Adiar

Binary Decision Diagrams in External Memory

Steffan Christ Sølvsten (✉), Jaco van de Pol,
Anna Blume Jakobsen, and Mathias Weller Berg Thomasen

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

Abstract. We follow up on the idea of Lars Arge to rephrase the Reduce and Apply operations 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. We provide asymptotic improvements to the few procedures that can be derived using Apply.
Our work has culminated in a BDD package named Adiar that is able to efficiently manipulate BDDs that outgrow main memory. This makes Adiar surpass the limits of conventional BDD packages that use recursive depth-first algorithms. It is able to do so while still achieving a satisfactory performance compared to other BDD packages: Adiar, in parts using the disk, is on instances larger than 9.5 GiB only 1.47 to 3.69 times slower compared to CUDD and Sylvan, exclusively using main memory. Yet, Adiar is able to obtain this performance at a fraction of the main memory needed by conventional BDD packages to function.

Keywords: Time-forward Processing · External Memory Algorithms · Binary Decision Diagrams

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 graphs [10, 11].

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 [14, 15], MCK [17], LTSmin [19], and MCMAS [24] and the recently envisioned symbolic model checking algorithms for CTL* in [3] and for CTLK in [18]. 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 [35] for the GPU and in 2016 by Van Dijk and Van de Pol [16] for multi-core processors [12].

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 295–313, 2022.
https://doi.org/10.1007/978-3-030-99527-0_16

The most widely used implementations of decision diagrams make use of recursive depth-first algorithms and a unique node table [16, 23, 34]. 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 [21,26]. 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 upper limit on the size of the BDDs.

Related Work. Prior work has been done to overcome the I/Os spent while computing on BDDs. David Long [25] 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. Ochi, Yasuoka, and Yajima [28] designed in 1993 breadth-first BDD algorithms that exploit a levelwise locality on disk. Their technique was improved by Ashar and Cheong [8] in 1994 and by Sanghavi et al. [31] in 1996. The fruits of their labour was the BDD library CAL capable of manipulating BDDs larger than available main memory. Kunkle, Slavici and Cooperman [22] extended in 2010 the breadth-first approach to distributed BDD manipulation.

The breadth-first algorithms in [8, 28, 31] 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, so they get synchronized with the iteration through the sorted input stream of nodes. Martin Šmérek implemented these algorithms in 2009 as they were described, but the performance was disappointing, since the intermediate unreduced BDD grew too large to handle in practice [personal communication, Sep 2021].

Contributions. Our work directly follows up on the theoretical contributions of Arge in [5, 6]. We simplified and improved on his I/O-optimal Apply and Reduce algorithms. In particular, we modified and pruned the intermediate representation, to prevent data duplication and to save on the number of sorting operations. 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.

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 Adiar is able to manipulate BDDs larger than the given main memory available, with only an acceptable slowdown compared to a conventional BDD library running exclusively in main memory.

1.1 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. Section 4 provides an overview of the resulting BDD package, Adiar, and Section 5 contains an experimental evaluation of it. Our conclusions and future work are 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.

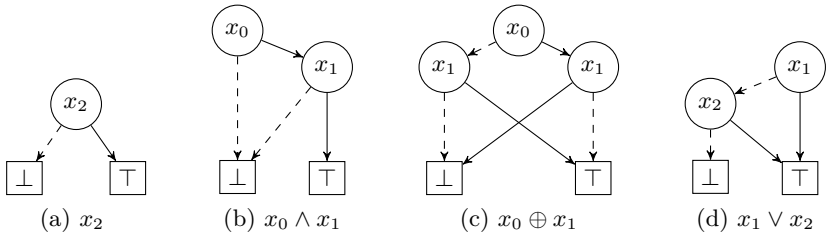
An I/O-algorithm takes inputs of size N, residing on the higher level of the two, i.e. in *external storage* (e.g. on a disk). The algorithm can only do computations on data that reside on the lower level, i.e. in *internal storage* (e.g. main memory). This internal storage can only hold a smaller and 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 we have that $N/B < \text{sort}(N) \ll N$, where $\text{sort}(N) \triangleq N/B \cdot \log_{M/B}(N/B)$ [1, 7] 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].

TPIE. The TPIE library [36] provides an implementation of I/O-efficient algorithms and data structures such that the use of B-sized buffers is completely transparent to the programmer. Elements can be stored in files that act like lists. One can **push** new elements to the end of a file and read the **next** elements from the file in either direction, 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. Furthermore, it provides an implementation of the I/O-efficient priority queue of [30] as developed in [29], 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 $\mathbb{B}^n \to \mathbb{B}$,



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

Fig. 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.

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

Common implementations of BDDs use recursive depth-first procedures that traverse the BDD and the unique nodes are managed through a hash table [9, 16, 20, 23, 34]. The latter allows one to directly incorporate the Reduce algorithm of [10] within each node lookup [9, 27]. They also use a memoisation table to minimise the number of duplicate computations [16, 23, 34]. 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, caused by looking up a node within the memoisation and following the low and high arcs [6, 21]. 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 [21]. This adds yet another $O(N)$ I/Os, where N is the number of nodes in the unreduced BDD.

Lars Arge provided in [5,6] a description of an Apply algorithm that is capable of only using $O(\mathrm{sort}(N_f \cdot N_g))$ I/Os and a Reduce that uses $O(\mathrm{sort}(N))$ I/Os

(see [6] for a detailed description). He also proved this to be optimal for both algorithms, assuming a levelwise ordering of nodes on disk [6]. Our algorithms, implemented in Adiar, differ from Arge's in subtle non-trivial ways. We will not elaborate further on his original proposal, since our algorithms are simpler and better at conveying the *time-forward processing* technique he used. Instead, we will mention where our Reduce and Apply algorithms differ from his.

3 BDD Manipulation 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 uids 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. 2.

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. $source$ and $target$ contain the level and identifier of internal nodes. 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. 3, our algorithms work in tandem by cycling between the node-based and arc-based representation.

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

Fig. 2: In-order representation of BDDs of Fig. 1

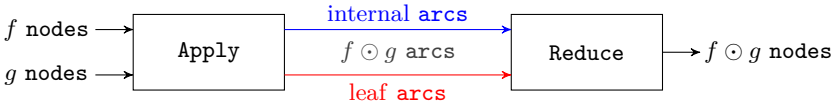
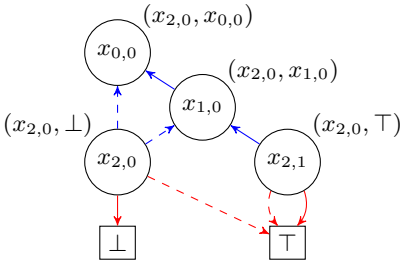


Fig. 3: The Apply–Reduce pipeline of our proposed algorithms



(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.

Fig. 4: Unreduced output of Apply when computing $x_2 \Rightarrow (x_0 \wedge x_1)$

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 their targets 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 when 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. 4.

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 a pair of input nodes v_f from f and v_g from g, a single node is created with label $\min(v_f.uid.label, v_g.uid.label)$ and recursion requests r_{low} and r_{high} are created for its two children. If the label of $v_f.uid$ and

```
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
```

Fig. 5: The Apply algorithm

$v_g.uid$ 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 pseudocode for the Apply procedure is shown in Fig. 5, where the **RequestsFor** function computes r_{low} and r_{high} for the pair of nodes (t_f, t_g). The goal of the rest of the algorithm is to obtain the information that **RequestsFor** needs in an I/O-efficient way. To this end, the two priority queues $Q_{app:1}$ and $Q_{app:2}$ are used to synchronise recursion requests for a pair of nodes (t_f, t_g) with the sequential order of reading nodes in f and g. $Q_{app:1}$ has elements of the form $(s \xrightarrow{is\_high} (t_f, t_g))$ and $Q_{app:2}$ has elements $(s \xrightarrow{is\_high} (t_f, t_g), low, high)$. The boolean *is_high* and the unique identifer s, being the request's origin, are used on lines 33 – 41, to output all ingoing arcs when the request is resolved.

Elements in $Q_{app:1}$ are sorted in ascending order by $\min(t_f, t_g)$, i.e. the node encountered first from f and g. Requests to the same (t_f, t_g) are grouped together by secondarily sorting the tuple lexicographically. $Q_{app:2}$ is sorted in ascending order by $\max(t_f, t_g)$, i.e. the second of the two to be visited, and ties are again broken lexicographically. This second priority queue is used in the case where $t_f.label = t_g.label$ but $t_f.id \neq t_g.id$, i.e. when both are needed to resolve the request but they are not necessarily available at the same time. To this end, the given request is moved from $Q_{app:1}$ into $Q_{app:2}$ on lines 19 – 23. Here, the request is extended with the unique identifiers *low* and *high* of $\min(v_f, v_g)$, which makes the children of $\min(v_f, v_g)$ available at $\max(v_f, v_g)$.

The next request to process from $Q_{app:1}$ or $Q_{app:2}$ is dictated by the **TopOf** function on line 10. In the case that both $Q_{app:1}$ and $Q_{app:2}$ are non-empty, let $r_1 = (s_1 \xrightarrow{is\_high_1} (t_{f:1}, t_{g:1}))$ be the top element of $Q_{app:1}$ and let the top element of $Q_{app:2}$ be $r_2 = (s_2 \xrightarrow{is\_high_2} (t_{f:2}, t_{g:2}), low, high)$. **TopOf**$(Q_{app:1}, Q_{app:2})$ returns $(r_1, \texttt{Nil}, \texttt{Nil})$ 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.

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 all nodes by the tuple (t_f, t_g), doubling the space used. 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 also did not forwarded a request's source s, 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 the source s, vital for the creation of the semi-transposed graph.

Proposition 1 (Following Arge 1996 [6]). *The Apply algorithm in Fig. 5 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.*

See the full paper [33] for the proof.

Pruning by shortcutting the operator The Apply procedure above, like Arge's original algorithm, follows recursion requests until a pair of leaves is met. Yet, for example in Fig. 4 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 subsequent Reduce will remove this node and its children in favour of the $\top$ leaf. Hence, the **RequestsFor** function can instead immediately create a request for the leaf. We implemented this in Adiar, since it considerably decreases the size of $Q_{app:1}$, $Q_{app:2}$, and of the output.

3.2 Reduce

Our Reduce algorithm in Fig. 6 works like other explicit variants with a single bottom-up sweep through the OBDD. Since the nodes are resolved and output in a bottom-up descending order, 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 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 the low and high arcs on lines 8 and 9. The base-cases for the Reduce algorithm are the arcs to leaves in F_{leaf}, which follow the exact same ordering. Hence, on lines 8 and 9, 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 26 – 32.

The algorithm otherwise proceeds similarly to the standard Reduce algorithm [10]. 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 a node in the given unreduced OBDD to its equivalent node in the output. $F_{j:1}$ contains the nodes v removed due to the first reduction rule and is populated on lines 7 – 12: if both children of v are the same then $[v.uid \mapsto v.low]$ is pushed to this file. $F_{j:2}$ contains the mappings for the second rule and is populated on lines 15 – 24. Nodes not placed in $F_{j:1}$ are placed in an intermediate file F_j and sorted by their children. This makes duplicate nodes immediate successors. Every unique node encountered in F_j is output to 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 it is unknown 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

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

Fig. 6: The Reduce algorithm

in both $F_{j:1}$ and $F_{j:2}$ to their parents on lines 26 – 32. 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 internally uses node-based auxiliary data structures, his Reduce algorithm had to create two copies of the input to reverse all internal arcs: a copy sorted by the nodes' low child and one sorted by their high children. Since $F_{internal}$ already has its arcs reversed, our design eliminates two expensive sorting steps and more than halves the memory used.

Another consequence of Arge's 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}. The semi-transposed input allows us to decrease the number of I/Os due to Q_{red} by $\Theta(\text{sort}(N_\ell))$ where N_ℓ are the number of arcs to leaves (see [33] for the proof). In practice, together with pruning the recursion during Apply, this can provide up to a factor 2 speedup [33].

Proposition 2 (Following Arge 1996 [6]). *The Reduce algorithm in Fig. 6 has an $O(sort(N))$ I/O complexity and an $O(N \log N)$ time complexity.*

See the full paper [33] for the proof. Arge proved in [6] that this $O(\text{sort}(N))$ I/O complexity is optimal for the input, assuming a levelwise ordering of nodes.

3.3 Other BDD Algorithms

By applying the above algorithmic techniques, one can obtain all other singly-recursive BDD algorithms; see [33] for the details. We now design asymptotically better variants of Negation and Equality Checking than what is possible by deriving them using Apply.

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

This is an improvement over the $O(\text{sort}(N))$ I/Os spent by Apply to compute $f \oplus \top$, where $\oplus$ is exclusive or. Furthermore, disk space is shared between BDDs.

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 the number of nodes, number of levels, or the label or size of each of the L levels do not match. This can be checked in $O(1)$ and $O(L/B)$ I/Os if the Reduce algorithm in Fig. 6 is made to also output the relevant meta-information.

If $f \equiv g$, the isomorphism relates the roots of the BDDs for f and g. 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, one can check for equality by traversing the product of both BDDs (as in Apply) and check for one of the following two conditions being violated.

- The children of the given recursion request (t_f, t_g) should either both be the same leaf or an internal node with the same label.
- On level i, exactly N_i unique recursion requests should be retrieved from the priority queues, where N_i are the number of nodes on level i.

If the first condition is never violated, it is guaranteed that $f \equiv g$, and so $\top$ is output. The second ensures that the algorithm terminates earlier on negative cases and lowers the provable complexity bound; see [33] for the proof.

Proposition 4. *Equality Checking has I/O complexity $O(sort(N))$ and time complexity $O(N \log N)$, where $N = \min(N_f, N_g)$ is the minimum of the respective sizes of the BDDs for f and g.*

If (1) on page 5 is extended such that $\bot, \top$ succeed all unique identifiers and $\bot < \top$, then Fig. 6 actually enforces a much stricter ordering; it outputs nodes in an order purely based on their label and the unique identifier of their children.

Proposition 5. *If G_f and G_g are outputs of Reduce in Fig. 6, then $f \equiv g$ if and only if the ith nodes of G_f and G_g match numerically.*

See the full paper [33] for the proof. The negation operation breaks this property by changing the leaf values without changing their order. So, in the case where f or g, but not both, have their negation flag set, one still has to use the $O(\text{sort}(N))$ algorithm above, but otherwise a simple linear scan of both BDDs suffices.

Corollary 1. *If the negation flag of the BDDs for f and g are equal, then Equality Checking can be done in $2 \cdot N/B$ I/Os and $O(N)$ time, where $N = \min(N_f, N_g)$ is the minimum of the respective sizes of the BDDs for f and g.*

Both Proposition 4 and Corollary 1 are an asymptotic improvement on the $O(\text{sort}(N^2))$ equality checking algorithm by computing $f \leftrightarrow g$ with Apply and Reduce and then test whether the output is the $\top$ leaf.

4 Adiar: An Implementation

The algorithms and data structures described in Section 3 have been implemented in a new BDD package, named Adiar[1, 2]. The most important operations are shown in Table 1. 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.

Adiar is initialised with 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, e.g. a dedicated harddisk. The BDD package is deinitialised by calling the `adiar_deinit()` function.

The `bdd` object in Adiar is a container for the underlying files for each BDD, while a `__bdd` object is used for possibly unreduced arc-based OBDDs. Reference counting on the underlying files is used to reuse the same files and to immediately delete them when the reference count decrements to 0. Files are deleted as early as possible by use of implicit conversions between the `bdd` and `__bdd` objects and an overloaded assignment operator, making the concurrently occupied space on disk minimal.

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

[2] Source code is publicly available at github.com/ssoelvsten/adiar

Adiar function	Operation	I/O complexity	Justification
`bdd_apply(`f`,`g`,`$\odot$`)`	$f \odot g$	$O(\text{sort}(N_f N_g))$	Prop. 1, 2
`bdd_ite(`f`,`g`,`h`)`	$f \ ? \ g \ : \ h$	$O(\text{sort}(N_f N_g N_h))$	[33], Prop. 2
`bdd_restrict(`f`,`i`,`v`)`	$f\|_{x_i=v}$	$O(\text{sort}(N_f))$	[33], Prop. 2
`bdd_exists(`f`,`i`)`	$\exists v : f\|_{x_i=v}$	$O(\text{sort}(N_f^2))$	[33], Prop. 2
`bdd_forall(`f`,`i`)`	$\forall v : f\|_{x_i=v}$	$O(\text{sort}(N_f^2))$	[33], Prop. 2
`bdd_not(`f`)`	$\neg f$	$O(1)$	Prop. 3
`bdd_satcount(`f`)`	$\#x : f(x)$	$O(\text{sort}(N_f))$	[33]
`bdd_nodecount(`f`)`	N_f	$O(1)$	Section 3.3
f `==` g	$f \equiv g$	$O(\text{sort}(\min(N_f, N_g)))$	Prop. 4

Table 1: Some of the operations supported by Adiar and their I/O-complexity.

5 Experimental Evaluation

While time-forwarding may be an asymptotic improvement over the recursive approach in the I/O-model, its usability in practice is another question entirely. We have compared *Adiar* 1.0.1 to the recursive BDD packages *CUDD* 3.0.0 [34] and *Sylvan* 1.5.0 [16] (in single-core mode). We constructed BDDs for some benchmarks in all tools in a similar manner, ensuring the same variable ordering.

The experimental results[3] were obtained on server nodes of the *Grendel* cluster at the Centre for Scientific Computing Aarhus. Each node has two 48-core 3.0 GHz Intel Xeon Gold 6248R processors, 384 GiB of RAM, 3.5 TiB of available SSD disk, run CentOS Linux, and compile code with GCC 10.1.0. We report the *minimum* measured running time, since it minimises any error caused by the CPU, memory and disk [13]; using the average or median does not significantly change any of our results. For comparability all compute nodes are 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, given a ratio between the node table and the cache of 64:1 and set to start its data structures 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. The size of the CUDD cache was set such it would have the same node table to cache ratio when reaching 300 GiB.

5.1 Queens

The solution to the 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 [22]: the variable x_{ij} represents whether a queen is placed on the ith row and the jth column and the solution to the problem then corresponds to the number of satisfying assignments to the formula $\bigwedge_{i=0}^{N-1} \bigvee_{j=0}^{N-1} (x_{ij} \wedge \neg \mathit{has\_threat}(i,j))$, 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).

[3] Available at Zenodo [32] and at github.com/ssoelvsten/bdd-benchmark

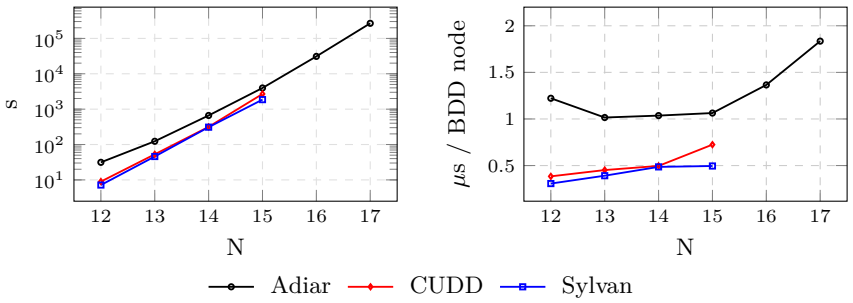


Fig. 7: Running time solving N-Queens (lower is better).

The ROBDD of the innermost conjunction can be directly constructed, without any BDD operations.

The current version of Adiar is implemented purely using external memory algorithms. These perform poorly when given small amounts of data. Hence, it is not meaningful to compare performance for $N < 12$ where the BDDs involved are 23.5 MiB or smaller. For $N \geq 12$, Fig. 7 shows how the gap in running time between Adiar and other BDD packages shrinks as instances grow. At $N = 15$, which is the largest instance solved by Sylvan and CUDD, Adiar is 1.47 times slower than CUDD and 2.15 times slower than Sylvan.

The largest instance solved by Adiar is $N = 17$ where the largest BDD constructed is 719 GiB in size. In contrast, Sylvan only constructed a 12.9 GiB sized BDD for $N = 15$. Even though Adiar has to use disk, it only becomes 1.8 times slower per processed node compared to its highest performance at $N = 13$. Conversely, Adiar is able to solve the $N = 15$ problem with much less main memory than both Sylvan and CUDD. Fig. 8 shows the running time on the same machine with its memory, including its file system cache, limited with *cgroups* to be 1 GiB more than given to the BDD package. Yet, Adiar is only 1.39 times slower when decreasing its memory down to 2 GiB, while Sylvan cannot function with less than 56 GiB of memory available.

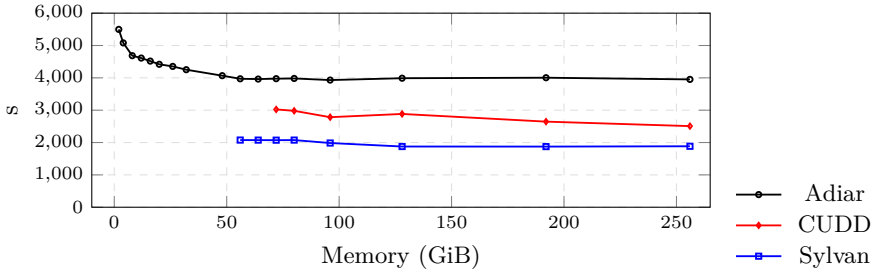


Fig. 8: Running time of 15-Queens with variable memory (lower is better).

We also ran experiments on counting the number of draw positions in a 3D-version of Tic-Tac-Toe, derived from [22]. Our results [33] paint a similar picture: Adiar is only 2.50 times slower than Sylvan for Sylvan's largest solved instance; Sylvan only creates BDDs of up to 34.4 GiB in size, whereas Adiar constructs a 902 GiB sized BDD; Adiar only slows down by a factor of 2.49 per processed node when using the disk extensively to solve the larger instances.

5.2 Combinatorial Circuit Verification

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 is distributed together with one circuit optimised for size C_s and one circuit optimised for depth C_d. The last three are the *More than a Million Gates* benchmarks, which we will ignore as they come without optimised versions.

Based on the approach of the *Nanotrav* program as distributed with CUDD, we verify the functional equivalence between each output gate of C_o and the corresponding gate in each optimised circuits C_d, and C_s. The BDDs are computed by representing every input gate by a decision variable, and computing the BDD of all other gates from the BDDs of their input wires. Finally, the BDDs for every pair of corresponding output gates are tested for equality. Memoisation ensures that the same gate is not computed twice, while a reference counter is maintained for each gate such that dead BDDs in the memoisation table may be garbage collected. Recall that Adiar stores each BDD in a separate file, while Sylvan and CUDD share nodes between different BDDs in a forest.

Table 2 shows the number of verified instances with each BDD package within a 15 days time limit. Adiar is able to verify three more benchmarks than both other BDD packages. This is despite the fact that most instances include hundreds of concurrent BDDs, while the disk is only 12 times larger than main memory. For example, the largest verified benchmark, *mem_ctrl*, has up to 1231 BDDs existing at the same time.

Table 3 shows the time it took Adiar to verify equality between the original and each of the optimised circuits, for the three largest cases verified. The table also shows the sum of the sizes of the output BDDs that represent each circuit. Throughout all solved benchmarks, equality checking took less than 1.47% of the total construction time and the $O(N/B)$ algorithm could be used in 71.6% of all BDD comparisons. The *voter* benchmark with its single output shows that

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

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

	depth	size
Time (s)	5862	5868
$O(\text{sort}(N))$	496	476
$O(N/B)$	735	755
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 (MiB)	3589	

(b) *sin*

	depth	size
Time (s)	0.058	0.006
$O(\text{sort}(N))$	1	0
$O(N/B)$	0	1
N (MiB)	5.74	

(c) *voter*

Table 3: 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 was used.

the $O(N/B)$ algorithm is about 10 times faster than the $O(\text{sort}(N))$ algorithm and can compare at least $2 \cdot 5.75$ MiB/0.006 s = 1.86 GiB/s.

6 Conclusions and Future Work

Adiar provides an I/O-efficient implementation of BDDs. The iterative BDD algorithms exploit a topological ordering of the BDD nodes in external memory, by use of priority queues and sorting algorithms. All recursion requests for a single node are processed together, eliminating the need for a memoisation table.

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 – for BDDs larger than a few GiB the use of Adiar has at most resulted in a 3.69 factor slowdown. 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 compared to use of swap memory with conventional BDD packages.

This performance comes at the cost of Adiar not being able to share nodes between BDDs. Yet, this increase in space usage is not a problem in practice and it makes garbage collection a trivial and cheap deletion of files on disk. On the other hand, the lack of sharing makes it impossible to check for functional equivalence with a mere pointer comparison. Instead, one has to explicitly check for the two DAGS being isomorphic. We have improved the asymptotic and practical performance of equality checking such that it is negligible in practice.

This lays the foundation on which we intend to develop external memory versions of the BDD algorithms that are still missing for symbolic model checking. Specifically, we intend to improve the performance of quantifying multiple variables and designing a relational product operation. Furthermore, we will improve performance for small instances that fit entirely into internal memory.

Acknowledgements

Thanks to the late Lars Arge, to Gerth S. Brodal, and to Mathias Rav for their inputs. Furthermore, thanks to the Centre for Scientific Computing Aarhus (phys.au.dk/forskning/cscaa/) for running our experiments on their cluster.

References

1. Aggarwal, A., Vitter, Jeffrey, 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. Amarú, L., Gaillardon, P.E., De Micheli, G.: The EPFL combinational benchmark suite. In: 24th International Workshop on Logic & Synthesis (2015)
3. 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
4. Arge, L.: The buffer tree: A new technique for optimal I/O-algorithms. In: Workshop on Algorithms and Data Structures (WADS). Lecture Notes in Computer Science, vol. 955, pp. 334–345. Springer, Berlin, Heidelberg (1995). https://doi.org/10.1007/3-540-60220-8_74
5. Arge, L.: The I/O-complexity of ordered binary-decision diagram manipulation. In: 6th International Symposium on Algorithms and Computations (ISAAC). Lecture Notes in Computer Science, vol. 1004, pp. 82–91 (1995). https://doi.org/10.1007/BFb0015411
6. 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
7. Arge, L.: External geometric data structures. In: 10th International Computing and Combinatorics Conference (COCOON). Lecture Notes in Computer Science, vol. 3106 (2004). https://doi.org/10.1007/978-3-540-27798-9_1
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. 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
10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
11. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (1992). https://doi.org/10.1145/136035.136043
12. Bryant, R.E.: Binary decision diagrams. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 191–217. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
13. Chen, J., Revels, J.: Robust benchmarking in noisy environments. arXiv (2016), https://arxiv.org/abs/1608.04295
14. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
15. 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

16. 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
17. 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
18. He, L., Liu, G.: Petri net based symbolic model checking for computation tree logic of knowledge. arXiv (2020), https://arxiv.org/abs/2012.10126
19. 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
20. Karplus, K.: Representing Boolean functions with if-then-else DAGs. Tech. rep., University of California at Santa Cruz, USA (1988)
21. Klarlund, N., Rauhe, T.: BDD algorithms and cache misses. In: BRICS Report Series. vol. 26 (1996). https://doi.org/10.7146/brics.v3i26.20007
22. Kunkle, D., Slavici, V., Cooperman, G.: Parallel disk-based computation for large, monolithic binary decision diagrams. In: 4th International Workshop on Parallel Symbolic Computation (PASCO). pp. 63–72 (2010). https://doi.org/10.1145/1837210.1837222
23. Lind-Nielsen, J.: BuDDy: A binary decision diagram package. Tech. rep., Department of Information Technology, Technical University of Denmark (1999)
24. 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
25. 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)
26. 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
27. 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
28. 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
29. Petersen, L.H.: External Priority Queues in Practice. Master's thesis, Department of Computer Science, University of Aarhus (2007)
30. Sanders, P.: Fast priority queues for cached memory. ACM Journal of Experimental Algorithmics **5**, 7–32 (2000). https://doi.org/10.1145/351827.384249
31. 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
32. Sølvsten, S.C., Van de Pol, J.: Adiar v1.0.1 : TACAS 2022 artifact. Zenodo (2021). https://doi.org/10.5281/zenodo.5638335

33. Sølvsten, S.C., Van de Pol, J., Jakobsen, A.B., Thomasen, M.W.B.: Efficient binary decision diagram manipulation in external memory. arXiv (2021), https://arxiv.org/abs/2104.12101
34. Somenzi, F.: CUDD: CU decision diagram package, 3.0. Tech. rep., University of Colorado at Boulder (2015)
35. Velev, M.N., Gao, P.: Efficient parallel GPU algorithms for BDD manipulation. In: 19th Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 750–755. IEEE Computer Society Press (2014). https://doi.org/10.1109/ASPDAC.2014.6742980
36. Vengroff, D.E.: A Transparent Parallel I/O Environment. In: DAGS Symposium on Parallel Computation. pp. 117–134 (1994)