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

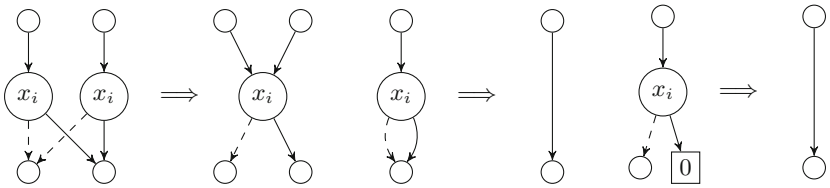
github.com/ssoelvsten/adiar

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.

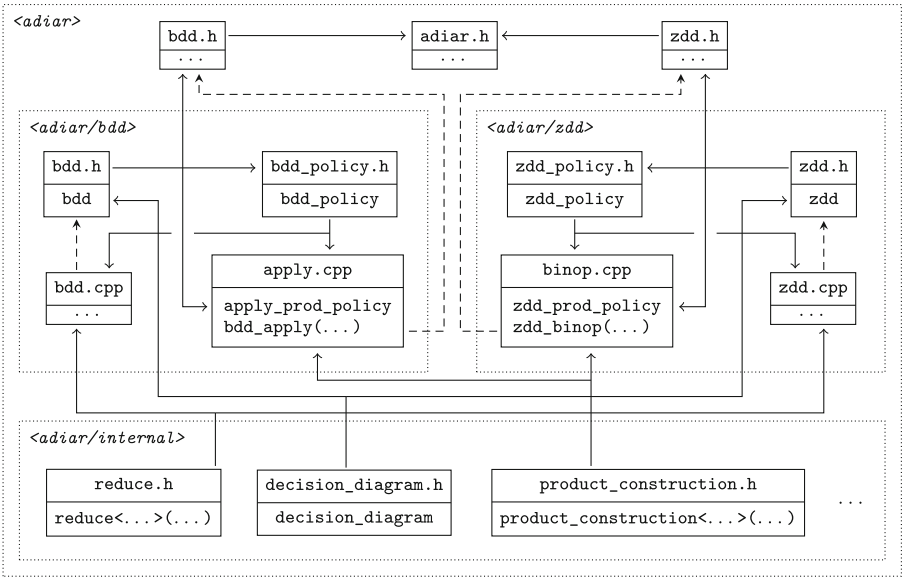


Fig. 2. Architecture of Adiar v1.1. Solid lines are direct inclusions of one file in another while dashed lines represent the implementation of a declared function.

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

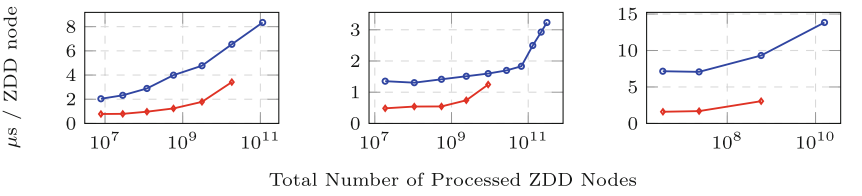
(b) Tic-Tac-Toe

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 6×6. From that point, the computation time and size of the ZDDs quickly explode. Adiar solved up to the 6×7 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 multi-threading. 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