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

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 Section 3 how Adiar can exploit the width of the input BDDs (defined in Section 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 Section 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.

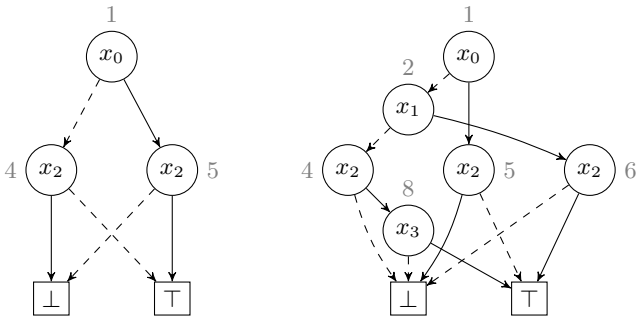
CAL [20] (based on [6, 17]) is to the best of our knowledge the only other BDD package to compute on BDDs that exceed main memory. To do so, it stores all BDD nodes in a unique node table and uses queues to execute its algorithms in the breadth-first manner. These node tables and queues can be offloaded 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 needs to use $\text{sort}(N) \triangleq N/B \cdot \log_{M/B}(N/B)$ I/Os. Furthermore, one can also



(a) $x_0 \equiv x_2$ (b) $(x_0 \oplus x_2) \wedge (x_0 \vee x_1 \vee x_3)$

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.

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 Section 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

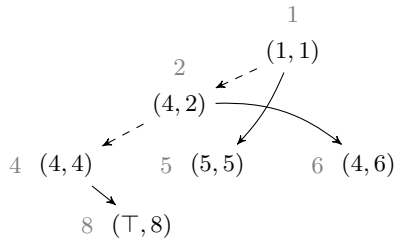


Fig. 2: Product Graph for BDDs in Fig. 1a and Fig. 1b.

in 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.

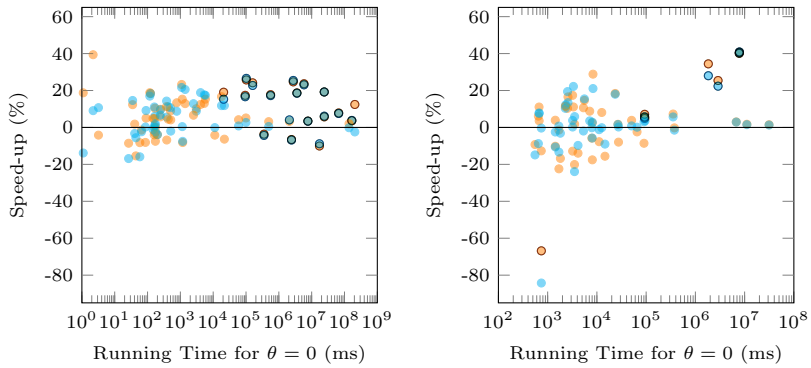
4 Experimental Evaluation

We have implemented the modified algorithm of Section 3 and run the benchmarks from [22, 24] with threshold $\theta = 0$, B (2 MiB), and ∞. 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 θ 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.



(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.

Of the 64 combinatorial instances, 14 (21.9% of all of these instances) had 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.

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%

Simultaneously, this also improves performance for instances not solvable with CUDD. For example, it improves the solving time of 16-Queens by 19.2%.

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 θ 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 θ 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 Section 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. 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. 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
4. 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
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 Transactions on Computers **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. pp. 139—-149. SODA '95, Society for Industrial and Applied Mathematics (1995)
9. Elgaard, J., Klarlund, N., Møller, A.: MONA 1.x: new techniques for WS1S and WS2S. In: Proc. 10th International Conference on Computer-Aided Verification, CAV '98. LNCS, vol. 1427, pp. 516–520. Springer-Verlag (06 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). 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
11. Klarlund, N.: Mona & Fido: The logic-automaton connection in practice. In: Computer Science Logic. LNCS, vol. 1414, pp. 311–326 (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). Lecture Notes in Computer Science, vol. 13994, pp. 353–371. Springer (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, Berlin, 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: Application and Theory of Petri Nets. pp. 416–435. Springer (1994). https://doi.org/0.1007/3-540-58152-9_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. Lecture Notes in Computer Science, vol. 14216, pp. 72–92. Springer (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 13903, Springer, Berlin, 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 (2021), https://arxiv.org/abs/2104.12101
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. Lecture Notes in Computer Science, vol. 13244, pp. 295–313. Springer (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 (01 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. Sølvsten, S.C.: BDD Benchmark. Zenodo (2024). https://doi.org/10.5281/zenodo.10803154