section 4. Here, the gap between Adiar and the depth-first BDD packages clearly show that the size of the BDDs stay small. This is due to the fact that the Chain algorithm in [41] repeatedly explores the (remaining) set of states from single pivot states.

Table 3: Total running time (seconds) of each version of Adiar on SCC Decomposition. The # column indicates the number of instances that were solved by all five versions.

| | # | Prop. 1 | Prop. 1 Opt. 1 | Prop. 1 Opt. 1+2 | Prop. 1+2 Opt. 1+2 | Prop. 1+2+3 Opt. 1+2 |
|---|---|---|---|---|---|---|
| *SCC* | 144 | 734040.5 | 562074.5 | 483553.0 | 460388.5 | 453712.4 |

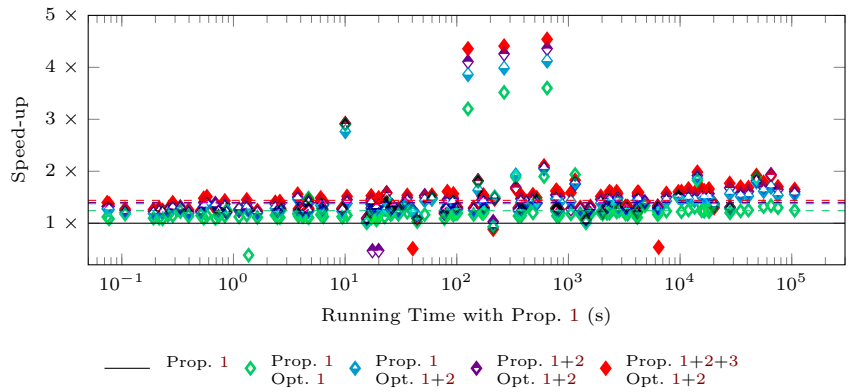



Fig. 8: Impact of optimisations on SCC Decomposition running time. Averages are drawn as dashed lines.

### A.3 RQ 3: Comparison to CAL (Breadth-first Implementation)

Table 4 and Fig. 9 also shows the running time of CAL for solving the SCC decomposition taks. Whereas CAL becomes slower than Adiar for the largest instances in Figs. 5a and 5b, the same is not evident in Fig. 9. This is further testament to the small size of the BDDs in this benchmark.

Table 4: Total Running time of Adiar (with Prop. 1, 2, and 3 and Opt. 1 and 2) and other implementations of BDDs for SCC decomposition. The # column indicates the number of instances that were solved by all BDD packages.

| | # | Adiar | BuDDy | CAL | CUDD | LibBDD | Sylvan |
|---|---|---|---|---|---|---|---|
| *SCC* | 147 | 567188.5 | 680.6 | 22679.93 | 1840.0 | 10201.9 | 862.2 |

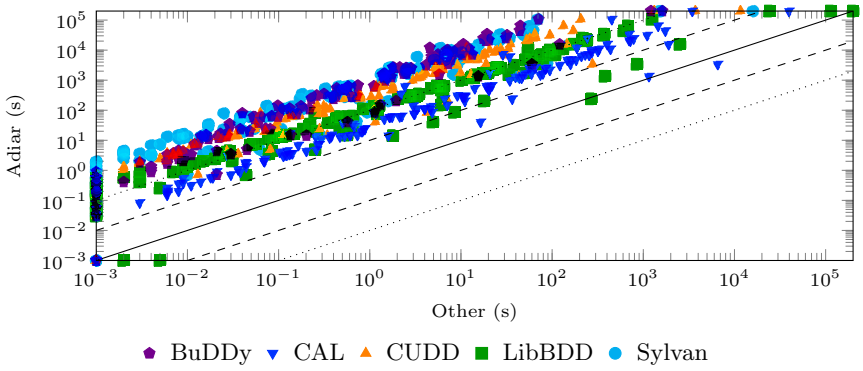



Fig. 9: Running time of Adiar on SCC decomposition compared to other implementations. Timeouts are shown as markers at the top and the right.

# Part III

# Future Work

# Chapter 9

# Manual Variable Reordering

## 9.1 Introduction

All of the paper manuscripts in Part II assume that the Binary Decision Diagrams [38] (BDDs) (based on [4, 117]) use the identity variable order, i.e. $x_1, x_2, \ldots, x_n$. This is done in the interest of simplicity and to ease presentation. Yet, the size of BDDs, and hence their usability, heavily depend on the variable ordering that they use [38].

It can become necessary to reorder the variables during computation to keep the BDDs sufficiently small [21, 49, 77]. Doing so can be left to the BDD package by means of *dynamic* variable reordering, i.e. the BDD package transparently derives and applies a new ordering on-the-fly based on the BDDs themselves. Yet, a *manual* variable reordering, i.e. a new ordering derived by the user and given to the BDD package, suffices in many cases. For example, an analysis of the problem domain and the given instance is often enough to derive a good variable ordering which can be fixed prior to any BDD computations [80, 100, 127, 132, 161]. If the ordering needs to change throughout computation, what the new ordering should be, and sometimes even when it should be applied, can at times be derived from the given instance in the problem domains [20, 67]. Furthermore, BDDs that co-exist during computation may need different variable orderings [21, 77, 136]. For example, when using a disjoint partitioning of the transition relation [48], each partition may induce a different optimal ordering [49, 167]. A BDD may be deserialised from the disk with a different ordering than is intended for the computation at hand [49, 167, 173]. In particular for Adiar, since its I/O-efficiency is at the cost of the BDDs not being shared, i.e. it stores its BDDs in separate files on disk, the addition of variable orderings to Adiar will allow co-existing BDDs to each have their own ordering. Manual reordering is needed to resolve computation on BDDs with incompatible orderings.

Since the other chapters assume the identity variable order, the terms *level* and *label* have been used interchangeably. In this chapter, a clear distinction needs to be made between a BDD node's label, $i$, which identifies its decision variable, $x_i$, and its level, $\ell_i$, which refers to its position in the BDD. Yet, it will become useful to think of a variable ordering as a permutation, $\pi$, from the symmetric group $S_n$ of $n$

elements which is either used (*i*) as a *variable reordering*, i.e. as a change of the order of the levels based on the permutation, or (*ii*) a *variable substitution*, i.e. variables are substituted based on the permutation without changing the variable ordering. For reasons that will become apparent later, we will in this chapter for the most part treat $\pi$ as as a permutation of a BDD's variables rather than its levels.

We refer to Chapters 1 and 7 for details on BDDs, Chapter 7 for details on Adiar's nested sweeping framework, Chapter 4 for Zero-suppressed Decision Diagrams (ZDDs), and to Chapter 8 for our previous work on monotone variable substitution. Furthermore, the notion of *levelised cuts* from Chapter 5 and the *restrict* operation in [38] will be vital for the analysis of our algorithm; we refer to Chapter 3 for a brief description of Bryant's restrict. We reuse notation from Chapters 3 and 7.

### Contributions

In Section 9.2, we analyse the following depth-first BDD operation from BuDDy [119].

`bdd_replace(`$f$`, `$\pi$`)` : Apply a variable substitution, $\pi$.

In Section 9.3 we show how this algorithm can be translated to the time-forward processing [11, 53] paradigm in Adiar. The time and I/O-complexity of the resulting algorithm is $\mathcal{O}(\text{sort}(T \cdot \sum_{i=1}^{n} C^{\emptyset}_{1:f[i]}))$, where $T$ is the size of the final substituted BDD, $n$ is the number of variables, and $C^{\emptyset}_{1:f[i]}$ is the size of the 1-level cut in the input right beneath level $i$, excluding arcs to terminals. Where the original depth-first algorithm uses $\mathcal{O}(N+n\cdot T)$ space, our algorithm uses $\mathcal{O}(N+T\cdot \max_i(C^{\emptyset}_{1:f[i]}))$ space. In practice, $n$ is in almost all cases much smaller than $\max_i(C^{\emptyset}_{1:f[i]})$. Section 9.3 also includes multiple ways to further improve upon the algorithm asymptotically and in practice.

Sections 9.4 and 9.5 provide two alternative and more efficient algorithms for the special case of $\pi$ either consisting only of adjacent variable swaps or it being a single swap of two non-adjacent variables.

In Adiar, these algorithms can be repurposed for manual variable reordering. To this end, each BDD, $f$, carries with it its own variable ordering $\pi_f$. The following two functions are added to Adiar's API:

`bdd_reorder(`$f$`, `$\pi'$`)` : Reorder $f$ into the ordering $\pi'$.

`bdd_reorder(`$f$`, `$g$`)` : Reorder $f$ to match the ordering $\pi_g$ of BDD $g$.

Furthermore, when combining two BDDs $f$ and $g$ they may have two incompatible orderings $\pi_f$ and $\pi_g$. In this case, either an exception is thrown or they are automatically put into a common ordering. For the latter, the ideas in [49, 167] may be relevant.

## 9.2 The `bdd_replace` function in BuDDy

Figure 9.1 shows the pseudo-code for the `bdd_replace` operation provided by the BDD package BuDDy [119]. For clarity, we omit the use of computation caches and other details.

```
1   bdd_replace(f, π):
2     l := bdd_replace(f.low, π)
3     h := bdd_replace(f.high, π)
4     return bdd_correctify(l, h, π(f.var))
5
6   bdd_correctify(f_l, f_h, ℓ)
7     if ℓ < f_l.var ∧ ℓ < f_h.var
8       return f_l = f_h ? f_l : Node { ℓ, f_l, f_h }
9
10    if f_l.var = f_h.var
11      f'_l := bdd_correctify(f_l.low, f_h.low, ℓ)
12      f'_h := bdd_correctify(f_l.high, f_h.high, ℓ)
13    else if f_l.var < f_h.var
14      f'_l := bdd_correctify(f_l.low, f_h, ℓ)
15      f'_h := bdd_correctify(f_l.high, f_h, ℓ)
16    else // f_l.var > f_h.var
17      f'_l := bdd_correctify(f_l, f_h.low, ℓ)
18      f'_h := bdd_correctify(f_l, f_h.high, ℓ)
19    return Node { min(f_l.var, f_h.var), f'_l, f'_h }
```

Figure 9.1: The `bdd_replace` algorithm.

**Proposition 9.1.** *The result of* `bdd_replace(f, π)` *is* $f[\vec{x} \mapsto \pi(\vec{x})]$.

*Proof Sketch.* Correctness on terminals is trivial. The results `l` and `h` on lines 2 and 3 are correct substitutions by induction. Also by induction, since $f$.`var` is not in the support of $f$.`low` nor $f$.`high` then neither is $\pi(f.\texttt{var})$ in the support of `l` and `h`. Hence, the level $\pi(f.\texttt{var})$ is free in both. The nested call to `bdd_correctify` on line 4 is a 2-ary product construction that bubbles the decision on $f$.`var` in the root of $f$ downto the empty level of $\pi(f.\texttt{var})$. □

The following rephrases the algorithm's correctness above as a lemma which will become key for all of our later analysis.

**Lemma 9.2.** *Each recursive output of* `bdd_replace(f, π)` *is a restriction of* `bdd_replace(f, π)` *itself, i.e. of* $f[\vec{x} \mapsto \pi(\vec{x})]$.

*Proof.* Any path from the root of $f$ to some subtree $f'$ reflects an assignment $\vec{b} \in \{\bot, \top\}^*$ to the variables $\vec{y}$ that are a subsequence of $\vec{x}$, i.e. $f' \equiv f[\vec{y} \mapsto \vec{b}]$. Hence, the variables in $\vec{y}$ are neither part of the remaining support of the subfunction $f'$. Let $\vec{x} \setminus \vec{y}$ be the subsequence of the variables in $\vec{x}$ that are not in $\vec{y}$.

Due to Proposition 9.1, the recursion, `bdd_replace(f', π)`, outputs the BDD that represents $f'[\vec{x} \mapsto \pi(\vec{x})]$. That is, $f'[\vec{x} \mapsto \pi(\vec{x})]$ is equivalent to $f[\vec{y} \mapsto \vec{b}][\vec{x} \mapsto \pi(\vec{x})]$. On the other hand, restricting the final BDD, `bdd_replace(f, π)`, based on $\pi(\vec{y})$ results in the subfunction $f[\vec{x} \mapsto \pi(\vec{x})][\pi(\vec{y}) \mapsto \pi(\vec{b})]$.

Since $\pi$ is a permutation, then there exists no $x_i, x_j \in \vec{x}$ such that $x_i \neq x_j$ and $\pi(x_i) = \pi(x_j)$. Hence, a point-wise argument on $\vec{x}$ suffices to show that for each

$x_i \in \vec{x}$, $f[\vec{y} \mapsto \vec{b}][\vec{x} \mapsto \pi(\vec{x})] \equiv f[\vec{x} \mapsto \pi(\vec{x})][\pi(\vec{y}) \mapsto \pi(\vec{b})]$. If $x_i \in \vec{y}$, i.e. $x_i$ is restricted on the path from $f$ to $f'$, then the substitution $x_i[\vec{y} \mapsto \vec{b}][\vec{x} \mapsto \pi(\vec{x})] \equiv b_i$ and similarly the substitution $x_i[\vec{x} \mapsto \pi(\vec{x})][\pi(\vec{y}) \mapsto \pi(\vec{b})]$ is the constant $b_i$ too. Otherwise if $x_i \in \vec{x} \setminus \vec{y}$, $x_i[\vec{y} \mapsto \vec{b}][\vec{x} \mapsto \pi(\vec{x})] \equiv \pi(x_i)$ and $x_i[\vec{x} \mapsto \pi(\vec{x})][\pi(\vec{y}) \mapsto \pi(\vec{b})] \equiv \pi(x_i)$. That is, assigning variables $\vec{y}$ prior to substitution (implicitly done when recursing on lines 2 and 3 in Fig. 9.1) or assigning $\pi(\vec{y})$ afterwards result in the same (sub)formula. □

**Corollary 9.3.** *Each invocation of* `bdd_replace` *outputs $T$ or fewer BDD nodes.*

The implementation in BuDDy has a computation cache for the `bdd_replace` but not its subprocedure, `bdd_correctify`. Fig. 9.2 shows how `bdd_replace` can make `bdd_correctify` resolve the same pair of nodes twice: both calls to `bdd_replace` with arguments $(v_1, \pi)$ and $(v_2, \pi)$ will each spawn a recursive call to `bdd_correctify`$(v_3, v_4, \pi(x_i))$. This can potentially cascade into an exponential blowup in the running time. Let us ignore this detail in BuDDy's implementation and assume that the subprocedure `bdd_correctify` also uses a computation cache.

**Proposition 9.4.** `bdd_replace`$(f, \pi)$ *uses $\mathcal{O}(N + n \cdot T)$ time and space.*

*Proof.* As the algorithm traverses the input of size $N$, the depth-first recursion on lines 2 and 3 in Fig. 9.1 is resolving paths of length up to $n$. Due to Corollary 9.3, both recursion results, `l` and `h`, are BDDs of size $T$ or smaller. The deeper this recursion has gotten, the fewer variables can be present in either `l`'s and `h`'s support. Variables are reintroduced one-by-one by invoking `bdd_correctify` on line 4. If such a variable is moved all the way to the bottom of `l` and `h` then doing so changes the entire subgraph. Hence, even though their size is bounded by $T$, the sub-BDDs themselves are quite possibly not part of the output nor reused in any other computation. Yet, due to memoisation and Lemma 9.2, subtrees that are commonly used at the same level of recursion are not retraversed or constructed anew.

That is, $\mathcal{O}(N)$ space and time is spent on traversing and memoising results from the input. Furthermore, $\mathcal{O}(n \cdot T)$ space and time is needed to construct the output. □

Figure 9.3 shows that the above bound is tight. Consider the BDD $f$ in Fig. 9.3a with $N = n$ BDD nodes. Intuitively, $f$ finds the longest prefix of variables $x_1, x_2, \ldots, x_n$ that is assigned $\top$ and then checks whether the remaining variables are assigned $\bot$. Figure 9.3c is the result of $\pi$ substituting $x_i$ for $x_{n-i+1}$, i.e. where the variable ordering

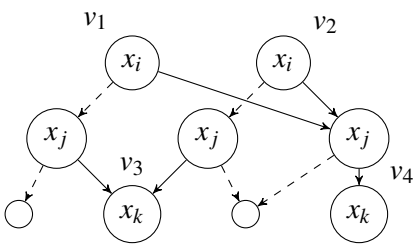
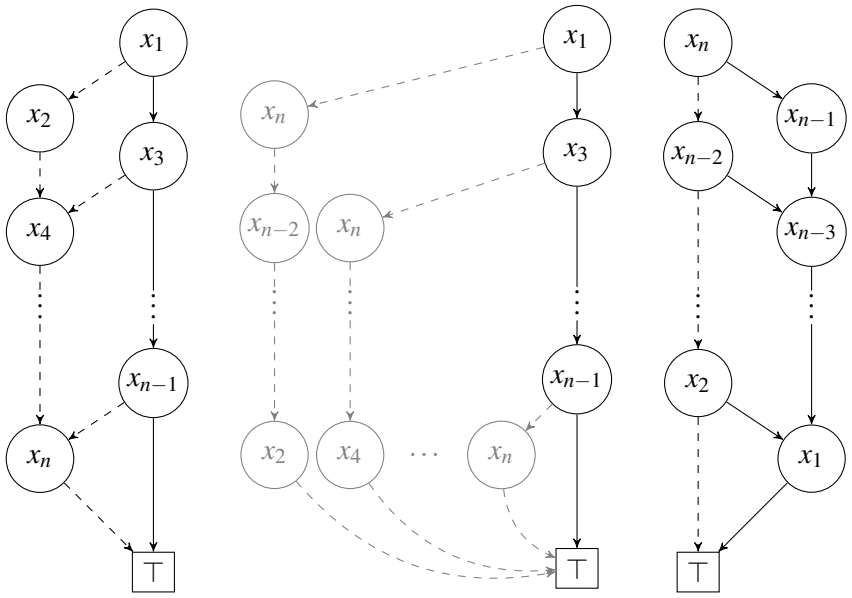



Figure 9.2: Example where memoisation is necessary for `bdd_correctify`.




(a) Input (b) Intermediate State (c) Output

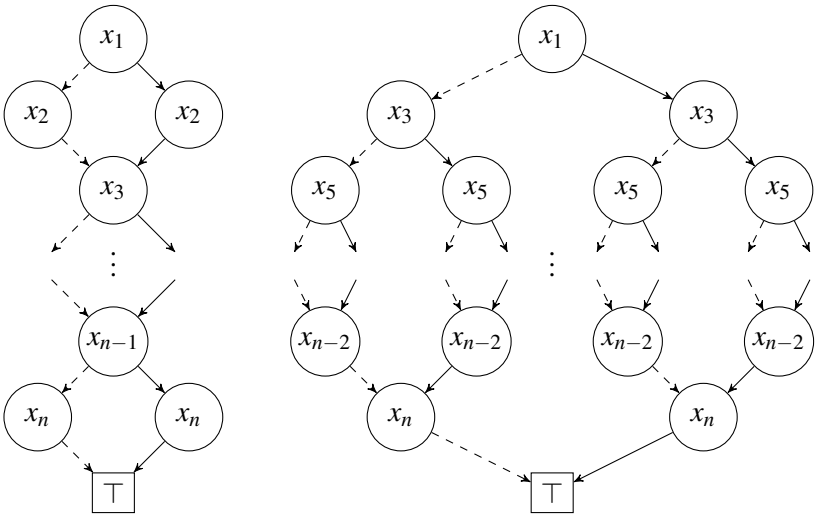
Figure 9.3: Example of `bdd_replace` using quadratic space. Edges to $\bot$ have been suppressed for readability. Furthermore, for the sake of readability, we depict the levels being permuted rather than the variables being substituted.

is reversed; for readability, we do not show the variables being substituted but instead the levels being permuted. The size of the output BDD, $T$, is also exactly $n$. Consider the state of `bdd_replace(`$f$`, `$\pi$`)` after having evaluated the low branch of the node for variable $x_{n-1}$. This node is reached by following the high-most path with BDD nodes, $x_1, x_3, \ldots, x_{n-1}$. Each node on this path has also resolved its low branch starting at $x_2, x_4, \ldots, x_n$ but not yet its high branch. As depicted in Fig. 9.3b, the reversal of the levels makes each low branch a different subtree. The combined size of these chains is

$$\sum_{i=1}^{n/2} i = \frac{n(n+2)}{8} = \frac{n}{4} + \frac{n^2}{8} = \frac{N}{4} + \frac{n \cdot T}{8} \quad .$$

Yet, Fig. 9.3 does not prove an $\Omega(N + n \cdot T)$ lower bound since both $N$ and $T$ have been fixed to $n$ instead of being free.

In fact, as $T$ grows larger, the time and space usage seemingly grows closer to an $\Omega(N+T)$ lower bound. For example, consider the BDD $f$ in Fig. 9.4a for the formula $\prod_{i=1}^{n/2}(x_{2i} \equiv x_{2i+1})$ with $\frac{3n}{2}$ BDD nodes. Furthermore, consider the substitution $\pi$ that groups the odd and even variables together, i.e. $x_i$ is mapped to $x_{i/2+1}$ if $i$ is odd and to $x_{n/2+i/2}$ otherwise. As shown in Fig. 9.4b, `bdd_replace(`$f$`, `$\pi$`)`'s output consists of $2^{n/2+1}$ BDD nodes; again, we depict the variables being reordered rather than substituted. Each recursive result has an exponential size of $\sim 2^{i/2+1}$ BDD nodes where $i$ is the number of variables in its support. Yet, the size of all these intermediate BDDs only sums up to $\sim 2^{n/2+2}$, i.e. $\sim 2T$.




(a) Input (b) Output

Figure 9.4: Example of `bdd_replace` using linear space. Edges to $\perp$ have been suppressed for readability. Furthermore, for the sake of readability, we depict the levels being permuted rather than the variables being substituted.

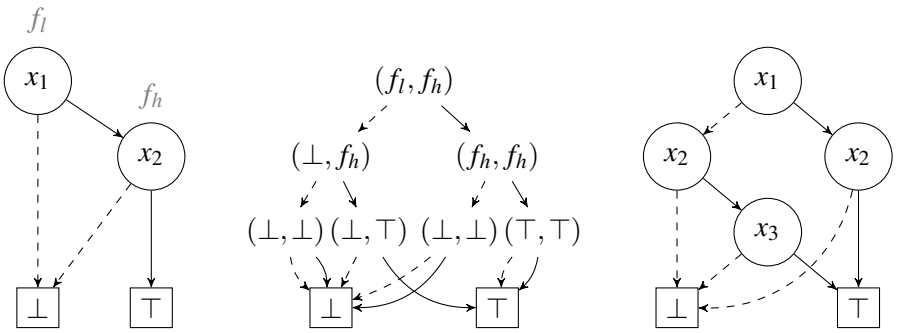
Finally, Lars Arge's analysis of the Apply algorithm [12, Section 3.2] provides BDDs that can be reapplied here. In particular, there exists inputs and outputs such that `bdd_replace` needs $\mathcal{O}(N)$ or $\mathcal{O}(T)$ I/Os to traverse them and/or to use its memoisation tables.

## 9.3 The `bdd_replace` function with Nested Sweeping

As mentioned previously, we only focused on *monotone* variable orderings in Chapter 8. This assumption was exploited in multiple ways to simplify implementation and improve performance.

For the general non-monotone case, we can translate the `bdd_replace` in Fig. 9.1 to the time-forward processing paradigm of Adiar. Since `bdd_replace`'s control-flow is similar to the one in `bdd_exists` and `bdd_forall`, i.e. both children are first resolved to then merge them with a nested product construction, the translation can be done with the nested sweeping framework from Chapter 7. Here, the outer bottom-up sweep simulates lines 2 and 3 in Fig. 9.1 while the inner top-down sweep executes the `bdd_correctify` on lines 6–19 and the inner bottom-up returns the result to the outer sweep as on line 4.

The basic idea of the algorithm is preserved, but the traversal and execution order is changed from depth-first to be bottom-up level by level. But, since the time-forward processing algorithms are levelised, the use of nested sweeping moves an entire level down at once. That is, the entire operation works akin to insertion sort: bottom-up each level is in its entirety moved down to its new position.




(a) Input (b) Unreduced Output (c) Reduced Output

Figure 9.5: Example of how `bdd_correctify(`$f_l$`, `$f_h$`, 3)` without guarded node-creation can create a suppressible BDD node.

**Lemma 9.5.** *Assuming $f$ is a reduced BDD, the result of each inner top-down sweep of `bdd_replace(`$f$`,`$\pi$`)` is also a reduced BDD forest.*

*Proof.* Since $f$ is reduced, no BDD node in $f$ is a duplicate of any other [38]. Hence, every BDD node represents a unique subfunction of $f$. Since the variable substitution, $\pi$, is injective, the subfunctions are also unique after substitution.

Furthermore, since $f$ is reduced, no BDD node in $f$ is suppressible, i.e. there exists no node $f'$ in $f$ such that $f'.\texttt{low} = f'.\texttt{high}$ [38]. Hence, as `bdd_correctify` is initially invoked, $f_l$ and $f_h$ are two different subfunctions of $f$. Assume for contradiction, that a suppressible node is constructed on line 19 as part of a nested `bdd_correctify` sweep. In particular, consider the first, i.e. top-most, such node created. In this case, the variables $f_l'$ and $f_h'$ on lines 10–18 in Fig. 9.1 resolve to the same BDD while $f_l$ and $f_h$ are not the same. Let us consider how this could happen for each of the three cases on lines 10–18:

- $f_l.\texttt{var} = f_h.\texttt{var}$:

  For $f_l'$ and $f_h'$ to be the same, the pairs $(f_l.\texttt{low}, f_h.\texttt{low})$ and $(f_l.\texttt{high}, f_h.\texttt{high})$ are the same. For this to be the case, $f_l$ and $f_h$ are duplicate nodes which contradicts the assumption that $f$ is reduced.

- $f_l.\texttt{var} < f_h.\texttt{var}$:

  For the results to be the same, the pair $(f_l.\texttt{low}, f_h)$ must be equal to $(f_l.\texttt{high}, f_h)$, i.e. $f_l.\texttt{low} = f_l.\texttt{high}$. Hence, $f_l$ in $f$ is a suppressible node and $f$ is not reduced.

- $f_l.\texttt{var} > f_h.\texttt{var}$:

  This case is symmetric to $f_l.\texttt{var} < f_h.\texttt{var}$ above.

Hence, the node creation on line 19 is guaranteed to not be suppressible. Yet, the guard on line 8 in Fig. 9.1 is needed as shown in Fig. 9.5. This happens due to $f_l$ becoming the same as $f_h$ because of lines 14–15 or lines 17–18 (the pairs $(f_h, f_h)$ and

$(\bot, \bot)$ in Fig. 9.5b, respectively). Yet, as argued above for line 8, suppressing this node cannot propagate upwards. □

One could hope Lemma 9.5 implies that the result of each inner top-down sweep has size at most $T$. This is sadly not the case. Intuitively speaking, where the depth-first implementation may create up to the input's depth number of separate BDDs, our time-forward processing variant may create up to the input's width number of copies of the output. In particular, the algorithm's space complexity is as follows.

**Proposition 9.6.** *The space complexity of* `bdd_replace(`$f$`,`$\pi$`)` *with nested sweeping is* $\mathcal{O}(N + T \cdot \max_i(C^{\emptyset}_{1:f[i]}))$*, where* $\max_i(C^{\emptyset}_{1:f[i]})$ *is the maximum* 1*-level cut.*

*Proof.* To do a bottom-up traversal of $f$, nested sweeping first needs to create the transposition of the input [178]. This uses an additional $N$ space.

After this, the result is created bottom-up. By Lemma 9.2 each subtree of the intermediate output forest is a restriction of the final output. Due to Lemma 9.5, each subtree is reduced and hence has size at most $T$. Prior to processing level $i$, there may be a root in the intermediate forest for each arc between level $i$ and the levels below, i.e. $C^{\emptyset}_{1:f[i]}$ arcs where $C^{\emptyset}_{1:f[i]}$ is the 1-level cut between levels $i$ and $i+1$. That is, the intermediate output forest may have up to $\max_i(C^{\emptyset}_{1:f[i]})$ number of roots and hence a size of up to $\max_i(C_{1:f[i]}) \cdot T$ BDD nodes. The nested sweeping framework in Chapter 7 may store up to two intermediate forests at the same time. In total, the outer and inner sweeps use at most $2\max_i(C^{\emptyset}_{1:f[i]}) \cdot T$ space.

The outer and inner priority queues contain collectively arcs of either graph, i.e. they contain at most $2(N + T \cdot \max_i(C^{\emptyset}_{1:f[i]}))$ arcs. □

Both examples in Figs. 9.3 and 9.4 have a maximum 1-level cut of size 2. That is, in either case, the space complexity is $\mathcal{O}(N+T)$. Maybe surprisingly, substituting from Fig. 9.4b back to Fig. 9.4a will also only require $\mathcal{O}(N+T)$ space. Yet, the bound in Proposition 9.6 is tight. The variable substitution used to create Fig. 9.4b from Fig. 9.4a is designed to construct a BDD whose shape resembles an exponentially sized diamond. One can also turn Fig. 9.4a into a BDD with two separate exponentially sized diamonds. To do so, one can use a substitution that groups the first $n/2$ even and odd and variables together and then groups the remaining even and odd variables together. Taking this double-diamond BDD as the input and using a substitution that swaps the order of the two diamonds, one will create an intermediate forest of size $2^{n/4} \cdot T = C^{\emptyset}_{1:f[i]} \cdot T$. In particular, one will create a separate copy of the bottom-most diamond for each of the $2^{n/4}$ nodes at the widest level of the top-most diamond.

**Proposition 9.7.** *The time and I/O complexity of* `bdd_replace(`$f$`,`$\pi$`)` *with nested sweeping is* $\mathcal{O}(sort(T \cdot \sum_{i=1}^{n} C^{\emptyset}_{1:f[i]}))$.

*Proof.* $\mathcal{O}(\mathrm{sort}(N))$ I/Os are spent on transposing the input by sorting its $2N$ arcs [178] (Chapter 7). Another $\mathcal{O}(\mathrm{sort}(N))$ I/Os is spent on the outer sweep's priority queue to forward results from the substitutesd to the to-be substituted levels [178].

The cost of each inner sweep depends on the size of the BDD forests it is processing [178, 180] (Chapters 3 and 7). As argued in the proof of Proposition 9.6, the maximum size of the forest at level $i$ is $C^{\emptyset}_{1:f[i]} \cdot T$. There are a total of $n$ variables, each of which is moved downwards with a nested sweep. The total size of all forests is at most $T \cdot \sum_{i=1}^{n} C^{\emptyset}_{1:f[i]}$. The final bound follows from [11] together with $\sum_{i=1}^{n} C^{\emptyset}_{1:f[i]} > N$ guaranteeing that the $\mathcal{O}(\text{sort}(T \cdot \sum_{i=1}^{n} C^{\emptyset}_{1:f[i]}))$ I/Os exceeds the $\mathcal{O}(\text{sort}(N))$ spent on the outer sweep.

The time-complexity follows from the same line of reasoning. □

```
1   zdd_replace(A, π):
2     l := zdd_replace(A.low, π)
3     h := zdd_replace(A.high, π)
4     return zdd_correctify(l, h, π(A.var))
5
6   zdd_correctify(A_l, A_h, ℓ)
7     if ℓ < A_l.var ∧ ℓ < A_h.var
8       return A_h ≠ 0 ? Node { ℓ, A_l, A_h } : A_l
9
10    if A_l.var = A_h.var
11      A'_l := zdd_correctify(A_l.low, A_h.low, ℓ)
12      A'_h := zdd_correctify(A_l.high, A_h.high, ℓ)
13    else if A_l.var < A_h.var
14      A'_l := zdd_correctify(A_l.low, A_h, ℓ)
15      A'_h := zdd_correctify(A_l.high, 0, ℓ)
16    else // A_l.var > A_h.var
17      A'_l := zdd_correctify(A_l, A_h.low, ℓ)
18      A'_h := zdd_correctify(0, A_h.high, ℓ)
19    return Node { min(A_l.var, A_h.var), A'_l, A'_h }
```

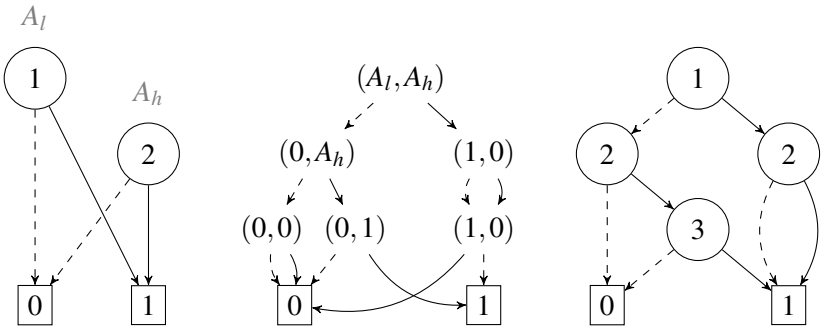
Figure 9.6: The `zdd_replace` algorithm

## Accounting for ZDD Semantics

The logic on lines 10–18 in Fig. 9.1 are based around the shape of suppressed nodes in BDDs. To make it correct for Zero-suppressed Decision Diagrams [140], one only needs to change this logic to account for the suppressed arc to the 0 leaf as shown in Fig. 9.6. In Adiar, this is easily implemented via the generic interface for the diagram's semantics which was introduced in Chapter 4.

Lemmas 9.2 and 9.5 also have to apply to ZDDs for all results above to carry over to ZDDs. The proof for Lemma 9.2 applies trivially to ZDDs. Hence, we only prove the ZDD variant of Lemma 9.5.

**Lemma 9.8.** *Assuming $A$ is a reduced ZDD, the result of each inner top-down sweep of `zdd_replace(A,π)` with nested sweeping is also a reduced ZDD forest.*




(a) Input (b) Unreduced Output (c) Reduced Output

Figure 9.7: Example of how `zdd_correctify(`$A_l$`, `$A_h$`, 3)` without guarded node-creation can create a suppressible ZDD node.

*Proof.* The argument against the creation of duplicate ZDD nodes carries over from the proof of Lemma 9.5. Hence, only the non-suppressiblity of line 19 requires a revised argumentation.

Since $A$ is reduced, then $A_l$.`high` and $A_h$.`high` will never be the 0 leaf. That is, $A_l$.`high` on line 12 and 15 and $A_h$.`high` on line 12 and 18 will always be a ZDD for a non-empty set of integers. Hence, $A'_h$ cannot collapse to the 0 leaf and the node creation on line 19 is guaranteed to not be suppressible. □

The node creation at level $\ell$ (line 8) still needs to be guarded, as is evident in Fig. 9.7. Yet, as argued above this suppression cannot propagate upwards.

## Optimisations for `bdd_replace` with Nested Sweeping

### Accounting for Non-Swapping Levels

Not all nested sweeps of `bdd_correctify` will move a level downwards. In many cases, the level is not out-of-order with respect to the levels below it. Instead, it may only need to be relabelled. In this case, one can directly relabel the variables in the outer bottom-up sweep similar to one of the optimisations for monotone substitutions in Chapter 8. In other words, the nested sweep needs only to be invoked on levels that are out-of-order, i.e. variables that make $\pi$ non-monotone. Going back to the analogy with insertion sort, one only needs to start a nested sweep for the levels that need to be swapped at least once.

Let $\Delta \triangleq \{x_i \mid \exists x_j > x_i \text{ in } f \text{ s.t. } \pi(x_j) < \pi(x_i)\}$ be the subset of variables that are out-of-order. Skipping nested sweeps for $x_i \notin \Delta$, i.e. variables that are are not out-of-order, decreases the time and I/O complexity of `bdd_reduce` with nested sweeping down to $\mathcal{O}(\text{sort}(N) + \text{sort}(T \cdot \sum_{i \in \Delta} C^{\emptyset}_{1:f[i]}))$. The space complexity is also decreased to $\mathcal{O}(N + T \cdot \max_{i \in \Delta}(C^{\emptyset}_{1:f[i]}))$. In the worst case, all levels need to be moved downward (such as is the case in Fig. 9.3), i.e. $\Delta = \{x_1, x_2, \ldots, x_n\}$. Hence, this does not change the worst-case performance of `bdd_replace` in Proposition 9.7.

In practice, this ought to skip most wasteful computations and hence improves performance significantly. The implementation of nested sweeping in Chapter 7 already supports that only a subset of the BDD's levels require a nested sweep.

#### Copy Nodes below Target Level

After having processed the 2-ary requests downto the target level, $\ell$ in Fig. 9.1, the nested `bdd_correctify` sweep is merely creating a copy of all the BDD nodes as-is. Since these nodes are already reduced (both with respect to Bryant's definition in [38] and the stronger notion in Chapter 3 [180]) there is no need to redo all of the work of reducing them. Instead, the top-down sweep can stop after level $\ell$ and merely output all remaining arcs in the priority queue to the nodes below. Then, all nodes below are copied in a simple scan prior to running the bottom-up `Reduce` sweep at levels $\ell$ and above.

In practice, this can safe a lot of time and I/Os if variables are not moved far.

#### Redundancy of Per-level `Reduce`

The expensive logic to suppress and merge nodes in the outer and inner bottom-up Reduce sweeps is redundant due to Lemma 9.5. Hence, these computations can be skipped to reduce the constant in the running time and the I/O cost of Proposition 9.7.

Having this idea in the back of our mind, the logic for doing so was already implemented for the `bdd_exists` and `bdd_forall` operations in Chapter 7. Something akin to Lemma 9.5 does not apply to these quantification operations. Yet, doing so would simplify the bottom-up computations at the cost of making the intermediate BDDs slightly larger in practice. The hope was that this decrease in computation time during the bottom-up Reduce sweeps outweighed the additional cost of the top-down sweeps working on larger unreduced graphs. In practice, it seemed to improve performance slightly. But, the results were unclear and so the idea was left out of Chapter 7. In the case of `bdd_replace`, skipping the logic for suppressing and merging nodes is not at the cost of the BDD's size. Hence, this idea has much better grounds to have a positive impact in this setting.

Yet, skipping these computations also removes the guarantee of the BDD nodes being sorted. This does not break canonicity with respect to [38] but it does break the stronger notion in Chapter 3. This makes the final BDD produced with this optimisation violate one precondition for the linear-scan equality checking algorithm in Chapter 3 and so the slower $\mathcal{O}(\text{sort}(N))$ algorithm would have to be used instead. To mitigate this, the implementation of nested sweeping already allows the canonicity computations to be enabled as part of the final bottom-up `Reduce` sweep.

## 9.4 Exchange and Jumps

A relevant special case of $\pi$ is if it only swaps two arbitrary variables $x_i$ and $x_j$ [94, 137]. To do so, the algorithm from Section 9.3, would have to separately move

the variables $x_i, x_{i+1} \dots x_{j-1}$ down one by one. Instead, we aim to design a `bdd_swap` operation that computes `bdd_replace(`$f$`,` $(x_i, x_j)$`)` in a single sweep. In particular, we aim for the following complexity.

**Proposition 9.9.** *`bdd_swap(`$f$`,` $x_i$`,` $x_j$`)` uses $\mathcal{O}(sort(N+T))$ time and I/Os.*

**Proposition 9.10.** *Computing `bdd_swap(`$f$`,` $x_i$`,` $x_j$`)` requires $\mathcal{O}(N+T)$ space.*

Furthermore, $T$ is at most $3N$ when swapping two variables [71]. The algorithm to do so can be split into the cases addressed below; each case is named according to [31, 94, 137]. Unlike in Propositions 9.6 and 9.7, each sweep below have no intermediate state where a BDD forest can become larger than $T$.

### Jump-Down

If $x_i < x_j$ and $x_j$ is not present in the BDD of $f$, i.e. $x_j$ is not in the support of $f$, then $f'$ can be computed with a single top-down sweep that spawns 2-ary requests for `bdd_correctify` at level $x_i$. These requests are then resolved on the remaining levels. This is similar to single-variable quantification in Chapter 3 and saves the initial transposition and reduction in Section 9.3. The case for $x_i > x_j$ and $x_i$ not being present is symmetric.

This algorithm requires $\mathcal{O}(\text{sort}(N+T))$ time and I/Os due to Lemma 9.2 and [11]. Likewise, it only uses $\mathcal{O}(N+T)$ space.

### Jump-Up

Without loss of generality, assume $x_i < x_j$ and $x_i$ is not present in the BDD of $f$ (the case for $x_i > x_j$ and $x_j$ not being present is symmetric). Then, $x_j$ can be propagated upwards through the BDD to the empty level of $x_i$ by extending the bottom-up `Reduce` sweep as follows:

- At the level of $x_j$, do not output any nodes. Instead, for each node $n$ forward $n$.`low` and $n$.`high` to $n$'s parents with an additional payload of $x_j := \bot$ and $x_j := \top$, respectively.

- For each node $n$ on a level between $x_i$ and $x_j$, there are now (possibly) twice as many arcs for $n$ (one for either assignment to $x_j$). Create two copies of $n$ with each respective payload. Here, one may have to reuse an arc with an empty payload twice to match with a doubled arc.

- When processing level $x_i$, insert a new node with variable $x_i$ for each pair of arcs to some node above the level of $x_i$. The node's low child is obtained from the arc with payload $x_i := \bot$. Symmetrically, the high child stems from the request with payload $x_i := \top$. Forward a single arc to its parents.

Intuitively, we propagate a single bit indicating the assignment to the variable at $x_j$ from level $x_j$ up to $x_i$. In this case, the nodes in-between need to be doubled to reflect the additional bit they store.

Most of the above logic is resolved by extending the sorting order of the requests within the priority queue to account for the payload. Specifically, the ordering should be extended such that the tiebreaker for arcs with the same source should first based on its payload and secondly as it was done previously in Chapter 3. Furthermore, if a payload is present in a request then the request's level is the maximum of the targeted parent's level and the level of $x_i$.

One would hope that Lemma 9.5 would also apply to this bottom-up sweep. Yet, Fig. 9.8 shows that this way of moving $x_i$ above another variable, $x$, can create duplicate nodes. Even though this specific example is built around the two nodes being each other's negations, the addition of complement edges [106, 124] (Section 1.3) does not resolve this: despite Fig. 9.8 translates to Figs. 9.9a and 9.9b where there are no duplicate nodes, Figs. 9.9c and 9.9d provide additional nodes where the bottom-up sweep would create duplicates of the ones produced by Figs. 9.9a and 9.9b, respectively. Hence, unlike in Section 9.3 one cannot skip the two expensive sorting steps of the `Reduce` in Chapter 3.

At most $2N$ unreduced nodes with their $4N$ arcs are passed through the priority queues. Hence, the complexity is still $\mathcal{O}(\mathrm{sort}(N))$ time and I/Os using $\mathcal{O}(N)$ space.

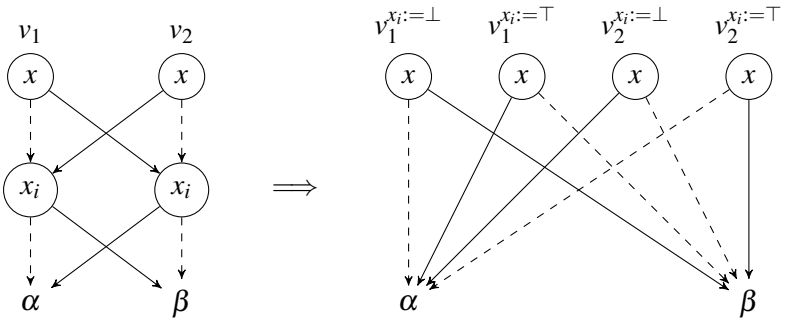



Figure 9.8: Example of bottom-up variable propagation creating duplicate nodes.

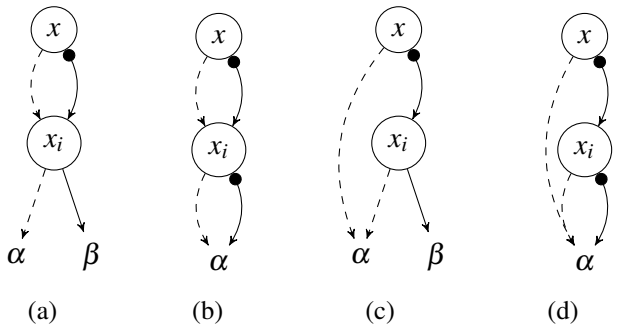



Figure 9.9: Extension of Fig. 9.8 to also cover *complement edges*.

### Exchange

If both $x_i$ and $x_j$ are present in $f$ then one can combine the *Jump-Up* `Reduce` sweep above to move the deepest variable up to the shallowest's level and then start a nested `bdd_correctify` sweep to move the shallowest down.

The *Jump-Up* sweep creates an intermediate BDD forest of at most size $2N$ in $\mathcal{O}(\text{sort}(N))$ time and I/Os. Since this is a reduced and restricted forest of the output, the nested `bdd_correctify` must, due to Lemma 9.2, also create a restricted forest of the output. Hence, its size is at most $T$. The reduction of the remaining levels above both $x_i$ and $x_j$ touches at most an additional $N$ or $T$ number of BDD nodes. Hence, it uses at most $\mathcal{O}(\text{sort}(N+T))$ time and I/Os and uses only $\mathcal{O}(N+T)$ space.

## 9.5 Adjacent Variable Swaps

The variable swapping [81] subprocedure used in the popular sifting reordering algorithm [161] (see also Chapter 10) is incompatible with the layout of the BDD nodes needed for time-forward processing. This is due to [161] exploits that swapping two adjacent variables, $x_i < x_j$, is a local transformation, i.e. only the BDD nodes on the two swapped levels are affected and no others need updating [81]. As described in Chapter 3, this is not the case for Adiar's algorithms since its nodes do not use pointers but unique identifiers which includes their level. Hence, one would have to recompute the entire BDD graph to update all parents of the swapped levels.

For simplicity, let $x_i < x_j$ be the only two adjacent variables that need to be swapped due to $\pi$. That is, $\pi(x_i) = x_j$, $\pi(x_j) = x_i$, and $\pi(x_k) = x_k$ for all other variables, $x_k$. The result of swapping $x_i$ and $x_j$ can be computed in a single top-down sweep as follows:

- Above level $x_i$, follow the structure of $f$ and output the transposed DAG.
- Similar to single-variable quantification in Chapter 3, turn the 1-ary requests for nodes at level $x_i$ into 2-ary requests for their children.
- At level $x_j$, process the 2-ary requests as in `bdd_correctify` but output their nodes with variable $\pi(x_j) = x_i$ instead.
- At level $\pi(x_i)$, turn the 2-ary requests into nodes with variable $\pi(x_i) = x_j$.
- Complete the remaining 1-ary requests beyond level $\pi(x_j)$. To improve the time and I/Os used, one can, similar to one of the optimisations in Section 9.3, copy over the BDD nodes directly.

The key idea in the steps above is to relabel the BDD nodes with variable $x_j$ into $\pi(x_j) = x_i$ as part of the top-down sweep. This frees up the level of $x_j$ prior to placing the new $\pi(x_i) = x_j$ nodes in it.

**Proposition 9.11.** *Computing* `bdd_replace(`$f$`, `$\pi$`)` *is possible in* $\mathcal{O}(N)$ *space if* $\pi$ *only consists of a single swap of two adjacent variables.*

*Proof.* Based on [31], the size of the reduced output, $T$, is at most $2N + \mathcal{O}(1)$. The rest follows from applying Lemma 9.2 anew. □

**Proposition 9.12.** *Computing* `bdd_replace(f, π)` *is possible in* $\mathcal{O}(sort(N))$ *time and I/Os if* $\pi$ *only consists of a single swap of two adjacent variables.*

*Proof.* This again directly follows from [31]. □

By forwarding requests from $x_i$ to $x_j$ and then to $\pi(x_i)$, this implicitly includes a sorting step to identify duplicate nodes. But, it does not make it canonical at levels $\pi(x_i) = x_j$ and above with respect to the stricter definition in Chapter 3. If this is needed then one has to sort the levels at $\pi(x_i) = x_j$ and above with a `Reduce`.

Since swapping two adjacent variables is a local operation, then the sweep described above can easily be extended to handle $|\Delta| \leq n/2$ swaps of adjacent but also disjoint pairs of variable within a single sweep. For example, for $n = 4$ and $\pi(x_0) = x_1$, $\pi(x_1) = x_0$, $\pi(x_2) = x_3$, and $\pi(x_3) = x_2$, $\pi$ consists of $|\Delta| = 2$ adjacent but also disjoint variable swaps. Since the operations are disjoint and only affect the BDD locally, then the asymptotic performance is also unaffected.

**Corollary 9.13.** *Computing* `bdd_replace(f, π)` *where* $\pi$ *only contains adjacent variable swaps is possible in* $\mathcal{O}(N)$ *space.*

**Corollary 9.14.** *Computing* `bdd_replace(f, π)` *where* $\pi$ *only contains adjacent variable swaps is possible in* $\mathcal{O}(sort(N))$ *time and I/Os.*

That is, the running time of the time-forwarded variable swapping described above is independent of $|\Delta|$. In fact, the conventional variable swapping procedure uses $\mathcal{O}(N)$ time, space, and I/Os if all levels are swapped in pairs of two. Hence, the closer $|\Delta|$ is to $n/2$, the closer the time complexity of the time-forwarded swapping above matches the conventional approach.

### The Devil in the Implementation Details

One should notice, that the sweep described above depends on treating the level of $x_j$ and $\pi(x_i) = x_j$ as two separate levels. This makes the levelised priority queue in Chapter 3 not immediately compatible with this sweep. In particular, we need to treat $\pi(x_i)$ as a separate $x_{j+1/2}$ bucket in the levelised priority queue. Yet, $j + 1/2$ is not an integer.

The three ideas that follow attempt to resolve this. But, they are unsatisfactory.

- One can use a non-levelised priority queue. Yet, this would be at the cost of reverting the performance increase in Chapter 3. In particular, the running time is decreased with $\sim 30\%$ or even $\sim 60\%$.
- Since $2(j + 1/2)$ is an integer, we can use the on-the-fly affine relabelling from Chapter 8 to make room for the $\pi(x_i)$ bucket. Yet, this requires halving $n$'s maximum possible value (currently $2^{21} - 3$) to leave space for doing so.

- The nodes on level $\pi(x_i)$ can be processed in a separate bucket next to the priority queue. This would decrease the amount of memory available which could impact performance. Yet, the amount of available memory is still more than with nested sweeping in Section 9.3.

To properly solve this, we can combine the last two ideas above to use the levelised priority queue almost as-is. The level of each request is, separately from the BDD nodes in $f$, multiplied by two. This makes space for an additional bucket in-between each level in $f$. Unlike affine relabelling, the buckets identifiers may exceed the maximum BDD label by more than a factor of two. The sweep itself only needs to account for this slight decoupling of the priority queue's level.

## 9.6 Related Work

We previously designed algorithms for variable substitution in Chapter 8. Yet, we focused only on monotone substitutions in the context of model checking, i.e. substitutions where the order between variables, $x_i < x_j$, is preserved after substitution, $\pi(x_i) < \pi(x_j)$. This implies only the labels need changing but not the BDD graph. The algorithm in Section 9.3 also covers the general case of non-monotone substitutions which was left as future work.

**Variable Substitution:** The `bdd_replace(`$f$`, `$\pi$`)` function proposed in Section 9.3 is directly based on the one in BuDDy [119] as described in Section 9.2. As Propositions 9.4 and 9.6 show, by changing the processing order from being depth-first to be level by level, we move complexity to be sensitive to the input's width rather than its depth. Furthermore, we add an additional log-factor. Based on our experiments in [178, 180], this log-factor is unnoticeable in practice. Yet, the dependence on the BDD's width could – especially for larger BDDs – turn into a considerable issue.

**Variable Swapping:** In conventional BDD packages, adjacent variable swaps [81] are usually used to change the order of the entire BDD forest [119, 155, 182]. This operation is also at the heart of the dynamic variable reordering operation called sifting [161] (see also Chapter 10). As mentioned in Section 9.5, a single variable swap is not cheap in the context of Adiar's BDD representation. Yet, as the number of swaps, $|\Delta|$, in $\pi$ get closer to $n/2$, our proposed alternative in Section 9.5 becomes on a par with the operation in [81, 161].

Similarly, as shown in Section 9.4, moving a single variable multiple levels can also be done asymptotically as well as with repeated variable swaps.

It is also possible to implement Rudell's variable swapping procedure as an I/O-efficient time-forward processing sweep. Yet, this does not resolve the above mentioned issue of also having to recompute the BDD graph where no variables are swapped. Furthermore, unlike the algorithm in Section 9.5, such a procedure would generate an unreduced OBDD [161] of up to $2N$ BDD nodes [31, 71]. To resolve

this, one would have to use a `Reduce` sweep afterwards to create the final ROBDD of size $T$. This procedure would also require reading the level of $x_j$ twice. We refer to Appendix 9.A for more details.

**Restrict-guided Rebuilding:** The respective authors in [21, 166, 186] designed algorithms to rebuild a BDD with any variable ordering using only $\mathcal{O}(N+T)$ space. Each algorithms in [21, 166, 186] is based on repeatedly using the `bdd_restrict` algorithm on the to-be substituted BDD. This fundamental BDD operation computes the subfunction where one or more input variables are fixed to some boolean values. In the case of [21, 166, 186], the variables' assignment reflects one (or more) paths in the reordered BDD under construction.

Bern, Meinel, and Slobodová's BDD reconstruction algorithms in [21] (based on [133]) runs in $\mathcal{O}(N \cdot T)$ time. The algorithm follows a depth-first traversal of the input and the output which needs $\mathcal{O}(N \cdot T)$ I/Os in the worst case [12, 58].

The algorithm by Tani and Imai [186] uses a breadth-first approach instead. Doing so, they decrease the auxiliary memory usage of the algorithm in [21] downto $\mathcal{O}(N+W_T)$ where $W_T$ is the output's width as a quasi-reduced OBDD [198], i.e. an OBDD where duplicate nodes are merged but redundant ones are not suppressed. This quasi-reduced OBDD is constructed by means of an auxiliary $\mathcal{O}(N)$ sized hash table. Since the breadth-first traversal in [186] only constructs a quasi-reduced OBDD, they use separate `Reduce` phase afterwards to compute the final ROBDD. This is similar to our `Apply`–`Reduce` tandem in Chapter 3. Since redundant nodes are unsuppressed, no arcs skip past any level. Furthermore, each unsuppressed node may turn one arc in an ROBDD into two.

The approach by Savický and Wegener [166] runs in $\mathcal{O}(N \cdot T \log T)$ time, traverses the input and output level by level, and is reliant on sorting its recursions. This makes it seem quite identical to the top-down sweeps in Chapter 3. Hence, Clausen and Nielsen attempted in [58] to repurpose this algorithm for the context of the time-forward processing paradigm in Adiar [58]. The semi-transposed output is during its construction also used to obtain the assignment for the nested calls to `bdd_restrict` [58]. But, the many nested calls to `bdd_restrict` makes an implementation in Adiar too slow in practice [58]. Furthermore, the sorting predicate in [166] results in up to

$$\mathcal{O}(\text{sort}(N) \cdot T \log_2(T) + \tfrac{T^2}{B} \log_2(T))$$

I/Os which makes it bad in theory [58]. It neither performs well in practice [58].

Our algorithm in Section 9.3 relies only implicitly on variable restriction (See Lemma 9.2). That is, the algorithm itself has no nested calls to `bdd_restrict`. This allows its time, I/O, and space complexity to also be output-sensitive with respect to the final ROBDD. Yet, this is at the additional cost of using up to quadratic space rather than only a linear amount.

## 9.7 Conclusion

We have designed an algorithm capable of applying a variable substitution $\pi$ using $\mathcal{O}(\text{sort}(N) + \text{sort}(T \cdot \sum_{i \in \Delta} C^{\emptyset}_{1:f[i]}))$ time and I/Os and $\mathcal{O}(N + T \cdot \max_{i \in \Delta}(C^{\emptyset}_{1:f[i]}))$ space, where $\Delta$ is the set of variables that are out-of-order with respect to $\pi$, $N$ and $T$ are respectively the input and output size, and $C^{\emptyset}_{1:f[i]}$ is the size of the input's 1-level cut at level $i$. Furthermore, if $\pi$ only swaps two non-adjacent variables (Section 9.4) or swaps many adjacent variables (Section 9.5) then the result can be obtained in $\mathcal{O}(\text{sort}(N+T))$ time and I/Os and $\mathcal{O}(N+T)$ space. Only some relatively small constants are hidden in the $\mathcal{O}$-notation. Hence, these algorithms can be used in practice as part of large-scale BDD computations.

The algorithms in Section 9.4 can be generalized to handle multiple variable substitutions during a single sweep as long as the levels affected by each substitution do not overlap. This generalization is possible without increasing its asymptotic cost. Furthermore, the *Jump-Down* sweep can also be used as the initial transposing sweep of `bdd_correctify` in Section 9.3 to further decrease $|\Delta|$.

Our general algorithm in Section 9.3 has two issues that need further attention. Most importantly, its space complexity is quadratic with respect to the input's and the output's size rather than only linear like the algorithms in [21, 166, 186]. As the goal of designing I/O-efficient algorithms is to manipulate large BDDs in external memory, this can turn into a major issue in practice. Furthermore, the number of nested sweeps, $|\Delta|$, should be further decreased if possible. Sections 9.3 to 9.5 attempt to identify cases that can be computed cheaply. To further improve the algorithm's performance in practice, more work should be done in this direction. For example, does there exists cases where an entire group of variables can be moved in a single nested sweep? Can this be done while preserving the output being reduced? Phrasing the questions this way makes both problems seem intertwined. That is, ideas that further decreasing the number of nested sweeps may very well further push the space complexity further to be linear.

For an experimental evaluation, the performance of our basic algorithm in Section 9.3 should be compared to (*i*) its various optimisation in Sections 9.3 to 9.5, (*ii*) the `bdd_replace` in BuDDy (Section 9.2) and equivalent functions, e.g. `bdd_compose`, in other BDD packages, (*iii*) the use of variable swapping, and (*iv*) Clausen and Nielsen's translation* of the algorithm from [166] in [58]. To this end, the BDDs in Figs. 9.3 and 9.4, [58, Proposition 4], and [12, Figure 6b] provide synthetic inputs that can be used to evaluate how performance scales. Using other substitutions than shown Figs. 9.3 and 9.4, e.g. adjacent variable swaps, allows one to also explore other aspects of the algorithms' behaviour, e.g. when a lot of subtrees are reused as-is. Furthermore, the circuit verification benchmark in Chapter 3 provides several real-life BDDs and variable orderings on which the algorithms can be run.

In this chapter, we have assumed the new variable ordering, $\pi$, is known a priori. This is not always the case. We show in Chapter 10 how to repurpose these algorithms

*Available on the `project/2022/reordering` branch of github.com/ssoelvsten/adiar.

into a space-efficient backbone for *dynamic* variable reordering.

## Acknowledgement

We want to thank Anders Benjamin Clausen and Kent Nielsen for their work as part of their BSc project [58] on reimplementing [166] in the context of Adiar's algorithms. Despite their negative results, the collaboration with them have cultivated some of the ideas and perspectives that are vital for our results, e.g. Lemma 9.2. Thanks to Clemens Dubslaff for noticing a major error in our analysis of Propositions 9.6 and 9.7. We also want to thank Casper Moldrup Rysgaard and Peyman Afshani for the valuable discussion on the consequences of Fig. 9.3. Furthermore, thanks to Rysgaard for his help discovering the example in Fig. 9.4. Finally, thanks to Natascha Schalburg for suggesting to change notation and to use case-analysis to prove Lemma 9.2.

## 9.A Rudell's Swap with Time-forward Processing

The swapping procedure in [81, 161] can be turned into a top-down time-forward processing sweep. For simplicity, let us again assume $x_i < x_j$ are the (only) two adjacent variables that need to be swapped due to $\pi$. To be I/O-efficient, the top-down sweep has to defer parts of the logic from [161] across the levels of $x_i$ and $x_j$, as follows:

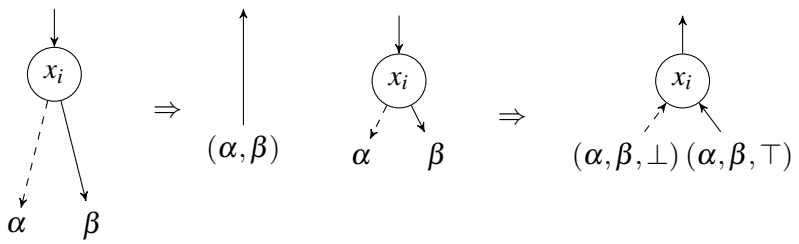
- As shown in Fig. 9.10, each BDD node with variable $x_i$ is forwarded to the level of variable $x_j$. If the node is independent of $x_j$ (Fig. 9.10a), i.e. neither $\alpha.\mathtt{var}$ nor $\beta.\mathtt{var}$ is $x_j$, then the pair $(\alpha,\beta)$ is forwarded to be output as a node with the new vairable $\pi(x_i) = x_j$ later. Otherwise, a node with variable $\pi(x_j) = x_i$ is output with the requests for its children recording the assignment to the decision variable (Fig. 9.10b).

- For each 1-ary request directly forwarded to $x_j$ from a level prior to $x_i$, output them anew with the decision variable $\pi(x_j) = x_i$.

- Now, continue with the 2-ary requests $(\alpha,\beta,b)$ from the BDD nodes previously with variable $x_i$. Similar to the Apply sweep from Chapter 3, one obtains the children of the $x_j$ nodes, $\alpha$ and $\beta$. As shown in Fig. 9.11, create new nodes as follows:

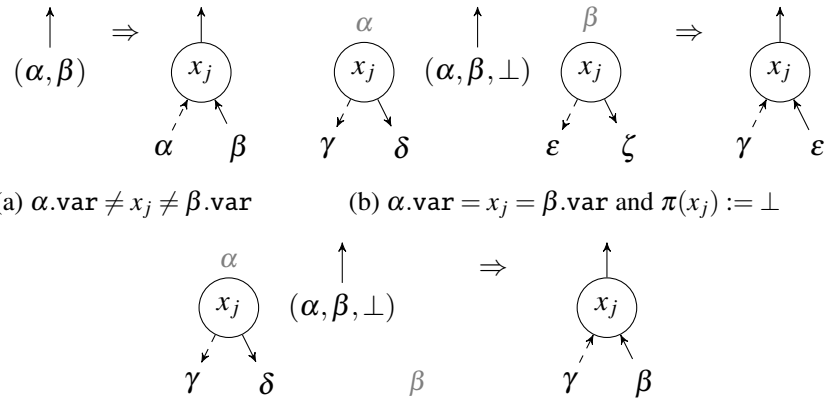
  - If the request does not contain $b$, then output a new node with the forwarded children (Fig. 9.11a).
  - For simplicity, assume both $\alpha.\mathtt{var}$ and $\beta.\mathtt{var}$ in $f$ is the decision variable $x_j$ and that $b = \bot$ as depicted in Fig. 9.11b. To match the evaluation of $\pi(x_j) = x_i$ in $b$, the children $\gamma$ and $\varepsilon$ should be used depending on the choice in $\pi(x_i) = x_j$. Hence, a new node is created with variable $\pi(x_i) = x_j$ and $\gamma$ as its low child and $\varepsilon$ as its high child.

    The case is symmetric for $b = \top$ by using $\delta$ and $\zeta$ as the children instead.




(a) $\alpha.\mathtt{var} \neq x_j \neq \beta.\mathtt{var}$ (b) Either $\alpha.\mathtt{var} = x_j$ or $\beta.\mathtt{var} = x_j$

Figure 9.10: Cases on level for $x_i$ for time-forwarded variable swapping.




(a) $\alpha.\mathtt{var} \neq x_j \neq \beta.\mathtt{var}$ (b) $\alpha.\mathtt{var} = x_j = \beta.\mathtt{var}$ and $\pi(x_j) := \perp$

(c) Either $\alpha.\mathtt{var} \neq x_j$ or $\beta.\mathtt{var} \neq x_j$

Figure 9.11: Cases for new nodes on level for $x_j$ for time-forwarded variable swapping.

  - As shown in Fig. 9.11c, the 2-ary case above where $\alpha.\mathtt{var} = \beta.\mathtt{var} = x_j$ generalises into the 1-ary case where one of the two children does not depend on $x_j$.

- All BDD nodes with neither variable $x_i$ nor $x_j$ follow the structure of $f$ as-is.

To both resolve the 1-ary requests for $x_j$ that need to be output as $\pi(x_j) = x_i$ and the 2-ary requests that need to output afterwards as $\pi(x_i) = x_j$, one needs to scan the level of $x_j$ twice. This costs an additional $\mathrm{scan}(N_j)$ I/Os. It is also possible to defer outputting the 2-ary requests. But, this would cost more than reading the level twice.

The nodes created at the level of $\pi(x_i)$ may be duplicates [161]. Hence, this top-down sweep needs to be followed up by a bottom-up `Reduce` sweep to create the final output of size $T$.

Due to [31, 71], the unreduced output cannot exceed $2N$ and so we can guarantee the time, space, and I/O complexities are not quadratic despite the fact this is a 2-ary product construction. Hence, Propositions 9.11 and 9.12 also apply to this algorithm. The proofs also follow from [31, 71]. Furthermore, since swapping two adjacent variables is a local operation, the above sweep can be generalized to apply multiple swaps as part of the same sweep. Again due to [31, 71], the asymptotic performance is also unaffected and Corollaries 9.13 and 9.14 again apply.

# Chapter 10

# Dynamic Variable Reordering

## 10.1 Introduction

Chapter 9 provides algorithms for changing the variable ordering in a BDD. Yet, in some cases, a good ordering cannot be inferred from the problem to-be solved. In others, a pre-computed order needs to be further improved. In either case, the new variable ordering has to be derived from the BDDs themselves. Finding the optimal variable ordering of a BDD is an NP-complete optimisation problem [29, 187]. Hence, deriving the optimal ordering is computationally too expensive to be done in practice. Instead, one will have to settle with a good enough solution rather than the optimal one. Even though dynamic variable reordering, i.e. deriving and using a better variable ordering on-the-fly, can slow down the BDD computation time by an entire order of magnitude [49], it can be key to finishing the computation [49, 161].

Similar to Chapter 9, we will make a clear distinction between the notions of a *label* (a variable) and its *level* in the BDD. We will consider a variable order $\pi$ as a permutation from the symmetric group $S_n$ which permutes the BDD's levels. We will reuse notation from Chapter 3 and Chapter 7 and also refer to them for more details on BDDs and Adiar's I/O-efficient algorithms. Furthermore, we will once more make use of the notion of $i$-level cuts from Chapter 5.

### Contributions

We propose multiple avenues for implementing external memory dynamic variable reordering based on the algorithms from Chapter 9. Specifically, Adiar's API would be extended with the following function which computes an improved variable ordering $\pi$ together with the resulting BDD, $f[\vec{x} \mapsto \pi(\vec{x})]$.

`bdd_reorder(`$f$`)` : Derive and apply a new ordering $\pi$ that makes $f$ smaller.

In particular, we investigate in Section 10.4 different metaheuristics such as simulated annealing [51, 108], genetic [92] and memetic [145, 147] algorithms, and swarm intelligence algorithms [17, 69, 70, 76, 107]. Furthermore, we also propose in Sec-

tion 10.5 to use the algorithms from Sections 9.4 and 9.5 to recreate the successful sifting algorithm from [161] in an external memory setting.

At the scale of an external memory algorithm, it is vital that our dynamic variable reordering algorithms are space efficient. Since dynamic variable reordering searches for a $\pi$ that makes $f$ smaller, we can guarantee that the size of the final BDD, $T$, is smaller than $f$'s size, $N$. Hence, we aim for our algorithms to only use $c \cdot N$ space where $c$ is a small constant. Furthermore, Sections 10.2 and 10.3 highlight multiple avenues to further decrease the computation time in practice.

## 10.2 Evaluating Multiple Candidates at Once

To support the algorithms in Sections 10.4 and 10.5, we need to be able to evaluate candidate variable orderings. That is, given a candidate $\pi$, we have to compute the size $T$ of $f$ with variable ordering $\pi$. Yet, we would want to maximise the number of candidate variable orderings, $\pi$, we evaluate. We show in the remainder of this section, how to improve upon the algorithms from Sections 9.4 and 9.5 when the resulting BDD is not of interest but only its size.

### Jumps

The *Jump-Up* and *Jump-Down* sweeps from Section 9.4 can be made faster if only the output size $T$ is of interest. In this case, one can skip outputting nodes or arcs and instead merely count the output size.

In Section 9.4, we assumed the target level was known, i.e. the level $x_j$ which $x_i$ should be moved to is given. Yet, it is useful to compute the resulting BDD sizes of moving $x_i$ up or down to any other level. This allows one to identify the best possible level to move $x_i$ to. One can do so with $n$ applications of the algorithms in Section 9.4. Yet, doing so requires $\text{sort}(n \cdot \text{sort}(N))$ time and I/Os. Instead, we would like to be able to evaluate all possible $x_j < x_i$ in a single *Jump-Up* sweep and all possible $x_j > x_i$ in a single *Jump-Down* sweep. To do so, we need to compute the BDD's size as-if $x_i$ is placed at each level, $x_j$, without destroying the information that is needed for later. More precisely, this can be done as follows:

**Jump-Down:** The levels above $x_i$ are trivially the same size as in the original BDD $f$ [72, 80]. Hence, the `bdd_correctify` sweep can start at level $x_i$ with the arcs to said level and the ones that go beyond it.

Prior to processing any level with some variable $x_j > x_i$, one could have chosen to move $x_i$ to the level right above $x_j$. In this case, the size of the final BDD would be all the nodes counted up to this point, the number of unique 2-ary requests in the priority queue (each of which would become a new BDD node), and the size of level $x_j$ and everything beyond it. This output size can be recorded in the output before processing the 2-ary requests for $x_i$ being moved below $x_j$.

In this case, one needs to look at all requests in the priority queue at each level (despite of the level of the individual requests). Hence, the algorithm's complexity

increases up to $\mathcal{O}(\text{sort}(n \cdot N))$ time and I/Os. More precisely, its performance is $\mathcal{O}(\text{scan}(N) + \text{sort}(\sum_{x_j > x_i} C^{\emptyset}_{2:f[j]}))$ where $C^{\emptyset}_{2:f[j]} \leq N + 1$ is the maximum 2-level cut [175] (Chapter 5) of $f$ at the level of $x_j$ and excluding arcs to terminals. On the other hand, it suffices to merely sort all requests for each level rather than using a priority queue. Doing so improves performance in practice [144, 180]. The space complexity is still $\mathcal{O}(N)$.

**Jump-Up:** Similar to *Jump-Down*, the levels below $x_i$ are trivially the same size as the original BDD $f$ [71, 80]. Hence, the bottom-up Reduce sweep may start at the level of $x_i$ with the priority queue being populated with the arcs to the children of the $x_i$ nodes together with the arcs that go past the level of $x_i$.

At each level for some variable $x_j < x_i$, one could place the $x_i$ nodes right below it. If commited to this level, the output consists of the unique nodes that would be created below $x_j$, the accumulated nodes below and the original nodes above. This value is placed in the output. Then, the level of $x_j$ is processed and the arcs with their respective payloads are forwarded to the level above.

Similar to *Jump-Down*, doing so requires $\mathcal{O}(\text{sort}(n \cdot N))$ time and I/Os (more precisely, $\mathcal{O}(\text{scan}(N) + \text{sort}(\sum_{x_j < x_i} \cdot C^{\emptyset}_{2:f[j]}))$ time and I/Os). The space complexity is still $\mathcal{O}(N)$. Yet, the priority queue can again be replaced with a simple sorting step.

## Adjacent Swaps

In Section 9.5, we presented an $\mathcal{O}(\text{sort}(N))$ procedure to apply a variable ordering, $\pi$, that only consists of adjacent variable swaps. This can be improved in the following two ways.

**Skip Reduce:** Since the top-down sweep in Section 9.5 already produces an ROBDD, the `Reduce` sweep from Chapter 3 can be skipped. Instead, merely the size of each level needs to be recorded. By extension, creating the semi-transposed list of $2T$ arcs for the `Reduce` can also be skipped. Doing so improves the running time by about a factor of two.

**Evaluate all Swaps:** One can compute the change in size, $\delta_{i,j}$, by swapping each two adjacent variables, $x_i < x_j$, with two top-down sweep in Section 9.5 – first evaluate all adjacent swaps where $x_i$'s level is even and then again where it is odd.

Yet if, as suggested above, no BDD needs to be output but only each $\delta_{i,j}$ needs to be computed then one can do it in a single $\mathcal{O}(\text{sort}(N))$ top-down sweep. Unlike the algorithm in Section 9.5, this sweep would have to evaluate swaps that overlap. To do so, one only needs some additional $\mathcal{O}(1)$-sized bookkeeping to both compute swapping each level up or down. In particular, one has to evaluate at the same time both the 2-ary requests as-if swapped with the level above while also generating the 2-ary requests as-if swapped with the one below. Yet, this still only requires $\mathcal{O}(N)$ BDD nodes forwarded and evaluated through the priority queues [31, 71]. Hence, it still uses only $\mathcal{O}(\text{sort}(N))$ I/Os and time and $\mathcal{O}(N)$ space.

While this is feasible, the performance gain compared to just running two separate top-down sweeps might be too miniscule to be worth the implementation effort.

## 10.3 Bounds and Auxiliary Heuristics

A lot of previous work has investigated ways to exclude irrelevant variable orderings, e.g. [134, 152], and to evaluate candidate orderings cheaply, e.g. [31, 71]. What follows is an exploration of some of these concepts and how they fit into the context of the external memory algorithms from Chapter 3.

### Symmetric Variables

Two variables $x_i$ and $x_j$ are *symmetric* [170] if exchanging them does not change the BDD and hence its size [152]. By looking at the nodes on two adjacent levels $x_i$ and $x_j$, one can cheaply identify whether they are symmetric and hence merely swapping them has no effect. See [152] for more details.

If both levels are narrow enough to fit into memory, i.e. $N_i + N_j \leq M$, then one can use *levelised random access* [181] (Chapter 6) to identify a pair of symmetric variables in $\mathcal{O}((N_i + N_j)/B)$ I/Os by means of the logic from [152]. Hence, if the width of $f$ is at most $M/2$ then one can identify all symmetric variables that are adjacent in $\mathcal{O}(N/B)$ I/Os. If it is wider than $M/2$, then one would have to use time-forward processing to forward the relevant information needed to identify symmetric adjacent variables. This would require $\mathcal{O}(\text{sort}(N))$ time and I/Os.

Finally, it might be possible to forward the relevant information in a top-down or bottom-up sweep as part of running the `bdd_replace` from Section 9.3 to simultaneously test for adjacent symmetric variables. Doing so would be at no additional asymptotic cost in terms of space and I/Os.

### Variable Blocks

In [134], a variable is confined to stay within a *block* of variables. This is done to prune the number of variable orderings $\pi$ that are unnecessarily explored as part of a dynamic variable reordering algorithm.

The $\beta \in [1, n]$ *blocks* are identified based on local minima in the *function profile*, i.e. the width of each individual level. This profile is already computed as part of the `Reduce` sweeps in Chapter 3. Hence, if only $B$ levels are needed to identify a local minimum, then one can identify the blocks of variables in $\mathcal{O}(\text{scan}(n))$ I/Os. The maximum number of levels in Adiar was in Chapter 8 decreased down to $2^{21} - 3$, i.e. $\sim 2$ MiB. Hence, in practice, $n \leq B = 2$ MiB and so all levels easily fit into memory and can be analysed at once.

### Bounds on Output Size Swapping

It is not be feasible in practice to evaluate each variable ordering. Yet, when searching around the global (or a local) minimum, $\pi$ will only differ slightly from previously resolved variable order. In this case, it might be possible to bound $T$ based on $N$, $n$, and other meta information about said BDD's graph. Examples of such bounds are:

- A monotone substitution results in $T = N$ as shown in Chapter 8.
- Two adjacent levels, $x_i < x_j$, with respective size $N_i$ and $N_j$, create a BDD of size $T$ after being swapped where $N - N_i - \frac{1}{2}N_j < T \leq N + 2 \cdot N_i$ [71].
- A *Jump-Down* of variable $x_i$ down to $x_j$ creates a BDD of size $T$ which is at least $\sum_{k=1}^{i-1} N_k + \max(N_i, 1 + \frac{1}{2}\sum_{k=i}^{j} N_k) + \sum_{k=j+1}^{n} N_k$ [71].
- A *Jump-Down* of variable $x_i$ down to $x_j$ makes $T \leq \sum_{k=1}^{i-1} N_k + (2 + \sum_{k=1}^{n} N_k)^2$ and $T \leq \sum_{k=1}^{n} N_k + 2 + (2^{\Delta_{x_j \ldots x_i}+2} - 2) \cdot N_i$ where $\Delta_{x_j \ldots x_i}$ is the number of levels between $x_j$ and $x_i$ (excluding both) [31].
- A *Jump-Up* of variable $x_i$ up to $x_j$ creates a BDD of size $T$ which is at least $\sum_{k=1}^{j-1} N_k + 1 + \Delta_{x_i \ldots x_j} + \frac{1}{2^{i-j}} N_i + \sum_{k=i+1}^{n} N_k$ where $\Delta_{x_i \ldots x_j}$ is the number of levels between $x_i$ and $x_j$ (excluding both) [71].
- A *Jump-Up* of variable $x_i$ up to $x_j$ makes $T \leq 2\sum_{k=1}^{i-1} N_k + 1 + \sum_{k=i+1}^{n} N_k + 2$ and $T \leq \sum_{k=1}^{j-1} N_k + 3\sum_{k=j}^{i-1} N_k + \sum_{k=i}^{n} N_k + 2$ [31]. That is, $T \leq 2N$ [31]. This also further bounds $T$ if $x_i$ and $x_j$ are adjacent variables [31].

Using the function profile produced next to each BDD in Chapter 3, one can resolve each of the above using only $\mathcal{O}(\text{scan}(n))$ time and I/Os.

One can hope to further generalise the above results from [31, 71] to cover more than one variable being moved at a time. Furthermore, using other meta information, such as the 1-level cut [175] (Chapter 5) just below the level of variable $x_i$, $C_{1:f[i]}$, one can further tighten these bounds. For example, one can derive the following relation between two adjacent levels.

**Lemma 10.1.** *The number of arcs in $f$ that skip across the level of $x_i$ is equal to $C_{1:f[i]} - 2 \cdot N_i$.*

*Proof.* Following the same argumentation as for Corollary 8 in Chapter 5, there are $C_{1:f[i]}$ arcs that go across the cut just below the level of $x_i$. Exactly $2 \cdot N_i$ of these arcs originate from the $N_i$ nodes at this level. □

**Corollary 10.2.** *Let $x_i$ be the variable that is adjacent and above $x_j$, the number of arcs to $x_j$ that do not originate from a node with variable $x_i$ is at most $C_{1:f[i]} - 2 \cdot N_i$.*

*Proof.* At most all arcs that skip over $x_i$ can lead to some BDD node with variable $x_j$. The bound then follows from Lemma 10.1. □

**Corollary 10.3.** *Let $x_i$ be the variable that is adjacent and above $x_j$, the number of arcs from $x_i$ that skips past $x_j$ is at least* $\max(0, C_{1:f[j]} - C_{1:f[i]} + 2 \cdot (N_i - N_j))$.

*Proof.* By Lemma 10.1, $C_{1:f[j]} - 2 \cdot N_j$ is the number of arcs that skip the level of $x_j$. By Corollary 10.2, at most $C_{1:f[i]} - 2 \cdot N_i$ of these arcs cannot originate from $x_i$. □

**Corollary 10.4.** *Let $x_i$ be the adjacent variable above $x_j$, the number of BDD nodes with $x_i$ that are independent of $x_j$ is at least* $\max(0, C_{1:f[j]} - C_{1:f[i]} + N_i - 2 \cdot N_j)$ .

*Proof.* By the pigeonhole principle, only $N_i$ arcs that originates at $x_i$ but skip $x_j$ can be distributed across the $N_i$ nodes at $x_i$ without making any node entirely skip $x_j$. Afterwards, each additional arc must make a BDD node independent of $x_j$. Decreasing the lower bound in Corollary 10.3 by $N_i$ provides the desired result. □

These results can potentially be used to tighten the above bounds from [31, 71]. But, the algorithms implemented in Chapter 5 only compute upper bounds on the 1-level cuts. Hence, very few of the Corollaries above are applicable in practice. Yet, the following theorem can also be applied in practice even if $C_{1:f[i]}$ is not computed exactly.

**Theorem 10.5.** *Consider two adjacent variables $x_i < x_j$. The size of the level for variable $x_j$ after swapping is at most $C_{1:f[i]} - N_i$.*

*Proof.* Each of the $N_i$ nodes with variable $x_i$ are replaced by one node with variable $x_j$ [94, 161]. Furthermore, Corollary 10.2 implies that at most $C_{1:f[i]} - 2 \cdot N_i$ of the original $x_j$ nodes are reachable and hence kept after the swap. □

One can further investigate other meta information that is relevant and whose computation can be piggy-backed onto the `Apply` and/or `Reduce` algorithms as in Chapters 5 and 6.

Finally, an unsafe expected size may in practice be a better guide for the variable search rather than the above safe bounds on the resulting BDD size [85]. For example, exchanging two variables will in most cases only result in $T = (1 \pm \frac{1}{2}) \cdot N$ [85].

## 10.4 Metaheuristics

As previously mentioned, finding an ordering, $\pi$, that minimises the size of $f$ is an NP-complete optimisation problem [29, 187]. Hence in practice, one has to resort to approximation algorithms and local search heuristics to find a $\pi$ that is a "good enough" local optimum. Following previous work on this area, we explore how to use the algorithms from Chapter 9 as part of metaheuristics.

In the light of a BDD being so large that it has to be stored in external memory, it is vital to have a guarantee on the local search algorithm's space complexity. In particular, our procedures should be close to linear in memory usage.

### Simulated Annealing

Inspired by statistical mechanics from material science, simulated annealing [51, 108] (SA) aims to find the (near) optimal solution by an analogy to material properties. The temperature, $\mathcal{T}$, of the system is raised to its melting point to then slowly lower its temperature back down to freezing. Doing so, the material settles into its lowest energy state.

Bollig, Löbbing and Wegener have investigated in [30, 31] the use of simulated annealing in the context of BDDs. Here, the system's atoms are the BDD variables and the system's state is a particular variable ordering $\pi$ [30]. Each ordering, $\pi$, has a certain set of *neighbouring* states, $\pi'$. The following repeats until no improvements have been made or $\mathcal{T}$ reaches the minimum temperature:

- Pick at random a single neighbour, $\pi'$, of the current ordering $\pi$. In [30, 31], $\pi'$ is obtained by means of a single *Jump-Up*, *Jump-Down* or *Exchange* of variables. The current variable ordering $\pi$ is replaced by $\pi'$ if either of the following two cases apply:
  - The BDD for $f$ with $\pi'$ is smaller than $f$ with $\pi$ [51, 108].
  - If $f$ grows larger with $\pi'$, then it is probabilistically accepted based on a Boltzmann factor which exponentially relates the size of $f$ with $\pi$ and with $\pi'$ to the temperature $\mathcal{T}$ [51, 108].

Intuitively, the search space is explored widely at higher temperatures, thereby escaping local minima. At lower temperatures, the best solution is identified among the local neighbourhood.

If a $\pi'$ is only accepted if its size is some $\Theta(N)$ BDD nodes, then one can guarantee a linear space usage: the algorithm holds onto at most 4 $\mathcal{O}(N)$ sized BDDs and an $\mathcal{O}(n)$ sized variable orderings at once. Hence, it uses $\mathcal{O}(N)$ space. The constant hidden in the $\mathcal{O}$-notation is usable in practice. This is, of course, at the cost of being stuck in the local neighbourhood surrounding the original ordering.

In [30, 31], each proposed neighbour, $\pi'$, is a single jump or exchange of variables. In CUDD [182], this is implemented by repeatedly swapping adjacent variable. This allows it to enumerate multiple intermediate orderings for each jump and exchange. The best of all these intermediate states are picked to be the $\pi'$. By including the changes to these algorithms in Section 10.2, the same can also be done for an external memory SA. Similar to our case, it aborts any computation of a $\pi'$ that grows too large. Hence, CUDD is also stuck within the same neighbourhood surrounding the original ordering.

### Genetic Algorithm

A genetic algorithm [92] (GA) works by considering a population of $\mathcal{P}$ variable orderings (chromosomes) and guiding them towards better solutions using natural selection. The BDD-based GA in CUDD [182] is based on the work of Drechsler, Becker, and Göckel [72] which starts out as follows:

- *Initialisation*: A population of $\mathcal{P}$ random variable orderings (chromosomes), $\pi_1, \pi_2, \ldots, \pi_{\mathcal{P}}$ are initialised uniformly at random [72, 92, 142]. Each variable ordering is associated with some *fitness* [92]. In this case, $\pi_i$'s fitness is the BDD size it induces [72].

  In CUDD, $\pi_1$ is initialised as the original ordering, $\pi$, of $f$ whereas $\pi_2$ is its reverse. Doing so preserves previous knowledge about a good ordering already present in $\pi$. All others are picked uniformly at random.

In CUDD $\mathcal{P} = \min(3 \cdot n, 120)$, as suggested by Drechsler, Becker, and Göckel [72]. The initial population is obtained by means of repeated variable swaps. If the intermediate BDD explodes beyond a certain threshold, then it is discarded. Similar to SA, this restricts the search to be within a local neighbourhood of the initial ordering. Using the algorithms from Chapter 9 to I/O-efficiently generate the initial population will suffer the same restriction.

The following three steps are then repeated until no improvements have been made for some time or until a computational limit has been reached.

- *Crossover*: Each variable ordering $\pi'_i$ (offspring) of the next generation is created based on two variable orderings, $\pi_j$ and $\pi_k$, (parents) from the current generation [92, Chapter 6.2] [72]. For each offspring, the likelihood for a current ordering $\pi_j$ to become its parent is proportional to $\pi_j$'s fitness [72].

  Both [72] and CUDD use *partially matched crossover* which creates two new variable orderings from each pair of parents. We refer to [72] for more details and to [9, 142] for other possible crossover operations.

  In CUDD, the maximal number of crossovers is by default $\min(3 \cdot n, 60)$. Furthermore, it computes the resulting BDD with repeated variable swaps. Hence, similar to generating the initial population, CUDD is restricted to crossovers that are reachable by means of swaps [166]. An I/O-efficient GA with the algorithms in Chapter 9 would be restricted in a similar way.

- *Mutation*: Each offspring, $\pi'_i$, may experience some small *mutation* to its ordering [92, Chapter 6.4] [72]. In [72], this is achieved by exchanging one or two pairs of variables; these variables are not necessarily adjacent.

  The likelihood for a mutation should be low to not turn it into a random search [92, Chapter 6.4]. Drechsler, Becker, and Göckel suggest the likelihood to experience a mutation is set to 15% [72].

  Mutations are not included in CUDD's GA.

- *Selection*: Finally, $\mathcal{P}$ of the next generation's orderings are (probabilistically) *selected* depending on their fitness to survive [72]. See [142] for a list of potential selection operations.

Intuitively, mutation provides the ability to do hill-climbing, selection introduces a notion of competition between the solutions, whereas crossover allows for solutions to cooperate [145] [92, Chapter 6.4].

To further improve the result, Drechsler, Becker, and Göckel suggest to apply sifting [161] on each initial $\pi_i$ to find $\mathcal{P}$ local minima [72]. Furthermore, sifting should be applied on the final $\mathcal{P}$ variable orderings after termination to reach the best (local) minima [72]. In the case of Adiar, either of the sifting algorithms in Section 10.5 can be used to do so.

This approach requires, next to the BDDs, only $\mathcal{O}(\mathcal{P} \cdot n)$ space to store all the $\mathcal{P}$ $n$-sized variable orderings. Hence, one should pick $\mathcal{P}$ such that $\mathcal{P} \cdot n$ fits in $\mathcal{O}(N)$ space; even better is if it fits in $\mathcal{O}(M)$ space. The algorithm in Section 9.3, together with the optimisations in Section 10.2, can be used to compute each candidate ordering's fitness from the original BDD of size $N$. Hence, the BDD associated with each $\pi_i$ does not need to be stored. In this case, with proper choice of $\mathcal{P}$, the algorithm uses $\mathcal{O}(N)$ space. Yet, it may be expensive to compute the quality of each $\pi'_i$ by reordering the original BDD $f$ rather than either its parents. Furthermore, computing $\pi'_i$ from the initial ordering with the algorithm in Section 9.3 may incur an explosion whereas with either its parents one does not. Hence, one may want to store the $\mathcal{P}$ (or fewer) BDDs with $\Theta(N)$ or fewer nodes.

To improve the running time, Günther and Drechsler [85] suggest to store the BDD sizes of the variable orderings in prior generations. Depending on the number of variable orderings stored, this may break the linear amount of space used. To solve this, one can bound the size of the memoisation table to be $\mathcal{O}(N/n)$ (or even better to be $\mathcal{O}(M/n)$); hash collisions are resolved by merely overwriting the old value. Furthermore, they suggest to improve the running time when possible by approximating rather than computing the BDD sizes [85]. Here, the bounds from Section 10.3 can be used.

To address the crossover step generating a lot of bad variable orderings that need to be evaluated, Thornton et al. [191] propose to only use order-reversal* and sifting for the crossover step. If no new variable order is obtained, a mutation is applied instead. In practice, they were able to obtain variable orderings of the same quality as in [72] with a much smaller population size $\mathcal{P}$.

### Memetic Algorithm

Drechsler, Becker, and Göckel only apply sifting [161] to the initial and final variable orderings in their GA [72]. They do so to avoid getting stuck in local minima [72]. But, sifting can be applied to intermediate variable orderings if care is taken [9]. Going back to the analogy of natural selection, this is akin to the Lamarckian principle of each individual also gaining knowledge and passing it down to its offspring [96]. Alternatively, each variable ordering can be thought of as meme [63] rather than a gene. These intermediate local search steps turns the GA into a memetic algorithm [145, 147] (MA), also called the *cooperative-competitive* approach. Recontextualising the original MA by Moscato and Norman [145, 147] in the context of BDD orderings, the algorithm again considers a population of $\mathcal{P}$ variable orderings (memes) and

*Reversing the order seems akin to the *inversion* step in [92, Chapter 6.3]. Hence, it may be worth looking not only looking into reversing the entire ordering but also only selected intervals of variables.

repeatedly applies the following three phases until the solution quality has stabilised or time has run out:

- *Challenge*: Similar to a *selection*, one meme, $\pi_i$, challenges another, $\pi_j$. If $\pi_i$ is better than $\pi_j$ then $\pi_j$ is discarded and instead replaced by an exact copy of $\pi_i$ [145]. Similar to simulated annealing, a challenge may be won by an inferior $\pi_i$ depending on a global temperature of the entire system, $\mathcal{T}$ [145, 147].

- *Cooperation*: Similar to a *crossover*, one meme, $\pi_i$, may propose to cooperate with $\pi_j$ by copying one (or more) subsequence(s) of $\pi_i$ into $\pi_j$ [145, 147]; the previous locations of the variables in $\pi_j$ are forgotten. Similar to challenges, a cooperation is accepted depending on its merit and the system's temperature [145, 147].

  How $\pi_i$ is best crossed over with $\pi_j$ has been briefly investigated in [9]. Furthermore, we can make use of the function's profile for $\pi_i$ and $\pi_j$ to guide which subsequences are shared with $\pi_j$. A subsequence in which the BDD size does not grow exponentially indicates the variables are related and their ordering is worth sharing.

- *Local Search*: After each cooperation, the resulting new ordering $\pi_j$ may be much worse due to having moved one or more variables very far from their original position into the copied subsequence [145]. To repair this, another local search algorithm is applied to $\pi_j$ [145, 147]. This could be the sifting algorithms in Section 10.5, .

The proposals for a challenge, resp. a cooperation, is restricted to only happen along a (linearly) few number of channels between different memes [145]. Intuitively, one can imagine a group of people working at each their own table, copying of solutions from the person on their left and right (challenges), and passing hints to the person in front of and behind them (cooperation).

If the diversity of the $\mathcal{P}$ is below some threshold, then a consensus has been reached around a local minima [145, 147]. In this case, one can stop and return the best of the $\mathcal{P}$ solutions [147]. Alternatively, one can *reheat* the system by mutation and quickly increasing $\mathcal{T}$. This reintroduce enough variations to escape the local minima [145].

Similar to GA, this approach requires $\mathcal{O}(N)$ or $\mathcal{O}(\mathcal{P} \cdot N)$ space depending on whether the intermediate BDDs are stored or not. Furthermore, as also suggested for GA in [85], one can hope to improve the MA's running time with memoisation and the bounds in Section 10.3.

### Swarm Intelligence

The work in [30, 31, 72, 85] is from the late 1990's and based on metaheuristics from the 1970's and 1980's. Since then other population-based metaheuristics have been proposed. These recent metaheuristics have proven to be effective relative to GA. This

is both in terms of the running time and the solution quality. One reason is that these newer approaches are better at retaining knowledge across generations [76].

We now highlight three such techniques which all are based on the emerging intelligence of a swarm of simple agents.

### Particle Swarm Optimisation

Mitra and Chattopadhyay [143] proposed to use particle swarm optimisation [76, 107] (PSO) to search for an improved BDD size. This technique takes inspiration from the simulation of bird flocks and fish schools [76, 107]. In particular, consider $\mathcal{P}$ particles moving through a $\mathcal{D}$-dimensional space where each point is associated with some *fitness*. Each particle has its own position and velocity and the memory of the best solution it has encountered itself (*lbest*). Furthermore, they also know of the best solution discovered by the entire swarm (*gbest*). For each time-step, the velocity and positions are updated based on *lbest*, *gbest* and a random perturbation. The search terminates if a maximum number of time-steps has been reached or if *gbest* has not improved for a certain number of steps. For more details, see [76, 107].

For BDDs, Mitra and Chattopadhyay let a swarm of size $\mathcal{P} \leq 100$ explore a $\mathcal{D} = n$ dimensional space [143]. Each $n$-dimensional point is associated with a BDD variable ordering by considering each coordinate's rank, i.e. its index in the $n$-dimensional vector after being sorted [143]. The position's fitness is proportional to the BDD's size [143]. The experimental results in [143] indicate this approach produces better variable orderings than any heuristic implemented in CUDD [182], i.e. SA, GA, and sifting [161].

Next to the BDDs, this approach requires $\mathcal{O}(\mathcal{P} \cdot n)$ space to store the $\mathcal{P}$ $n$-dimensional positions and velocities. Hence similar to GA, one should pick $\mathcal{P}$ based on $n$, $N$, and maybe $M$. Also similar to GA and MA, this approach will use $\mathcal{O}(N)$ or $\mathcal{O}(\mathcal{P} \cdot N)$ additional space to store the BDDs throughout the search.

### Spider Monkey Optimisation

Siddiqui, Tariq Beg, and Naseem Ahmad [171] follow up on [143] by looking into another swarm intelligence approach, spider monkey optimisation [17] (SMO), based on the foraging behaviour and group dynamics of spider monkeys [171]. This introduces a hierarchical and territory-respecting group-dynamic to the particles. See [17, 171] for an exact description.

Again, $\mathcal{O}(\mathcal{P} \cdot n)$ space is required to store the $n$-dimensional positions of each solution. Furthermore, the trade-off between only using $\mathcal{O}(N)$ or $\mathcal{O}(\mathcal{P} \cdot N)$ space still applies. Yet, one can investigate whether to only store one BDD for each subgroup rather than each individual; the intuition being, that the individual of each subgroup are close together and hence their solutions are quite similar. This may be a useful compromise in practice.

**Ant Colony Optimisation:**

Ant Colony Optimisation [69, 70] (ACO) is a metaheuristic that is inspired by the use of pheromone trails of ants [69]. Consider $\mathcal{P}$ ants that explore paths through a graph. Each path they pick is associated with some fitness and pheromones are accordingly left on it. The presence of pheromones affects the probability that a later ant retraverses (parts of) this path. As time progresses, pheromones evaporate. A good path is preserved by ants retraversing it and reinforcing the pheromone trail. See [69] for a more detailed description. To the best of our knowledge, no one has yet investigated using ACO to derive better BDD variable orderings.

For applying ACO to BDD orderings, note that a variable ordering $\pi$ can be thought of as a *hamiltonian path* [105] through a fully connected graph with $n$ nodes, one node for each variable; the variable in the $i$'th node on said path corresponds to $\pi(x_i)$. Initially pheromones can be placed on $n-1$ edges according to the original variable ordering in $f$. To pick the root variable, one can add an additional node with an edge to each of the $n$ variables. The initial strength of pheromones on these root edges could reflect the variable's rank in the original ordering. If the variable blocks from [134] are available, then the pheromones on the root edges can be normalised such that initially picking either block as the root is equally likely.

Each ant chooses an edge $(i, j)$ from node $i$ to $j$ based on the pheromones, randomness, and the distance between $i$ and $j$ [69]. It is not obvious how such a distance between two variables $x_i$ and $x_j$ should be defined for BDDs. On the one hand, such a distance measure should capture the dependency between the two variables. Yet, it should also be independent of any variable order. The probability based metric in [191] could be part of such a distance measure.

Similar to PSO and SMO, knowledge is preserved across generations. Yet, unlike the positions in $n$-dimensional space from [143, 171], there is a much more intuitive correspondence between variable orderings and the pheromone trails. For example, one should expect that pheromones will over time be concentrated on the edges between highly-related variables.

Unlike GA, MA, PSO, or SMO this approach does not depend on maintaining individuals in a population of size $\mathcal{P}$; instead the number of variable orderings derived between updating the pheromone trails implicitly decides the size of $\mathcal{P}$. Hence, the memory requirement of ACO is independent of $\mathcal{P}$. Instead, the deposited pheromones need to be stored using $\mathcal{O}(n^2)$ space. This restricts this technique to only be applicable if $n$ is bounded by $\mathcal{O}(\sqrt{N})$ or $\mathcal{O}(\sqrt{M})$. In practice, $n$ can be in the thousands and still fit within 1 GiB of internal memory. Furthermore, as the solution converges, the previous best ordering will be very close to the ones to-be evaluated. Hence, only the previous best BDD of size at most $N$ needs to be stored.

## 10.5 Sifting

The most succesful dynamic variable reordering algorithm is Rudell's sifting algorithm [161]: variables are moved one at a time up and down through the BDD to find its

optimal position (assuming all other variables are fixed). Each variable is moved by means of successive adjacent variable swaps. This has been a succesful approach for conventional BDD implementations, since they can swap two adjacent levels efficiently. This approach has since been extensively improved, e.g. [71, 134, 135, 151, 152] (see also Section 10.3).

### Rudell's Sifting

The original sifting algorithm by Rudell [161] can be recreated using the algorithms from Sections 9.4 and 10.2 as follows:

For each variable $x_i \in \{x_1, x_2, \ldots, x_n\}$ (in some order):

- Evaluate all $n$ positions of $x_i$ in $f$ by means of the extended *Jump-Up* and a *Jump-Down* sweeps from Section 10.2.
  This requires $\mathcal{O}(\text{sort}(\sum_{j=1}^{n} C_{2:f[j]}))$ time and I/Os.
- If a level $\ell$ has been identified that decreases the size of $f$, move $x_i$ up or down to level $\ell$ using the respective algorithm from Section 9.4. All the levels in-between are changed monotonically.
  This uses another $\mathcal{O}(\text{sort}(N))$ time and I/Os.

In [161], the variables are processed in order of the size of their respective levels in $f$. The idea of blocks from [134] and the lower bounds from [71] can be repurposed by limiting the number of levels that are processed by each iteration's initial two *Jump-Down* and *Jump-Up* sweeps.

For each of the $n$ variables, the above algorithm uses $\mathcal{O}(\text{sort}(\sum_{j=1}^{n} C_{2:f[j]}) + \text{sort}(N))$ time and I/Os, i.e. somewhere between $\mathcal{O}(\text{sort}(N))$ and $\mathcal{O}(\text{sort}(n \cdot N))$ time and I/Os depending on the number of arcs that skip past one or more levels. Hence, in total the above algorithm may use up to $\mathcal{O}(n \cdot \text{sort}(n \cdot N))$ time and I/Os. This time complexity is about a factor of $n/B$ slower than Rudell's original algorithm [161]. Yet in practice, this I/O-efficient version is probably closer to the original algorithm's performance than this worst-case bound suggests.

The above algorithm holds only onto a single copy of $f$ whose size is decreased from its original $N$ BDD nodes. Furthermore, each sweep above uses only $\mathcal{O}(N)$ space and creates a new BDD of size $N$ or smaller. Hence, the above algorithm uses only $\mathcal{O}(N)$ space.

### Parallel Sifting

In Section 9.5, we also described how to permute the levels of a BDD in $\mathcal{O}(\text{sort}(N))$ time and I/Os if only adjacent variables need to be swapped. Furthermore, in Section 10.2, we described how to evaluate the outcome of all $n-1$ swaps at once and at a lower cost in practice. This can be used as follows:

Repeat until no additional swaps have been identified:

- Compute the change in $f$'s size for all $n-1$ swaps, $\delta_{1,2}, \delta_{2,1}, \ldots, \delta_{n-1,n}$, in $\mathcal{O}(\text{sort}(N))$ time and I/Os.
- In $\mathcal{O}(\text{sort}(N))$ I/Os, pick and apply a disjoint subset of the computed variable swaps.

That is, instead of moving variables one-by-one across all levels with multiple swaps as in [161], which would cost an entire $\mathcal{O}(n^2 \cdot \text{sort}(N))$ time and I/Os, we move multiple variables a little bit at a potentially much lower cost. Hence, we dub this *parallel sifting*. This also has the potential benefit of variables being able to "communicate" and "react" to each other's moves between each iteration.

Since only one copy of $f$ is present at any time, $n-1 < N$, and each sweep only requires $\mathcal{O}(N)$ space, the entire procedure uses only $\mathcal{O}(N)$ space with only a small constant hidden inside of the $\mathcal{O}$ notation.

One can extend the swap-evaluating sweep with the logic from [152] to also identify adjacent symmetric variables. The idea of confining variables inside of $\beta$ *blocks* [134] is also easily applied by not including the $\beta - 1$ swaps between the block boundaries in each iteration's first sweep. Similarly, the lower bounds of [71] can be applied to not compute the change in BDD size for trivially bad swaps.

The description above is on purpose quite vague about how to pick the subset of variable swaps to apply each iteration. If one always greedily picks the swaps that decrease $f$'s size the most then the BDD size would be monotonically decreasing towards a local minimum. This would make it quite similar to the 2-window approach in [77] and hence it is expected to converge quite fast to a solution at the cost of the solution's quality [77]. Furthermore, the steps towards such a local minimum would not include a swap that undoes a previous one. Hence, such a greedy approach must terminate within $n$ iterations, making it use $\mathcal{O}(n \cdot \text{sort}(N))$ time and I/Os. Alternatively, similar to randomised rounding [158], one can pick between two overlapping swaps probabilistically relative to their impact on BDD size. Inspired by SA, this probability can also depend on a global temperature $\mathcal{T}$. This can potentially introduce enough hill-climbing into the search to improve the quality of the final solution.

Similar to the creation of $k$-moves in the Lin-Kernighan [118] heuristic for the travelling salesman problem, the above loop builds multiple (possibly interacting) chains of swaps. To escape local minima, one can, also similar to [118], at certain intervals apply a permutation. Such a permutation would require the algorithms from Section 9.4 or even the algorithm from Section 9.3.

## 10.6 Related Work

There has been extensive research into algorithms and heuristics to derive a better BDD variable order. A lot of the related work, has already been covered as part of Sections 10.3 to 10.5. We now cover some additional observations that were previously not needed to provide context.

**Restrict-guided Rebuilding:** The aim of the output-sensitive algorithms in [21, 133, 166, 186] was to address that swap-based approaches, e.g. sifting, get stuck in a local minimum [21, 133, 166].

Bern, Meinel, and Slobodová used their algorithm to decrease memory usage during symbolic simulation of circuits [20, 21]. After having identified on-the-fly which circuit gates create BDDs beyond the current memory limit, they analyse these gates anew to derive a new ordering. If this new ordering decreases the memory usage, then it is kept. Otherwise, the memory limit is doubled. Though this is an interesting proof of concept for our manual reordering algorithms in Chapter 9 (possibly with the improvements in Section 10.2), our aim in this chapter has been the derivation of the variable ordering directly from the BDD. Furthermore, the non-linear space usage of our algorithm in Section 9.3 bars us from properly reapplying their ideas.

In contrast to the swap-based derivation of offspring as part of CUDD's genetic algorithm [182], Savický and Wegener note that their output-sensitive algorithm can also be used to efficiently compute the offspring [166]. In particular, a GA is not restricted in its search to the BDDs that are reachable with swaps without exploding when using their algorithm [166]. More generally, their algorithm and the ones in [21, 133, 186] can be used for all of the metaheuristics in Section 10.4 as space-efficient oracles to evaluate each variable ordering without any restrictions on the neighbourhood. In our setting where BDDs are larger than the machine's internal memory, there is an even smaller tolerance for a intermediate BDD size exploding. Hence it is highly relevant to put future research efforts towards an I/O-efficient (and practically usable) algorithm with linear space guarantees.

**Heuristical Order Derivation:** Similar to Section 10.3, one could hope to be able to infer relevant information about how to improve a BDD's ordering based on its meta information or traversing its BDD graph.

For example, Thornton, Williams, Drechsler, Drechsler, and Wessels [191] noticed there is a relationship between a probability based metric on the given BDD and its optimal variable ordering [191]. This metric is computable in linear time with the algorithm in [141] [191]. Hence, they propose a dynamic variable reordering algorithm by reconstructing the BDD $f$ based on the probability metric. This reconstruction includes multiple non-deterministic choices. Hence, they further suggest to exchange variables based on the given heuristic to further decrease the BDDs size.

The linear time analysis of the BDD in [141, 191] can be translated into a sweep that uses $\mathcal{O}(\text{sort}(N))$ time and I/Os. The algorithms from Sections 9.3, 9.4 and 10.2 can be used to both reconstruct the BDD based on the metric and also to further improve its size afterwards.

**Metaheuristics:** Section 10.4 already compares the application of our I/O-efficient algorithms from Chapter 9 to the implementation of the metaheuristics in CUDD [182]. In CUDD, all changes to the variable ordering during the search is implemented by means of swaps. If the intermediate BDD size explodes beyond some threshold,

CUDD aborts evaluating that part of the search space. Computing some orderings, $\pi'$, from another one, $\pi$, may also result in an (intermediate) explosion if our I/O-efficient algorithms from Chapter 9 are used instead (with the optimisations in Section 10.2). Yet, the algorithms from Chapter 9 do not incur any BDD explosion if they are used to recreate the very swap steps by CUDD. That is, an I/O-efficient dynamic reordering procedure in Adiar would not be more restricted in its search than the ones in CUDD.

**Sifting:** As also described in Section 10.5, Rudell's sifting algorithm [161] works by moving variables one-by-one through the entire BDD to their seemingly optimal position. Since this approach moves one variable at a time – leaving all other variable's position fixed – its search-space is restricted to a small neighbourhood of the original variable ordering. Furthermore, the optimal variable orderings might be unreachable with adjacent variable swaps alone without the BDD exploding [21, 133, 166]. Hence, the sifting algorithm may become stuck in a local minimum [21]. Yet, compared to the metaheuristics in Section 10.4, sifting finds a new variable ordering much faster [182]. Yet, this is at the cost of the solution's quality [31, 72, 143].

Our sifting algorithms in Section 10.5 follows in the same vein by being designed to converge quite fast to a local minimum. The faithful translation of Rudell's algorithm exposes its incompatibility with Adiar's BDD representation; whereas the original algorithm in [161] runs in $\mathcal{O}(n \cdot N)$ time, the direct translation requires $\mathcal{O}(n \cdot \text{sort}(n \cdot N))$ time. Yet in practice, as also argued in Section 10.5, we expect the algorithms performance is much closer to the desired $\mathcal{O}(n \cdot \text{sort}(N))$ time complexity. To further address this, we also propose *parallel sifting* in Section 10.5 to decrease the amount of wasted work spent on merely copying non-swapped levels as part of each $\mathcal{O}(\text{sort}(N))$ sweep. If swaps are chosen greedily, then this requires at worst $\mathcal{O}(n \cdot \text{sort}(N))$ time and I/Os as desired. We hope to use the algorithms in Sections 9.3 and 9.4 to compensate for an otherwise unsurmountable hillclimb. In general, we see lots of potential for using (parallel) sifting as a subprocedure to speed up the metaheuristics in Section 10.4.

**CAL:** The external memory BDD package CAL [165] (based on [13, 149]) does support the sifting algorithm [159]. Yet, similar to Adiar's unique identifiers, CAL's pointers also include both a BDD nodes' level and its index [165]. Hence, swapping two adjacent levels cannot be a local operation in CAL [159]. Ranjan et al. [159] proposed the following two ways to solve this while also preserving the memory layout of CAL:

1. The update of the swapped BDD nodes' parents' is deferred until later by leaving some BDD nodes behind with a *forwarding address* [159]. Yet, their experiments indicate this introduces too costly an overhead in practice both in running time and memory usage [159].

2. Hence, they instead proposed to convert the BDDs from their breadth-first representation into the conventional depth-first representation, perform the

conventional reordering algorithms, and then convert the result back to the breadth-first representation [159]. This drops the BDD nodes' levels from the pointers during the reordering procedure. Yet, the logic to swap two adjacent levels needs access to the entire two levels that need to be swapped [161]. Hence, similar to CAL's other algorithms, its reordering is limited to BDDs whose levels fit into memory [12].

Our proposed approach in this chapter, on the other hand, applies to BDDs of all shapes and sizes.

**Heuristic Learning:** In Section 10.4, we have only focused on metaheuristics to directly search for a smaller BDD. Göckel, Drechsler, and Becker investigated in [73, 83] the ability to learn heuristics for variable (re)ordering based on a *training set*. Here, they used a genetic algorithm to derive a sequence of reordering operations, *sifting*, *greedy sifting*, and *order reversal*, that both minimise the running time and the BDD's size. This further emphasises the need to further improve the algorithm in Section 9.3 in the case the variable order is inverted.

One could also imagine the use of more recent machine learning techniques to derive heuristics that guide the reordering algorithms in Chapter 9 and this chapter. Such a heuristic can use different meta information, e.g. the variable profile (Section 10.3) and the adjacent swap profile (Section 10.2). To the best of our knowledge, no work has been done in this direction.

## 10.7 Conclusion

We have shown multiple avenues for how to use the algorithms from Chapter 9 to recreate the many different dynamic variable reordering algorithms in the external memory setting. In each case, we have strived towards guaranteeing a linear amount of space is used such that they also are usable in practice.

The metaheuristics in Section 10.4 and the sifting algorithms in Section 10.5 are each other's opposites: whereas the prior produces good variable orders, the latter terminates quickly. Yet, they are not incompatible. On the contrary, as already briefly mentioned for genetic and memetic algorithms in Section 10.4, (parallel) sifting can be used as part of a metaheuristic to accelerate its search. For example, (parallel) sifting can be used in simulated annealing at lower temperatures. For ant colony simulation, one can run (parallel) sifting on each candidate variable ordering $\pi$ to then treat the result of each sifting iteration as a separate ordering whose fitness is also left as an additional pheromone trail.

One interesting property of most metaheuristics in Section 10.4 is that they are quite trivially parallelisable. If internal memory is managed properly, then the BDD operations from Chapters 3 and 7 can also be made thread-safe. This can be used in the algorithms of Sections 10.4 and 10.5 as follows:

- One can concurrently evaluate up to $P$ variable orderings in each metaheuristic, where $P$ is the number of processors in the machine. Hence, the population based metaheuristics may want to set the size of the population $\mathcal{P}$ to some multiple of $P$.

  Furthermore, since some of the algorithms from Chapter 9 starts with transposing the given BDD $f$, we can save computation time and space by sharing a single read-only copy of $f$ that already is semi-transposed.

- During parallel sifting, the algorithm may have to make a choice between swapping one or more variables either up or down. In this case, the algorithm may run either choice in parallel. Such a split can recursively be repeated up to $\mathcal{O}(\log_2(P))$ times.

- Instead of optimising a single BDD $f$, one can optimise the collective size of up to $P$ BDDs. But, even better would be to use the algorithms from Chapter 9 on an entire shared forest of BDDs in a single sweep.

## Acknowledgement

Thanks to Dubslaff and Wirtz for referring to [175] in [74] which sparked the ideas for Lemma 10.1, Corollaries 10.2 to 10.4, and Theorem 10.5. Furthermore, thanks to Riko Jacob for suggesting to further parallelise the metaheuristics by sharing the transposed BDD between threads. Finally, thanks to Sebastian Lague for sharing the idea of Ant Colony Simulation as part of his “Coding Adventures” video series.

# Chapter 11

# Unique Node Table

## 11.1 Introduction

The performance of the Adiar BDD package has come a long way since its initial version in Chapter 3. This is in particular thanks to the optimisations in Chapters 5 and 6. But, as Chapters 5, 7 and 8 quite clearly show, while we have been able to achieve an acceptable performance for moderately sized BDDs and beyond, Adiar's performance is too slow for small instances to be useful in many practical applications.

This performance issue could, at least in part, be due to the fact that Adiar is implemented on top of the TPIE [144, 196] library [160, 163]. If the external memory data structures in STXXL [64] were used instead, then the running time may improve for BDDs of all sizes [163]. To address performance on the small scale, one may want to ditch support for external memory and instead switch to algorithms and data structures that are highly optimised for internal memory. For example, one could try to use a sequence heap [162] and the `IPS`$^4$`o` [14] parallel sorting algorithm. Furthermore, the sorting predicates in Chapter 3 are merely dependent on integer keys. Hence, one could also investigate the use of radix sort [90], its optimisations [62, 129, 164, 197], and the radix heap [3]. One can use the C++ standard library as a baseline to evaluate each backend.

In this chapter, we will instead approach this problem with the assumption that we have reached the limit as to what is achievable with time-forward processing. If so, it is worth investigating whether Adiar's I/O-efficient algorithms can work in tandem with the conventional depth-first approach.

### Contributions

We show how to use levelised random access from Chapter 6 to process BDDs that are stored in the unique node table [33, 141] from Section 1.2. Conversely, we also show how Adiar's `Reduce` algorithm can output the final BDD back into the unique node table. That is, the conversion between the two algorithmic paradigms can be piggy-backed onto Adiar's I/O-efficient algorithms from Chapter 3.

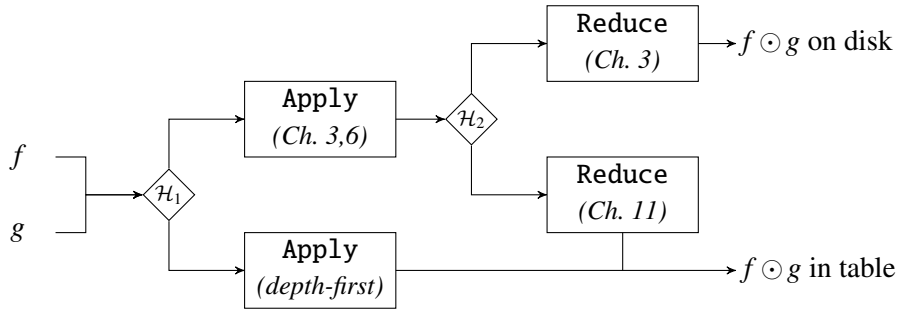



Figure 11.1: The Apply–Reduce pipeline extended to use depth-first algorithms rather than time-forward processing whenever possible. The two input BDDs, $f$ and $g$, are either stored on the disk or in the unique node table.

As shown in Fig. 11.1, this can be used to process to BDDs $f$ and $g$ that are stored on the disk and/or in the table. The conventional depth-first `Apply` algorithm is used if both $f$ and $g$ are stored in the unique node table and some heuristic inside of decision $\mathcal{H}_1$ can guarantee that the result, $f \odot g$, fits into the node table. The time-forward processing `Apply` from Chapter 3 is used otherwise. In this case, levelised random access from Chapter 6 is used on BDDs on the disk that are narrow and on BDDs that are stored in the table. Decision $\mathcal{H}_2$ provides a second chance to output the final result back into the unique node table: if the unreduced OBDD can fit into the table, the altered `Reduce` algorithm is used. Otherwise, the original `Reduce` from Chapter 3 is used to create a ROBDD on the disk.

## 11.2 To the Disk and Back Again

Key to the algorithms in Chapter 3 is that the conventional use of *pointers* is replaced with *unique identifiers*. Each BDD node is uniquely identified by a tuple $(\ell, \texttt{id})$ where $\ell \in \mathbb{N}$ identifies the level of the BDD node and $\texttt{id} \in \mathbb{N}$ is the level identifier. We made great efforts in Chapters 3, 5 and 7 to emphasize that the level identifier should *not* be mistaken for an index. Rather, it is merely an integer which reflects when the BDD node will occur in the input stream, relative to the others.

Yet, as the levelised random access optimisation in Chapter 6 shows, the `id` can in some cases be used as an index. In particular, we notice that the `Reduce` algorithm in Chapter 3 outputs the $N_\ell$ BDD nodes on level $\ell$ in reverse with identifiers $N_\ell + 1, N_\ell + 2, \ldots, \texttt{max\_id}$. Hence, after having loaded the entire level into internal memory, the index for the BDD node with the level identifier `id` is $N_\ell - (\texttt{max\_id} + 1 - \texttt{id})$.

In this chapter, we show how to use the `id` as an index for BDD nodes that are stored in a unique node table. We assume that this table is index-based, i.e. each BDD node refers its children by their index, `idx`, in the table. Each `idx` can be converted into a unique identifier by setting `id` to `idx` and $\ell$ to be the level of the node at index `idx`. Yet, doing so requires an additional memory access each time a unique identifier is converted. In practice, a 32-bit integer for `idx` is sufficient for our use-case. As also

noted in Chapter 3, 32-bits fit into the `id` of the 64-bit unique identifier currently used in Adiar. Hence, this index-based unique node table can also use unique identifiers directly at a small increase to the size of each BDD node. Doing so provides the level of each child without having to look them up.*

The following two subsections show how this allows us to transition to and from a conventional unique node table as part of the time-forward processing algorithms from Chapter 3.

### From the Node Table onto the Disk

For this presentation, we will only consider the `Apply` sweep in Chapter 3. Without loss of generality, assume that (at least) the second input, $g$, is a BDD that is stored in the unique node table. The algorithm from Chapter 3 can be used as-is if levelised random access from Chapter 6 is used to access the BDD nodes of $g$ within the unique node table.

As a result, the `Apply` algorithm can process two BDDs from a unique node table and produce the semi-transposed OBDD on the disk that is to-be processed as part of a succeeding `Reduce` sweep. Furthermore, it can combine a BDD inside of the table with a large BDD on the disk.

**Levelised Priority Queue:** To use the levelised priority queue in Chapter 3 as part of this time-forward processing sweep, one will have to know the BDD's levels a priori. While this could be computed prior to each time-forward processing sweep, it might be beneficial for each BDD, $f$, in the node table to be accompanied by a second BDD which is a conjunction of all its variables, i.e. the cube $\bigwedge_{x_i \in f} x_i$.

**Levelised Cuts:** To use levelised cuts from Chapter 5, the recursively constructed BDDs also need to be accompanied by other information. At least, they need (an upper bound on) their size to apply Theorem 6 in Chapter 5.

### From the Disk into the Node Table

The `Reduce` sweep's input is a semi-transposed unreduced OBDD of size $N$ stored on the disk. Assume that the unique node table still has room for $N$ new BDD nodes. In this case, the modified `Reduce` sweep in Fig. 11.2 feeds its output directly into the unique node table.

In the original `Reduce` sweep (Fig. 9 in Chapter 3), duplicate nodes are resolved by means of two sorting steps. In Fig. 11.2 this is done instead via the unique node table, i.e. the use of `make_node` on line 11 (see Fig. 1.5 in Chapter 1 for more details on this function). Doing so drastically simplifies the algorithm: the order in which the priority queue, $Q_{red}$, provides the BDD nodes matches the order in which the parent

*Interestingly, if unique identifiers are used, then one may as well drop a BDD node's level. Instead, each node can be reused across multiple levels since its parents already has assigned it its level. This is akin to [189, 190]. For example, a single BDD node could then be used to represent $x_i$ for any $x_i$.

```
1  Reduce(F_internal, F_leaf)
2    Q_red ← ∅
3    while Q_red ≠ ∅ ∨ F_leaf.has_next() do
4      j ← max(Q_red.top().source.label, F_leaf.peek().source.label)
5
6      while Q_red.top().source.label = j do
7        e_high ← PopMax(Q_red, F_leaf)
8        e_low ← PopMax(Q_red, F_leaf)
9
10       uid ← e_high.source
11       uid' ←make_node(uid.label, e_high.target, e_low.target)
12
13       while arcs from F_internal.peek() matches _→uid do
14         (s→uid) ← F_internal.next()
15         Q_red.push(s→uid')
16       od
17     od
18   od
```

Figure 11.2: The Reduce algorithm from Chapter 3 [179, 180] changed such that it outputs to a unique node table.

arcs are provided in $F_{internal}$. Hence, unlike the original `Reduce` sweep, every node can be processed individually rather than the entire level at once.

Since BDD nodes are processed one by one, the size of $Q_{red}$ is (similar to the top-down sweeps) upper bounded by the maximum 2-level cut (Chapter 5) rather than the 1-level cut.

## 11.3 Conclusion

We show that the time-forward processing algorithms in Chapter 3 can be run on BDDs from a unique node table and also feed the results back into the same table. That is, the conversion from one paradigm to the other can be done implicitly throughout the BDD computations. For the nested sweeping framework in Chapter 7, the initial transposition sweep can be used to move from the depth-first to the time-forwarding paradigm. Moving back is slightly more complicated, since it is only certain whether the output will fit into the unique node table when the last nested sweep has been resolved.

With Fig. 11.1 we covered how this can be used to transparently switch from one algorithmic paradigm to another. We suggest to implement this as follows:

1. Implement a unique node table[†] and add levelised random access to all of Adiar's time-forward processing algorithms.

   Adiar would then dedicate some amount of its internal memory, e.g. $M/4$, for the unique node table and the computation caches. The remaining internal

memory is left for the time-forward processing algorithms.

2. Implement depth-first versions of Adiar's algorithms[‡]. The `policies` that were introduced in Chapter 4 is of great benefit to this end: it is only necessary to implement a depth-first algorithm that manages the recursion requests provided by the `policy`.

   Whether to use time-forward processing or depth-first algorithms can at this point be left to the user.

3. Finally, allow Adiar to heuristically pick which paradigm to use ($\mathcal{H}_1$ in Fig. 11.1). In particular, the user may want choose between one of the following two options when processing BDDs in the unique node table:

   - Similar to Chapter 5, compute a priori a safe upper bound on the size of the (unreduced) output. Use time-forward processing if this bound exceeds the free space still left in the unique node table.
   - Try to use depth-first recursion. Restart the operation with time-forward processing if the depth-first approach exceeds the limits of the unique node table.

   The prior option tries to minimise waste of computation. The latter uses the unique node table optimistically in the hopes to maximize performance.

   One will also have to consider whether to free space in the node table via garbage collection or to leave it be and use time-forward processing instead. To this end, one ought to keep track of the number of dead and newly created nodes in the table since the previous garbage collection.

Long [121] showed that recursive BDD algorithms run two times faster if the BDD nodes are laid out in a depth-first order. This is because this ordering decreases the number of cache misses when the BDD is traversed with a depth-first algorithm [121]. Similarly, we can expect that the performance of the time-forward processing algorithm will depend on the order in which the BDD nodes are placed in the table. Future work could investigate how to lay out the table, and design garbage collection algorithms, to improve the locality of either or both algorithmic paradigms.

[†]When implementing this, one can start with recreating[‡] a simple unique node table such as the one in BuDDy [119]. One can later switch to the table proposed by Pastva and Henzinger [153]. Doing so will drastically improve performance [153]. Furthermore, this particular node table can allocate new nodes on-the-fly, meaning its initialisation time is $\mathcal{O}(1)$ time rather than $\mathcal{O}(M)$.

[‡] It may seem tempting to reuse another BDD package directly. But, this would introduce numerous complications due to mismatching sets of features. Furthermore, this would hinder any possibility to change the design of this table for our use-case. Finally, doing so would result in a substantial maintenance burden due to the lack of reusing code and logic via `policies` as in Chapter 4.

# Bibliography


[1] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. Feature model differences. In *Advanced Information Systems Engineering*, pages 629–645. Springer, 2012. doi:10.1007/978-3-642-31095-9_14. 13

[2] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535. 17

[3] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990. doi:10.1145/77600.77615. 213

[4] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978. doi:10.1109/TC.1978.1675141. 9, 173

[5] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: configuration analysis and verification of federated OpenFlow infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, pages 37–44. Association for Computing Machinery, 2010. doi:10.1145/1866898.1866905. 13, 31

[6] Ehab Al-Shaer, Will Marrero, Adel El-Atawy, and Khalid ElBadawi. Network configuration in a box: towards end-to-end verification of network reachability and security. In *International Conference on Network Protocols*, pages 123–132. IEEE, 2009. doi:10.1109/ICNP.2009.5339690. 13, 31

[7] Christoph Albrecht. IWLS 2005 benchmarks, 2005. URL `https://iwls.org/iwls2005/benchmarks.html`. 30

[8] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL combinational benchmark suite. In *24th International Workshop on Logic & Synthesis*, 2015. 30

[9] Anamika and Manu Bansal. Effect of various crossover operators in memetic algorithm on multi-input adders. *Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering*, 2, 2014. 202, 203, 204

[10] Lars Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *6th International Symposium on Algorithms and Computations*, pages 82–91, 1995. doi:10.1007/BFb0015411. 15, 17

[11] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Workshop on Algorithms and Data Structures*, pages 334–345. Springer, 1995. doi:10.1007/3-540-60220-8_74. 17, 174, 181, 184

[12] Lars Arge. The I/O-complexity of ordered binary-decision diagram. In *BRICS RS preprint series*, volume 29. Department of Computer Science, University of Aarhus, 1996. doi:10.7146/brics.v3i29.20010. 15, 17, 178, 189, 190, 211

[13] Pranav Ashar and Matthew Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *International Conference on Computer-Aided Design*, page 622–627. IEEE Computer Society Press, 1994. ISBN 0897916905. 210

[14] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Transactions on Parallel Computing*, 9(1), 2022. doi:10.1145/3505286. 213

[15] Junaid Babar, Chuan Jiang, Gianfranco Ciardo, and Andrew Miner. Binary decision diagrams with edge-specified reductions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 303–318. Springer, 2019. doi:10.1007/978-3-030-17465-1_17. 10

[16] J. W. de Bakker and D. Scott. A theory of programs. Unpublished, 1969. 12

[17] Jagdish Chand Bansal, Harish Sharma, Shimpi Singh Jadon, and Maurice Clerc. Spider monkey optimization algorithm for numerical optimization. *Memetic Computing*, 6:31–47, 2014. doi:10.1007/s12293-013-0128-0. 195, 205

[18] Roberto J. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Artificial Intelligence*, pages 203–208. AAAI Press, 1997. URL `https://cdn.aaai.org/AAAI/1997/AAAI97-032.pdf`. 12

[19] Nikola Beneš, Luboš Brim, Jakub Kadlecaj, Samuel Pastva, and David Šafránek. AEON: Attractor bifurcation analysis of parametrised Boolean networks. In *Computer Aided Verification*, pages 569 – 581. Springer, 2020. doi:10.1007/978-3-030-53288-8_28. 15, 31

[20] Jochen Bern, Christoph Meinel, and Anna Slobodová. Global rebuilding of OBDDs avoiding memory requirement maxima. In *Computer Aided Verification*, pages 4–15. Springer, 1995. doi:10.1007/3-540-60045-0_36. 173, 209

[21] Jochen Bern, Christoph Meinel, and Anna Slobodová. Global rebuilding of OBDDs avoiding memory requirement maxima. *Computer-Aided Design of Integrated Circuits and Systems*, 15:131–134, 1996. 173, 189, 190, 209, 210

[22] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Programming Language Design and Implementation*, pages 103–114. Association for Computing Machinery, 2003. doi:10.1145/781131.781144. 14

[23] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190. Springer, 2011. doi:10.1007/978-3-642-22110-1_16. 14, 30

[24] Dirk Beyer and Andreas Stahlbauer. BDD-based software verification. *Software Tools for Technology Transfer*, 16:507–518, 2014. doi:10.1007/s10009-014-0334-1. 14

[25] Dirk Beyer, Karlheinz Friedberger, and Stephan Holzner. PJBDD: A BDD library for Java and multi-threading. In *Automated Technology for Verification and Analysis*, pages 144–149. Springer, 2021. doi:10.1007/978-3-030-88885-5_10. 8, 15, 30, 31

[26] Armin Biere and Marijn Heule. The effect of scrambling CNFs. In *Proceedings of Pragmatics of SAT 2015 and 2018*, volume 59 of *EPiC Series in Computing*, pages 111–126. EasyChair, 2019. doi:10.29007/9dj5. 12

[27] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207. Springer, 1999. doi:10.1007/3-540-49059-0_14. 12

[28] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. ddNF: An efficient data structure for header spaces. In *Hardware and Software: Verification and Testing*, pages 49–64. Springer, 2016. doi:10.1007/978-3-319-49052-6_4. 13

[29] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996. doi:10.1109/12.537122. 195, 200

[30] Beate Bollig, Martin Löbbing, and Ingo Wegener. Simulated annealing to improve variable orderings for OBDDs. In *International Workshop on Logic Synthesis*, 1995. 201, 204

[31] Beate Bollig, Martin Löbbing, and Ingo Wegener. On the effect of local changes in the variable ordering of ordered decision diagrams. *Information Processing Letters*, 59(5):233–239, 1996. doi:10.1016/0020-0190(96)00119-6. 184, 187, 188, 193, 197, 198, 199, 200, 201, 204, 210

[32] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic logic dimulation and temporal logic. In *Proc. of the IMEC-IFIP Intl. Workshop Applied Formal Methods for Correct VLSI Design*, pages 759–767, 1989. 12

[33] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th Design Automation Conference (DAC)*, pages 40–45. Association for Computing Machinery, 1990. doi:10.1109/DAC.1990.114826. 7, 8, 9, 213

[34] Julian Bradfield and Igor Walukiewicz. The mu-calculus and model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, chapter 26. Springer, 2018. doi:10.1007/978-3-319-10575-8_26. 12

[35] Sebastiaan Brand, Thomas Bäck, and Alfons Laarman. A decision diagram operation for reachability. In *Formal Methods*, pages 514–532. Springer, 2023. doi:10.1007/978-3-031-27481-7_29. 13

[36] Michael Brickenstein and Alexander Dreyer. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326–1345, 2009. doi:10.1016/j.jsc.2008.02.017. 14

[37] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. Lessons from the evolution of the Batfish configuration analysis tool. In *ACM SIGCOMM*, page 122–135. Association for Computing Machinery, 2023. doi:10.1145/3603269.3604866. 13, 31

[38] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. 6, 7, 8, 9, 12, 17, 20, 173, 174, 179, 183

[39] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992. ISSN 0360-0300. 9, 12

[40] Randal E. Bryant. Chain reduction for binary and zero-suppressed decision diagrams. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–98. Springer, 2018. doi:10.1007/978-3-319-89960-2_5. 10

[41] Randal E. Bryant. TBuDDy: A proof-generating BDD package. In *Formal Methods in Computer-Aided Design*, pages 49–58, 2022. 13

[42] Randal E. Bryant and Marijn J. H. Heule. Dual proof generation for quantified Boolean formulas with a BDD-based solver. In *Automated Deduction – CADE 28*, pages 433–449. Springer, 2021. doi:10.1007/978-3-030-79876-5_25.

[43] Randal E. Bryant and Marijn J. H. Heule. Generating extended resolution proofs with a BDD-based SAT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 76–93. Springer, 2021. doi:10.1007/978-3-030-72016-2_5. 13

[44] Randal E. Bryant and Carl-Johan H. Seger. Formal verification of digital circuits using symbolic ternary system models. In *Computer-Aided Verification*, pages 33–43. Springer Berlin Heidelberg, 1990. doi:10.1007/BFb0023717. 13

[45] Randal E. Bryant, Armin Biere, and Marijn J. H. Heule. Clausal proofs for pseudo-Boolean reasoning. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–461. Springer, 2022. doi:10.1007/978-3-030-99524-9_25. 13

[46] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51. Association for Computing Machinery, 1991. doi:10.1145/123186.123223. 12

[47] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992. doi:10.1016/0890-5401(92)90017-A. 12

[48] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994. doi:10.1109/43.275352. 12, 173

[49] Gianpiero Cabodi, Stefano Quer, Christoph Meinel, Harald Sack, Anna Slobodová, and Christian Stangier. Binary decision diagrams and the multiple variable order problem. In *International Workshop on Logic Synthesis*, pages 346–352. IEEE/ACM, 1998. 173, 174, 195

[50] Luigi Capogrosso, Luca Geretti, Marco Cristani, Franco Fummi, and Tiziano Villa. HermesBDD: A multi-core and multi-platform binary decision diagram package. In *Design and Diagnostics of Electronic Circuits and Systems*, pages 87–90, 2023. doi:10.1109/DDECS57882.2023.10139480. 8, 14, 15, 30

[51] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Optimization Theory and Applications*, 45(1): 41–51, 1985. doi:10.1007/BF00940812. 195, 201

[52] Rohit Chadha, Umang Mathur, and Stefan Schwoon. Computing information flow using symbolic model-checking. In *Foundation of Software Technology and Theoretical Computer Science*, pages 505–516. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.FSTTCS.2014.505. 14

[53] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139—-149. Society for Industrial and Applied Mathematics, 1995. ISBN 0898713498. 17, 174

[54] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *International Journal on Software Tools for Technology Transfer*, 8:4–25, 2006. doi:10.1007/s10009-005-0188-7. 13

[55] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002. doi:10.1007/3-540-45657-0_29. 13

[56] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Functional Programming*, pages 268–279. Association for Computing Machinery, 2000. doi:10.1145/351240.351266. 32

[57] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, pages 52–71. Springer, 1982. doi:10.1007/BFb0025774. 3, 12

[58] Anders Benjamin Clausen and Kent Nielsen. I/O-efficient static variable reordering for binary decision diagrams. Bachelor's thesis, Department of Computer Science, University of Aarhus, 2022. 189, 190, 191

[59] The Coq Development Team. The Coq Proof Assistant. Zenodo, 2017. URL `https://doi.org/10.5281/zenodo.1003420`. 3

[60] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *Computer-Aided Design*, pages 126–129. IEEE, 1990. doi:10.1109/ICCAD.1990.129859. 12

[61] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, pages 365–373. Springer, 1990. doi:10.1007/3-540-52148-8_30. 12

[62] Ian J. Davis. A fast radix sort. *The Computer Journal*, 35(6):636–642, 1992. doi:10.1093/comjnl/35.6.636. 213

[63] Richard Dawkins. *The selfish gene*. Oxford university press, 1976. 203

[64] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: Standard template library for XXL data sets. In *Algorithms – ESA 2005*, pages 640–651. Springer, 2005. doi:10.1007/11561071_57. 213

[65] Tom van Dijk and Jaco van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19:675–696, 2016. doi:10.1007/s10009-016-0433-2. 8, 11, 14, 24, 28, 31

[66] Tom van Dijk, Jeroen Meijer, and Jaco van de Pol. Multi-core on-the-fly saturation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 58–75. Springer, 2019. doi:10.1007/978-3-030-17465-1_4. 13

[67] Tom van Dijk. Personal Communication, 10 2021. 173

[68] Tom van Dijk, Robert White, and Robert Meolic. Tagged BDDs: Combining reduction rules from different decision diagram types. In *Formal Methods in Computer Aided Design*, pages 108–115. IEEE Computer Society Press, 2017. 10

[69] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996. doi:10.1109/3477.484436. 195, 206

[70] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Positive feedback as a search strategy. Technical report, Dip Elettronica, Politecnico di Milano, Italy, 1999. 195, 206

[71] Robert Drechsler and Wolfgang Gunther. Using lower bounds during dynamic BDD minimization. In *Design Automation Conference*, pages 29–32, 1999. doi:10.1109/DAC.1999.781225. 184, 188, 193, 197, 198, 199, 200, 207, 208

[72] Rolf Drechsler, Bernd Becker, and Nicole Göckel. Genetic algorithm for variable ordering of OBDDs. *Computers and Digital Techniques*, 143(6): 364–368, 1996. 196, 201, 202, 203, 204, 210

[73] Rolf Drechsler, Nicole Göckel, and Bernd Becker. Learning heuristics for OBDD minimization by evolutionary algorithms. In *Parallel Problem Solving from Nature — PPSN IV*, pages 730–739. Springer, 1996. doi:10.1007/3-540-61723-X_1036. 211

[74] Clemens Dubslaff and Joshua Wirtz. Compiling binary decision diagrams with interrupt-based downsizing. In *Principles of Verification: Cycling the Probabilistic Landscape : Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*, pages 252–273. Springer, 2025. doi:10.1007/978-3-031-75778-5_12. 212

[75] Clemens Dubslaff, Nils Husung, and Nikolai Käfer. Configuring BDD compilation techniques for feature models. In *Systems and Software Product Line*, page 209–216. Association for Computing Machinery, 2024. doi:10.1145/3646548.3676538. 13, 31

[76] Russel Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, 1995. doi:10.1109/MHS.1995.494215. 195, 205

[77] Eric Felt, Gary York, Robert Brayton, and Alberto Sangiovanni-Vincentelli. Dynamic variable reordering for BDD minimization. In *European Design Automation Conference*, pages 130–135, 1993. doi:10.1109/EURDAC.1993.410627. 24, 173, 208

[78] Chen Fu, Ernst Moritz Hahn, Yong Li, Sven Schewe, Meng Sun, Andrea Turrini, and Lijun Zhang. EPMC gets knowledge in multi-agent systems. In *Verification, Model Checking, and Abstract Interpretation*, pages 93–107. Springer, 2022. doi:10.1007/978-3-030-94583-1_5. 11, 13

[79] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169, 1997. doi:10.1023/A:1008647823331. 10, 11, 24

[80] Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvement of Boolean comparison method based on binary decision diagrams. In *International Conference on Computer-Aided Design ()*. IEEE, 1988. doi:10.1109/ICCAD.1988.122450. 173, 196, 197

[81] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation.*, pages 50–54, 1991. doi:10.1109/EDAC.1991.206358. 186, 188, 192

[82] Peter Gammie and Ron van der Meyden. MCK: Model checking the logic of knowledge. In *Computer Aided Verification*, pages 479–483. Springer, 2004. doi:10.1007/978-3-540-27813-9_41. 14

[83] Nicole Gockel, Rolf Drechsler, and Bernd Becker. Learning heuristics for OKFDD minimization by evolutionary algorithms. In *Asia and South Pacific Design Automation Conference*, pages 469–472, 1997. doi:10.1109/ASPDAC.1997.600307. 211

[84] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer, 1996. doi:10.1007/3-540-60761-7. 4

[85] Wolfgang Günther and Rolf Drechsler. Improving EAs for sequencing problems. In *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, pages 175–180. Morgan Kaufmann Publishers Inc., 2000. URL `https://dl.acm.org/doi/abs/10.5555/2933718.2933742`. 200, 203, 204

[86] Mark C. Hansen, Hakan Yalcin, and John P. Hayes. Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999. doi:10.1109/54.785838. 30

[87] Jelle Hellings, George H.L. Fletcher, and Herman Haverkort. Efficient external-memory bisimulation on DAGs. In *ACM SIGMOD International Conference on Management of Data*, pages 553–564. Association for Computing Machinery, 2012. doi:10.1145/2213836.2213899. 20

[88] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker Storm. *Software Tools for Technology Transfer*, 24:589–610, 2022. doi:10.1007/s10009-021-00633-z. 11, 13

[89] Tobias Heß, Chico Sundermann, and Thomas Thüm. On the scalability of building binary decision diagrams for current feature models. In *Systems and Software Product Line*, pages 131–135. Association for Computing Machinery, 2021. doi:10.1145/3461001.3474452. 13

[90] Paul Hildebrandt and Harold Isbitz. Radix exchange—an internal sorting method for digital computers. *Journal of the ACM*, 6(2):156–163, 1959. doi:10.1145/320964.320972. 213

[91] Peter Hitchcock and David Michael Ritchie Park. Induction rules and termination proofs. In *International Colloquium on Automata, Languages and Programming*, 1972. 12

[92] John H. Holland. Adaption in natural and artificial systems, 1975. 195, 201, 202, 203

[93] Nils Husung, Clemens Dubslaff, Holger Hermanns, and Maximilian A. Köhl. OxiDD: A safe, concurrent, modular, and performant decision diagram framework in Rust. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24)*. Springer, 2024. doi:10.1007/978-3-031-57256-2_13. 8, 14, 15, 31

[94] Nagisa Ishiura, Hiroshi Sawada, and Shuzo Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Computer-Aided Design Digest of Technical Papers*, pages 472–475. IEEE, 1991. doi:10.1109/ICCAD.1991.185307. 183, 184, 200

[95] Anna Blume Jakobsen, Rasmus Skibdahl Melanchton Jørgensen, Jaco van de Pol, and Andreas Pavlogiannis. Fast symbolic computation of bottom SCCs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 110–128. Springer, 2024. doi:10.1007/978-3-031-57256-2_6. 13

[96] David S. Johnson and Lyle McGeoch. The traveling salesman problem: A case study. In Emile Aarts and Jan Karel Lenstra, editors, *Local Search in Combinatorial Optimization*. Princeton University Press, 2003. doi:10.2307/j.ctv346t9c. 203

[97] Martin Jonáš and Jan Strejček. Q3B: An efficient BDD-based SMT solver for quantified bit-vectors. In *Computer Aided Verification*, pages 64–73. Springer, 2019. doi:10.1007/978-3-030-25543-5_4. 13

[98] Toni Jussila, Carsten Sinz, and Armin Biere. Extended resolution proofs for symbolic SAT solving with quantification. In *Theory and Applications of Satisfiability Testing*, pages 54–60. Springer, 2006. doi:10.1007/11814948_8. 13

[99] Roope Kaivola and Neta Bar Kama. Timed causal fanin analysis for symbolic circuit simulation. In *Formal Methods in Computer-Aided Design*, pages 99–107, 2022. doi:10.34727/2022/isbn.978-3-85448-053-2_16. 13

[100] Roope Kaivola and John O'Leary. Verification of arithmetic and datapath circuits with symbolic simulation. In Anupam Chattopadhyay, editor, *Handbook of Computer Architecture*, pages 1–52. Springer, 2022. doi:10.1007/978-981-15-6401-7_37-1. 173

[101] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in Intel® core™ i7 processor execution engine validation. In *Computer Aided Verification*, pages 414–429. Springer, 2009. doi:10.1007/978-3-642-02658-4_32. 13

[102] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977. doi:10.1007/BF00290339. 14

[103] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Multi-valued decision diagrams for logic synthesis and verification. Technical Report UCB/ERL M96/75, Electronics Research Laboratory, University of California, Berkeley, 1996. 11

[104] Gijs Kant, Alfons Laarman, Jeroenn Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-performance language-independent model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 692–707, Berlin, Heidelberg, 2015. Springer. doi:10.1007/978-3-662-46681-0_61. 13

[105] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations. The IBM Research Symposia Series*, pages 85–103. Springer, 1972. doi:10.1007/978-1-4684-2001-2_9. 206

[106] Kevin Karplus. Representing Boolean functions with if-then-else DAGs. Technical report, University of California at Santa Cruz, USA, 1988. 9, 10, 24, 185

[107] James Kennedy and Russel Eberhart. Particle swarm optimization. In *International Conference on Neural Networks*, pages 1942–1948, 1995. doi:10.1109/ICNN.1995.488968. 195, 205

[108] S. Kirkpatrick, C. D. Gelett, and M. P. Vechi. Optimization by simulated annealing. *Science*, 220(4598):621–630, 1983. 195, 201

[109] Nils Klarlund and Theis Rauhe. BDD algorithms and cache misses. In *BRICS Report Series*, volume 26, 1996. doi:10.7146/brics.v3i26.20007. 8, 15

[110] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(04):571–586, 2002. doi:10.1142/S012905410200128X. 11

[111] F. Kordon, P. Bouvier, H. Garavel, F. Hulin-Hubard, N Amat, E. Amparore, B. Berthomieu, D. Donatelli, S. Dal Zilio, P. G. Jensen, L. Jezequel, E. Paviot-Adet, J. Srba, and Y. Thierry-Mieg. Complete results for the 2023 edition of the model checking contest, 2023. URL `https://mcc.lip6.fr/2023/results.php`. 30

[112] Dexter Kozen. Results on the propositional $\mu$-calculus. In *Automata, Languages and Programming*, pages 348–359. Springer Berlin Heidelberg, 1982. doi:10.1007/BFb0012782. 12

[113] Daniel Kunkle, Vlad Slavici, and Gene Cooperman. Parallel disk-based computation for large, monolithic binary decision diagrams. In *4th International Workshop on Parallel Symbolic Computation (PASCO)*, pages 63–72, 2010. doi:10.1145/1837210.1837222. 30

[114] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 200–204. Springer, 2002. doi:10.1007/3-540-46029-2_13. 11, 13

[115] Yung-Te. Lai and Sarma Sastry. Edge-valued binary decision for multi-level hierarchical verification. In *Design Automation Conference*, pages 608–613, 1992. doi:10.1109/DAC.1992.227813. 10, 11, 24

[116] Casper Abild Larsen, Simon Meldahl Schmidt, Jesper Steensgaard, Anna Blume Jakobsen, Jaco van de Pol, and Andreas Pavlogiannis. A truly symbolic linear-time algorithm for SCC decomposition. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 353–371. Springer, 2023. doi:10.1007/978-3-031-30820-8_22. 13

[117] C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985 – 999, 1959. doi:10.1002/j.1538-7305.1959.tb01585.x. 9, 173

[118] Shen Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973. 208

[119] Jørn Lind-Nielsen. BuDDy: A binary decision diagram package. Technical report, Department of Information Technology, Technical University of Denmark, 1999. 8, 9, 14, 31, 174, 188, 217

[120] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: an open-source model checker for the verification of multi-agent systems. *Software Tools for Technology Transfer*, 19(1):9–30, 2017. doi:10.1007/s10009-015-0378-x. 14

[121] David E. Long. The design of a cache-friendly BDD library. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design*, pages 639–645. Association for Computing Machinery, 1998. doi:10.1145/288548.289102. 8, 15, 217

[122] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, and George Varghese. Network verification in the light of program verification. Technical report, Microsoft Research, 2013. URL `https://microsoft.com/en-us/research/publication/network-verification-in-the-light-of-program-verification/`. 13, 31

[123] Alberto Lovato, Damiano Macedonio, and Fausto Spoto. A thread-safe library for binary decision diagrams. In *Software Engineering and Formal Methods*, pages 35–49. Springer, 2014. doi:10.1007/978-3-319-10431-7_4. 14, 31

[124] Jean-Christophe Madre and Jean-Paul Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *25th Design Automation Conference*, pages 205–210. IEEE Computer Society Press, 1988. 9, 10, 12, 24, 185

[125] Magnus Madsen and Jaco van de Pol. Polymorphic types and effects with Boolean unification. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020. doi:10.1145/3428222. 14, 31

[126] Magnus Madsen, Jaco van de Pol, and Troels Henriksen. Fast and efficient boolean unification for Hindley-Milner-style type and effect systems. *Proceedings of the ACM on Programing Languages*, 7(OOPSLA 2), 2023. doi:10.1145/3622816. 14, 31

[127] Sharad Malik, Albert R. Wang, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference on Computer-Aided Design*. IEEE, 1988. doi:10.1109/ICCAD.1988.122451. 173

[128] João P. Marques Silva and Karem A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Computer Aided Design*, pages 220–227. IEEE, 1996. doi:10.1109/ICCAD.1996.569607. 12

[129] Peter M. McIlroy, Keith Bostic, and Douglas McIlroy. Engineering radix sort. *Computing Systems*, 6:5–27, 1993. 213

[130] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification*, pages 1–13. Springer, 2003. doi:10.1007/978-3-540-45069-6_1. 12

[131] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955. doi:10.1002/j.1538-7305.1955.tb03788.x. 12

[132] Jeroen Meijer and Jaco van de Pol. Bandwidth and wavefront reduction for static variable ordering in symbolic model checking. arXiv, 2015. URL `https://arxiv.org/abs/1511.08678`. 173

[133] Christoph Meinel and Anna Slobodová. On the complexity of constructing optimal ordered binary decision diagrams. In *Mathematical Foundations of Computer Science*, pages 515–524. Springer, 1994. doi:10.1007/3-540-58338-6_98. 189, 209, 210

[134] Christoph Meinel and Anna Slobodova. Speeding up variable reordering of OBDDs. In *Computer Design VLSI in Computers and Processors*, pages 338–343, 1997. doi:10.1109/ICCD.1997.628892. 198, 206, 207, 208

[135] Christoph Meinel, Fabio Somenzi, and Thorsten Theobald. Linear sifting of decision diagrams. In *Proceedings of the 34th Annual Design Automation Conference*, pages 202–207. Association for Computing Machinery, 1997. doi:10.1145/266021.266066. 207

[136] Tom Melham. Symbolic trajectory evaluation. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, chapter 25, pages 831–870. Springer, 2018. doi:10.1007/978-3-319-10575-8_25. 13, 173

[137] M. R. Mercer, R. Kapur, and D. E. Ross. Functional approaches to generating orderings for efficient symbolic representations. In *Design Automation Conference*, pages 624–627. ACM/IEEE, 1992. doi:10.1109/DAC.1992.227810. 183, 184

[138] Donald Michie. "memo" functions and machine learning. *Nature*, 218:19–22, 1968. doi:10.1038/218019a0. 8

[139] D. Michael Miller and Mitchell A. Thornton. QMDD: A decision diagram structure for reversible and quantum circuits. In *36th International Symposium on Multiple-Valued Logic*, pages 30–36, 2006. doi:10.1109/ISMVL.2006.35. 11, 14, 24

[140] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, page 272–277. Association for Computing Machinery, 1993. doi:10.1145/157485.164890. 9, 10, 11, 21, 181

[141] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *27th Design Automation Conference*, pages 52–57. Association for Computing Machinery, 1990. doi:10.1145/123186.123225. 7, 8, 209, 213

[142] Seyedali Mirjalili. *Genetic Algorithm*, pages 43–55. Springer, 2019. doi:10.1007/978-3-319-93025-1_4. 202

[143] A. Mitra and S. Chattopadhyay. Variable ordering for shared binary decision diagrams targeting node count and path length optimisation using particle swarm technique. *IET Computers & Digital Techniques*, 6:353–361, 2012. doi:10.1049/iet-cdt.2011.0051. 205, 206, 210

[144] Thomas Mølhave. Using TPIE for processing massive data sets in C++. *SIGSPATIAL Special*, 4(2):24–27, 2012. doi:10.1145/2367574.2367579. 20, 25, 197, 213

[145] Pablo Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical report, California Institute of Technology, 1989. 195, 202, 203, 204

[146] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001. doi:10.1145/378239.379017. 12

[147] Michael Norman and Pablo Moscato. A competitive-cooperative approach to complex combinatorial search. Technical report, California Institute of Technology, 1991. 195, 203, 204

[148] C. Norris IP and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996. doi:10.1007/BF00625968. 4

[149] Hiroyuki Ochi, Koichi Yasuoka, and Shuzo Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *International Conference on Computer-Aided Design*, pages 48–55, 1993. 210

[150] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Saïs. Recovering and exploiting structural knowledge from CNF formulas. In *Principles and Practice of Constraint Programming*, pages 185–199. Springer, 2002. doi:10.1007/3-540-46135-3_13. 12

[151] Shipra Panda and Fabio Somenzi. Who are the variables in your neighbourhood. In *International Conference Computer-Aided Design*, pages 74–77, 1995. doi:10.1109/ICCAD.1995.479994. 207

[152] Shipra Panda, Fabio Somenzi, and Bernard F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *International Conference on Computer-Aided Design*, pages 628–631. IEEE Computer Society Press, 1994. URL `https://dl.acm.org/doi/10.5555/191326.191598`. 198, 207, 208

[153] Samuel Pastva and Thomas Henzinger. Binary decision diagrams on modern hardware. In *Conference on Formal Methods in Computer-Aided Design*, pages 122–131, 2023. 8, 9, 15, 17, 30, 217

[154] Doron Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification*, pages 409–423. Springer, 1993. doi:10.1007/3-540-56922-7_34. 4

[155] Andrej Pištek. Dynamic variable reordering for binary decision diagrams. Master's thesis, Twente University, 2023. URL `http://essay.utwente.nl/96753/`. 188

[156] V. R. Pratt. A decidable $\mu$-calculus (preliminary report). In *Annual Symposium on Foundations of Computer Science*, pages 421–427, 1981. 12

[157] Jean-Pierre. Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982. doi:10.1007/3-540-11494-7_22. 3

[158] Prabhakar Raghavan and Clark D. Tompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7: 365–374, 1987. doi:10.1007/BF02579324. 208

[159] Rajeev K. Ranjan, Wilsin Gosti, Robert K. Brayton, and Alberto Sangiovanni-Vincenteili. Dynamic reordering in a breadth-first manipulation based BDD package: challenges and solutions. In *Proceedings International Conference*

*on Computer Design VLSI in Computers and Processors*, pages 344–351, 1997. doi:10.1109/ICCD.1997.628893. 210, 211

[160] Mathias Rav. Personal Communication, 11 2024. 213

[161] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 42–47, 1993. doi:10.1109/ICCAD.1993.580029. 24, 173, 186, 188, 192, 193, 195, 196, 200, 203, 205, 206, 207, 208, 210, 211

[162] Peter Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5:7–32, 2000. doi:10.1145/351827.384249. 213

[163] Peter Sanders. Personal Communication, 08 2024. 213

[164] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *Algorithms – ESA 2004*, pages 784–796. Springer, 2004. doi:10.1007/978-3-540-30140-0_69. 213

[165] Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierarchy. In *Proceedings of the 33rd Annual Design Automation Conference*, page 635–640. Association for Computing Machinery, 1996. doi:10.1145/240518.240638. 31, 210

[166] Petr Savický and Ingo Wegener. Efficient algorithms for the transformation between different types of binary decision diagrams. *Acta Informatica*, 34(4): 245–256, 1997. doi:10.1007/s002360050083. 189, 190, 191, 202, 209, 210

[167] Christoph Scholl, Bernd Becker, and Andreas Brogle. The multiple variable order problem for binary decision diagrams: Theory and practical application. In *Asia and South Pacific Design Automation Conference*, pages 85–90, 2001. doi:10.1109/ASPDAC.2001.913285. 173, 174

[168] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. In *Formal Methods in System Design*, pages 147–189. Springer, 1995. doi:10.1007/BF01383966. 13

[169] Irfansha Shaik and Jaco van de Pol. Optimal layout synthesis for deep quantum circuits on NISQ processors with 100+ qubits, 2024. 12

[170] Claude E. Shannon. A symbolic analysis of relay and switching circuits. *Electrical Engineering*, 57(12):713–723, 1938. doi:10.1109/EE.1938.6431064. 198

[171] Mohammed Balal Siddiqui, Mirza Tariq Beg, and Syed Naseem Ahmad. Variable ordering in binary decision diagram using spider monkey optimization for node and path length optimization. *IEICE Transactions on Fundamentals of*

*Electronics, Communications and Computer Sciences*, 106(7):976–989, 2023. 205, 206

[172] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining BDDs. In *Computer Science – Theory and Applications*, pages 600–611. Springer, 2006. doi:10.1007/11753728_60. 13

[173] Anna Slobodová. On the composition problem for OBDDs with multiple variable orders. In *Mathematical Foundations of Computer Science 1998*, pages 645–655. Springer, 1998. 173

[174] Martin Šmérek. Personal Communication, 09 2021. 19

[175] Steffan Christ Sølvsten and Jaco van de Pol. Predicting memory demands of BDD operations using maximum graph cuts. In *Automated Technology for Verification and Analysis*, pages 72–92. Springer, 2023. doi:10.1007/978-3-031-45332-8_4. 18, 19, 21, 25, 29, 197, 199, 212

[176] Steffan Christ Sølvsten and Jaco van de Pol. Adiar 1.1: Zero-suppressed decision diagrams in external memory. In *NASA Formal Methods Symposium*, Berlin, Heidelberg, 2023. Springer. doi:10.1007/978-3-031-33170-1_28. 18, 19, 21, 25

[177] Steffan Christ Sølvsten and Jaco van de Pol. Predicting memory demands of BDD operations using maximum graph cuts (extended paper), 2023. URL `https://arxiv.org/abs/2307.04488`. 19, 21, 25

[178] Steffan Christ Sølvsten and Jaco van de Pol. Multi-variable quantification of BDDs in external memory using nested sweeping (extended paper). arXiv, 2024. 14, 19, 22, 25, 29, 180, 181, 188

[179] Steffan Christ Sølvsten, Jaco van de Pol, Anna Blume Jakobsen, and Mathias Weller Berg Thomasen. Efficient binary decision diagram manipulation in external memory. arXiv, 2021. 19, 29, 216

[180] Steffan Christ Sølvsten, Jaco van de Pol, Anna Blume Jakobsen, and Mathias Weller Berg Thomasen. Adiar: Binary decision diagrams in external memory. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 295–313, Berlin, Heidelberg, 2022. Springer. doi:10.1007/978-3-030-99527-0_16. 14, 19, 21, 31, 181, 183, 188, 197, 216

[181] Steffan Christ Sølvsten, Casper Moldrup Rysgaard, and Jaco van de Pol. Random access on narrow decision diagrams in external memory. In *Model Checking Software*, pages 137–145. Springer, 2024. doi:10.1007/978-3-031-66149-5_7. 18, 19, 22, 25, 198

[182] Fabio Somenzi. CUDD: CU decision diagram package, 3.0. Technical report, University of Colorado at Boulder, 2015. 8, 14, 28, 31, 188, 201, 205, 209, 210

[183] Fausto Spoto. The Julia static analyzer for Java. In *Static Analysis*, pages 39–57. Springer, 2016. doi:10.1007/978-3-662-53413-7_3. 14

[184] Kaile Su, Abdul Sattar, and Xiangyu Luo. Model checking temporal logics of knowledge via OBDDs. *The Computer Journal*, 50(4):403–420, 2007. doi:10.1093/comjnl/bxm009. 14

[185] Paul Tafertshofer and Massoud Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, 10:243–270, 1997. doi:10.1023/A:1008691605584. 11, 24

[186] Seiichiro Tani and Hiroshi Imai. A reordering operation for an ordered binary decision diagram and an extended framework for combinatorics of graphs. In *Algorithms and Computation*, pages 575–583. Springer, 1994. doi:10.1007/3-540-58325-4_225. 189, 190, 209

[187] Seiichiro Tani, Kiyoharu Hamaguchi, and Shuzo Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *Algorithms and Computation*, pages 389–398. Springer Berlin Heidelberg, 1993. doi:10.1007/3-540-57568-5_270. 195, 200

[188] Dimitrios Thanos, Alejandro Villoria, Sebastiaan Brand, Arend-Jan Quist, Jingyi Mei, Tim Coopmans, and Alfons Laarman. Automated reasoning in quantum circuit compilation. In *Model Checking Software*, pages 106–134. Springer, 2024. doi:10.1007/978-3-031-66149-5_6. 11, 14

[189] Joan Thibault and Khalil Ghorbal. Functional decision diagrams: A unifying data structure for binary decision diagrams. Research Report RR-9306, INRIA Rennes - Bretagne Atlantique and University of Rennes 1, France, 2019. URL `https://inria.hal.science/hal-02369112`. 10, 215

[190] Joan Thibault and Khalil Ghorbal. Ordered functional decision diagrams. Research Report RR-9333, Inria, 2020. URL `https://inria.hal.science/hal-02512117`. 10, 215

[191] M. A. Thornton, J. P. Williams, R. Drechsler, N. Drechsler, and D. M. Wessels. SBDD variable reordering based on probabilistic and evolutionary algorithms. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 381–387, 1999. doi:10.1109/PACRIM.1999.799556. 203, 206, 209

[192] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer, 1983. doi:10.1007/978-3-642-81955-1_28. 12

[193] Arash Vahidi. JDD: a pure Java BDD and Z-BDD library. Bitbucket, 2003. URL `https://bitbucket.org/vahidi/jdd`. 31

[194] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, pages 491–515. Springer, 1991. doi:10.1007/3-540-53863-1_36. 4

[195] Tom van Dijk, Ernst Moritz Hahn, David N. Jansen, Yong Li, Thomas Neele, Mariëlle Stoelinga, Andrea Turrini, and Lijun Zhang. A comparative study of BDD packages for probabilistic symbolic model checking. In *Dependable Software Engineering: Theories, Tools, and Applications*, pages 35–51. Springer, 2015. doi:10.1007/978-3-319-25942-0_3. 23

[196] Darren Erik Vengroff. A Transparent Parallel I/O Environment. In *DAGS Symposium on Parallel Computation*, pages 117–134, 1994. 25, 213

[197] Jan Wassenberg and Peter Sanders. Engineering a multi-core radix sort. In *Euro-Par 2011 Parallel Processing*, pages 160–169. Springer, 2011. doi:10.1007/978-3-642-23397-5_16. 213

[198] Ingo Wegener. The size of reduced OBDD's and optimal read-once branching programs for almost all Boolean functions. *IEEE Transactions on Computers*, 43(11):1262–1269, 1994. doi:10.1109/12.324559. 189

[199] Bwolen Yang, Randal E. Bryant, David R. O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan, and Fabio Somenzi. A performance study of BDD-based model checking. In *Formal Methods in Computer-Aided Design*, pages 255–289. Springer, 1998. doi:10.1007/3-540-49519-3_18. 31