Predicting Memory Demands of BDD Operations Using Maximum Graph Cuts

Steffan Christ Sølvsten(✉) and Jaco van de Pol

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

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

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

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

Keywords: Binary Decision Diagrams · Directed Acyclic Graphs · Maximum Graph Cuts · External Memory Algorithms

1 Introduction

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

Adiar [38] is a redesign of the classical BDD algorithms such that they are optimal in the I/O model of Aggarwal and Vitter [1], based on ideas from Lars Arge [4]. As shown in Fig. 1, this enables Adiar to handle BDDs beyond the limits of main memory with only a minor slowdown in performance, unlike conventional BDD implementations. Adiar is implemented on top of the TPIE library [28,41], which provides external memory sorting algorithms, file access, and priority

É. André and J. Sun (Eds.): ATVA 2023, LNCS 14216, pp. 72–92, 2023.
https://doi.org/10.1007/978-3-031-45332-8_4

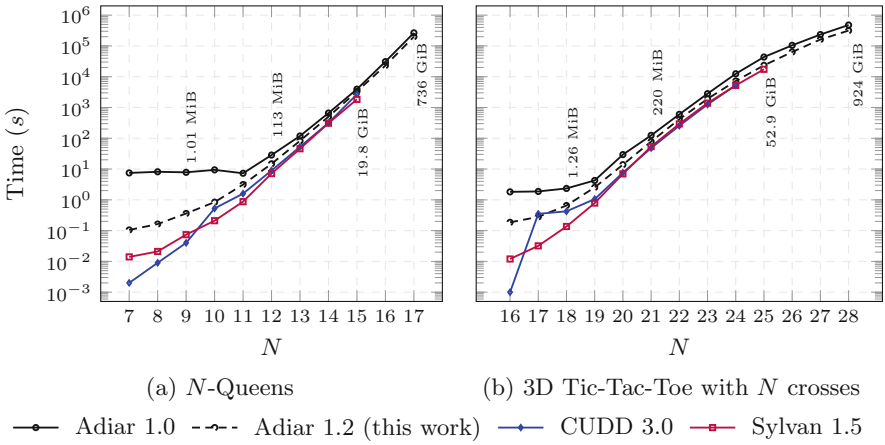


Fig. 1. Running time solving combinatorial BDD benchmarks. Some instances are labelled with the size of the largest BDD constructed to solve them.

queues. These external memory data structures work by loading one or more blocks from files on disk into internal memory and manipulating the elements within these blocks before storing them again on disk. Their I/O-efficiency stems from a carefully designed order in which these blocks are retrieved, manipulated, and stored. Yet, initializing the internal memory in preparation to do so is itself costly. This is evident in Fig. 1 (cf. Sect. 4.3 for more details) where Adiar's performance is several orders of magnitude worse than conventional BDD packages for smaller instance sizes. In fact, Adiar's performance decreases when the amount of internal memory increases.

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

The linearithmic I/O- and time-complexity of Adiar's algorithms also applies to the lower levels of the memory hierarchy, i.e. between the cache and RAM. Hence, there is no reason to believe that the bad performance for smaller instances is inherently due to the algorithms themselves; if they used an internal memory variant of all auxiliary data structures, then Adiar ought to perform well for much smaller instances.

We argue that simple solutions are unsatisfactory: A first idea would be to start running classical, depth-first BDD algorithms until main memory is

exhausted. In that case, the computation is aborted and restarted with external memory algorithms. But, this strategy doubles the running time. While it would work well for small instances, the slowdown for large instances would be unacceptable. Alternatively, both variants could be run in parallel. But, this would halve the amount of available memory and again slow down large instances.

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

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

1.1 Contributions

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

Investigating the structure of BDDs from the perspective of i-level cuts for $i \in \{1, 2\}$ in Sect. 3.1 and 3.2, we obtain sound upper bounds on the maximum i-level cuts of a BDD operation's output, purely based on the maximum i-level cut of its inputs. Using these upper bounds, Adiar can decide in constant time whether to run the next algorithm with internal or external memory data structures. Here, only one variant is run, all memory is dedicated to it, and the exact type of the auxiliary data structures are available to the compiler.

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

moderate instances of the EPFL combinational benchmark circuits [2], the use of i-level cuts decreases the running time from 56.5 h down to 4.0 h.

2 Preliminaries

2.1 Graph and Cuts

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

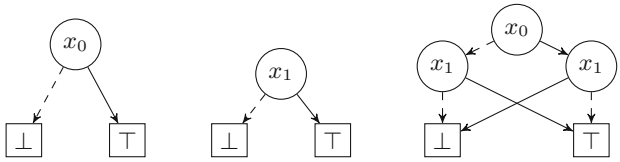
A cut of a directed graph (V, A) is a partitioning (S, T) of V such that $S \cup T = V$ and $S \cap T = \emptyset$. Given a weight function $w : A \to \mathbb{R}$ the weighted maximum cut problem is to find a cut (S, T) such that $\sum_{a \in S \times T \cap A} w(a)$ is maximal, i.e. where the total weight of arcs crossing from some vertex in S to one in T is maximised. Without decreasing the weight of a cut, one may assume that all sources in V are part of the partition S and all sinks are part of T. The maximum cut problem is NP-complete for directed graphs [30] and restricting the problem to directed acyclic graphs (DAGs) does not decrease the problem's complexity [22].

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

2.2 Binary Decision Diagrams

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

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



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

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

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

Zero-Suppressed Decision Diagrams. A Zero-suppressed Decision Diagram (ZDD) [26] is a variation of BDDs where the first reduction rule is changed: a node v for the variable $\mathit{label}(v)$ with $\mathit{out}(v) = \{v \xrightarrow{\bot} v', v \xrightarrow{\top} v''\}$ is not suppressed if v is a *don't care* node, i.e. if $v' = v''$, but rather if it assigns the variable $\mathit{label}(v)$ to $\bot$, i.e. if $v'' = \bot$. This makes ZDDs a better choice in practice than BDDs to represent functions f where its on-set, $\{\boldsymbol{x} \mid f(\boldsymbol{x}) = \top\}$, is sparse.

The basic notions behind the BDD algorithms persist when translated to ZDDs, but it is important for correctness that the ZDD operations account for the shape of the suppressed nodes. For example, the *union* operation needs to replace recursion requests for (v_f, v_g) with $(v_f, \bot)$ if $\mathit{label}(v_f) < \mathit{label}(v_g)$ and with $(\bot, v_g)$ if $\mathit{label}(v_f) > \mathit{label}(v_g)$.

Levelised Algorithms in Adiar. BDDs and ZDDs are usually manipulated with recursive algorithms that use two hash tables: one for memoisation and another to enforce the two reduction rules [7,27]. Lars Arge noted in [4,5] that this approach is not efficient in the I/O-model of Aggarwal and Vitter [1]. He proposed to address this issue by processing all BDDs iteratively level by level with the time-forward processing technique [13,24]: recursive calls are not executed at the time of issuing the request but are instead deferred with one or more priority queues until the necessary elements are encountered in the inputs.

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

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

In [38], we implemented this approach in the BDD package Adiar. Furthermore, with version 1.1 we have extended this approach to ZDDs [34].

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

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

For BDD algorithms that produce an output BDD, e.g. the Apply algorithm, Adiar first constructs (V', A') level by level. When the output BDD node $t \in V'$ is created from nodes $v_f \in V_f$ and $v_g \in V_g$, the top of the priority queues provides all ingoing arcs, which are placed in the output. Outgoing arcs to a terminal, $out(t) \cap (V' \times \mathbb{B} \times \mathbb{B})$, are also immediately placed in a separate output. On the other hand, recursion requests from t to its yet unresolved non-terminal children, $out(t) \backslash (V' \times \mathbb{B} \times \mathbb{B})$, have to be processed later. To do so, these unresolved arcs are put back into the priority queue as arcs $(t \xrightarrow{b} (v'_f, v'_g)) \in V' \times \mathbb{B} \times (V_f \times V_g)$ where the arc's target is the tuple of input nodes $v'_f \in V_f$ and $v'_g \in V_g$. This essentially makes the priority queue contain all the yet unresolved arcs of the output. For example, when using Apply to produce Fig. 2c from Fig. 2a and 2b, the root node of the output is resolved to have uid $(0,0)$ and the priority queue contains arcs $(0,0) \xrightarrow{\bot} (\bot, (1,0))$ and $(0,0) \xrightarrow{\top} (\top, (1,0))$. Both of these arcs are then later resolved, creating the nodes $(1,0)$ and $(1,1)$, respectively.

Yet, these *top-down sweeps* of Adiar produce sequences of arcs rather than nodes. Furthermore, the DAG (V', A') is not necessarily a reduced OBDD. Hence, as shown in Fig. 4, Adiar follows up on the above top-down sweep with

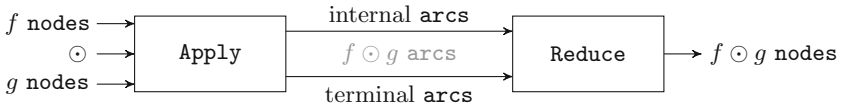


Fig. 4. The Apply–Reduce pipeline in Adiar

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

3 Levelised Cuts of a Directed Acyclic Graph

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

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

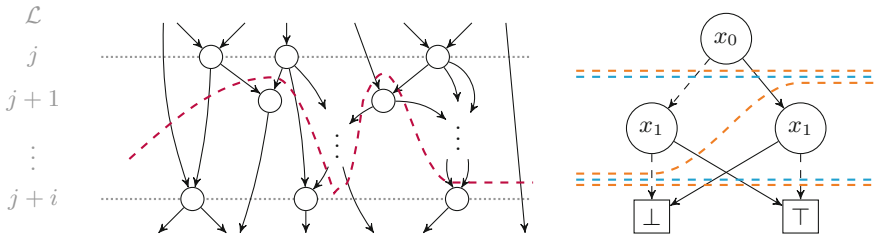
Intuitively, $\mathcal{L}$ is a labeling of vertices $v \in V$ that respects a topological ordering of V. Since (V, A) is a DAG, such a topological ordering always exists and hence such an $\mathcal{L}$ must also always exist. Specifically, let π_V in be the longest path in (V, A) and π_v be the longest path of any given $v \in V$ to any sink $t \in V$, then $\mathcal{L}(v)$ can be defined to be the difference of their lengths, i.e. $|\pi_V| - |\pi_v|$.

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

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

As will become apparent later, deriving the i-level cut with maximum weight for $i \in \{1, 2\}$ will be of special interest. Figure 5b shows two 1-level cuts and three 2-level cuts in the BDD for the exclusive-or of the two variables x_0 and x_1. A 1-level cut is by definition a cut between two adjacent levels whereas a 2-level cut allows nodes on level $j + 1$ to be either in S or in T. In Fig. 5b, both the maximum 1-level and 2-level cuts have size 4.

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



(a) Definition of an i-level cut. (b) Example of cuts in a BDD.

Fig. 5. Visualization of i-level (purple), 1-level (cyan) and 2-level (orange) cuts. (Color figure online)

Proof. For a specific $j \in \mathcal{L}(V)$ we can compute the size of the 1-level cut at j in $O(A)$ time by computing the sum of $w((s,t))$ over all arcs $(s,t) \in A$ where $\mathcal{L}(s) \leq j$ and $\mathcal{L}(t) > j$. This cut is by definition unique for j and hence maximal. Repeating this for each $j \in \mathcal{L}(V)$ we obtain the maximum 1-level cut of the entire DAG in $O(|\mathcal{L}(V)| \cdot |A|) = O(|V| \cdot |A|)$ time. □

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

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

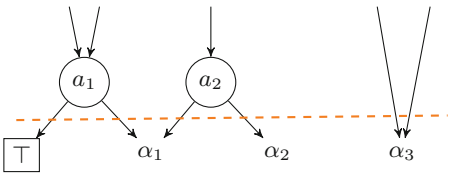
Lampis, Kaouri, and Mitsou [22] prove NP-completeness for computing the maximum cut of a DAG by a reduction from the *not-all-equal SAT problem* (NAE3SAT) to a DAG with 5 levels. That is, they prove NP-completeness for computing the size of the maximum i-level cut for $i \geq 4$. This still leaves the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

3.1 Maximum Levelised Cuts in BDD Manipulation

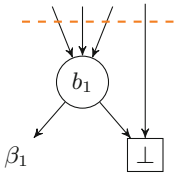
For an OBDD, represented by the DAG (V, A), we will consider the levelisation function $\mathcal{L}_{OBDD}$ where all nodes with the same label are on the same level.

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

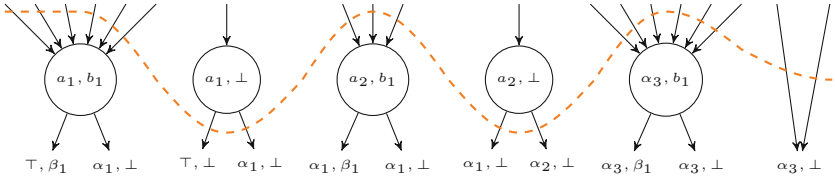
For a BDD f with the DAG (V, A), let $N_f \triangleq |V \backslash \mathbb{B}|$ be the number of internal nodes in V. Let $C_{i:f}$ denote the size of the unweighted maximum i-level cut in (V, A); in Sect. 3.2 we will consider weighted maximum cuts, where one or more terminals are ignored. Finally, we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



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



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



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

Fig. 6. Relation between the maximal 2-level cut of two BDDs' internal arcs and the maximum 2-level cut of their product.

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

Proof. This is proven via the stronger statement that the maximum cut of a multi-rooted decision diagram is less than or equal to $N + r$ where N is the number of internal nodes and $r \geq 1$ is the number of roots. This, in turn, is done by induction on the number of internal nodes N (See the full paper [37]). □

This bound is tight for i-level cuts, as is evident from Fig. 5b where the size of the maximum (i-level) cut is 4. Yet, in general, one can obtain a better upper bound on the maximum i-level cut of the (unreduced) output of each BDD operation when the maximum i-level cut of the input is known.

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

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

Every node of the output represents a tuple (v_f, v_g) where v_f, resp. v_g, is an internal node of f, resp. g, or is one of the terminals $\mathbb{B} = \{\bot, \top\}$. An example of this situation is shown in Fig. 6. The node (v_f, v_g) contributes with $\max(|in((v_f, v_g))|, |out((v_f, v_g))|)$ to the maximum 2-level cut at that level. Since it is a BDD node, $|out((v_f, v_g))| = 2$. We have that $|in((v_f, v_g))| \leq |in(v_f)| \cdot |in(v_g)|$ since all combinations of in-going arcs may potentially exist and lead to this product of v_f and v_g. Expanding on this, we obtain

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

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

The bounds in Theorem 2 are better than what can be derived from Theorem 1 since $C_{i:f}$ and $C_{i:g}$ are themselves cuts and hence their product must be at most the bound based on the possible number of nodes. They are also tight: the maximum i-level cut for $i \in \{1, 2\}$ of the BDDs for the variables x_0 and x_1 in Fig. 2a and 2b both have size 2 while the BDD for the exclusive-or of them in Fig. 2c has, as shown in Fig. 5b, a maximum i-level cut of size 4.

Theorem 2 is of course only an over-approximation. The gap between the upper bound and the actual maximum i-level cut arises because Theorem 2 does not account for pairs (v_f, v_g), where node v_f sits above f's maximum 2-level cut and v_g sits below g's maximum 2-level cut, and vice versa. In this case, outgoing arcs of v_f are paired with ingoing arcs of v_g, even though this would be strictly larger than the arcs of their product. Furthermore, similar to Theorem 1, this bound does not account for arcs that cannot be paired as they reflect conflicting assignments to one or more input variables. For example, in the case where the out-degree is greater for both nodes, the above bound mistakenly pairs the low arcs with the high arcs and vice versa.

3.2 Improving Bounds by Accounting for Terminal Arcs

Some of the imprecision in the over-approximation of Theorem 2 highlighted above can partially be addressed by explicitly accounting for the arcs to each terminal. For $B \subseteq \mathbb{B}$, let w_B be the weight function that only cares for arcs to internal BDD nodes and to the terminals in B.

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

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

The constant hidden within the $O(|V| \cdot |A|)$ running time of the algorithm in the proof of Proposition 1 is smaller than the one in the proof of Proposition 2. Hence, the following slight over-approximation of $C^B_{2:f}$ given $C^B_{1:f}$ may be useful (proved in the full paper [37]).

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

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

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

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

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

3.3 Maximum Levelised Cuts in ZDD Manipulation

The results in Sect. 3.1 and 3.2 are loosely yet subtly coupled to the reduction rules of BDDs. Specifically, Theorem 1 is applicable to ZDDs as-is but Theorem 2 and its derivatives provide unsound bounds for ZDDs. This is due to the fact that, unlike for BDDs, a suppressed ZDD node may re-emerge during a ZDD product construction algorithm. For example in the case of the *union* operation, when processing a pair of nodes with two different levels, its high child becomes the product of a node v in one ZDD and the $\bot$ terminal in the other – even if there was no arc to $\bot$ in the original two cuts for f and g.

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

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

3.4 Adding Levelised Cuts to Adiar's Algorithms

The description of Adiar in Sect. 2.2 leads to the following observations.

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

Specifically, the priority queues always contain an i-level cut (S, T), where S is the set of processed diagram nodes and T is the set of yet unresolved diagram nodes. For example, the 2-level cuts depicted in Fig. 5b reflect the states of the top-down priority queue within the Apply to compute the exclusive-or of Fig. 2a and 2b to create Fig. 2c. In turn, the 1-level cuts in Fig. 5b are also the states of the bottom-up priority queue of the Reduce sweep that follows.

Hence, the upper bounds on the 1 and 2-level cuts in Sect. 3.1, 3.2, and 3.3 are also upper bounds on the size of all auxiliary data structures. That is, upper bounds on the i-level cuts of the input can be used to derive a sound guarantee of whether the much faster internal memory variants can fit into memory. To only add a minimal overhead to the performance, computing these i-level cuts should

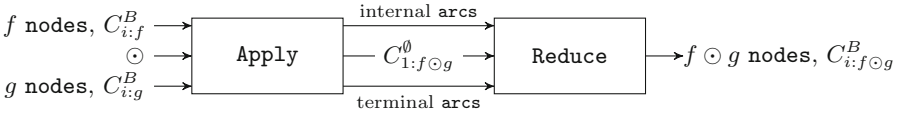
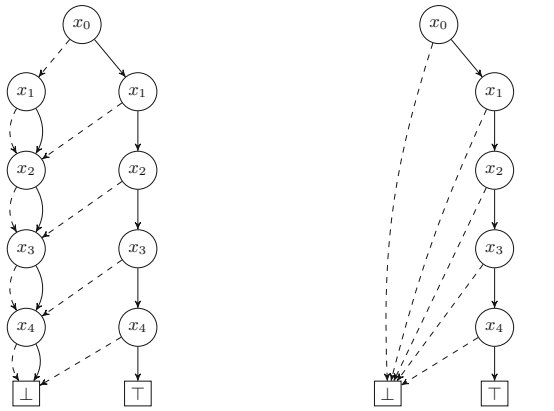


Fig. 7. The Apply–Reduce pipeline in Adiar with i-level cuts.



(a) Unreduced OBDD (b) Reduced OBDD

Fig. 8. Example of reduction increasing the 1 and 2-level maximum cut.

be done as part of the preceding algorithm that created the very input. This extends the tandem in Fig. 4 as depicted in Fig. 7 with the i-level cuts necessary for the next algorithm.

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

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

***i*-Level Cuts Within the Bottom-Up Reduce.** To compute the 1-level and 2-level cuts of the output during the Reduce algorithm, the algorithms in the proofs of Proposition 1 and 2 need to be incorporated. Since the Reduce algorithm works bottom-up, it cannot compute these cuts exactly: the bottom-up nature only allows information to flow from lower levels upwards while an exact result also requires information to be passed downwards. Specifically, Fig. 8 shows an unreduced BDD whose maximum 1 and 2-level cut is increased due to the reduction removing nodes above the cut. Both over-approximation algorithms below are tight since for the input in Fig. 8 they compute the exact result.

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

1. After having obtained all outgoing arcs for unreduced nodes for level k, the priority queue only contains outgoing arcs from a level $\ell < k$ to a level $\ell' > k$. All of these arcs (may) contribute to the cut.
2. After having obtained all outgoing arcs for level k, all yet unread arcs to terminals $b \in B$ are from some level $\ell < k$ and (may) contribute to the cut.
3. BDD nodes v removed by the first reduction rule in favor of its reduced child v' and $w_B(\_ \to v') = 1$ (may) contribute up to $|in(v')|$ arcs to the cut.
4. BDD nodes v' that are output on level k after merging duplicates (definitely) contribute with $w_B(v'.low) + w_B(v'.high)$ arcs to the cut.

1 and 2 can be obtained with some bookkeeping on the priority queue and the contents of the file containing arcs to terminals. 4 can be resolved when reduced nodes are pushed to the output. Yet, 3 cannot just use the immediate indegree of the removed node v since, as in Fig. 8, it may be part of a longer chain of redundant nodes. Here, the actual contribution to the cut at level $j = k$ is the indegree to the entire chain ending in v. Due to the single bottom-up sweep style of the Reduce algorithm, the best we can do is to assume the worst and always count reduced arcs $s' \to t'$ where a node v has been removed between s' and t' as part of the maximum cut.

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

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

This requires knowing $|in(v')|$. The Reduce algorithm in [38] reads from a file containing the parents of an unreduced node v, so information about the reduced result v' can be forwarded to its unreduced parents. Hence, one can accumulate the number of parents, $|in(v)|$. If $|in(v')|$ is not affected by the first reduction rule then this is an upper bound of $|in(v')|$. Otherwise, it still is sound in combination with the above over-counting to solve the 3rd type of contribution.

4 Experimental Evaluation

We have extended Adiar to incorporate the ideas presented in Sect. 3. Each algorithm has been extended to compute sound upper bounds for the next phase. Based on these, each algorithm chooses during initialisation between running with TPIE's internal or external memory data structures. This choice is encapsulated within C++ templates, which avoids introducing any costly indirection when using the auxiliary data structures.

Section 3.4 motivates the following three levels of granularity:

– **#nodes:** Theorem 1 is used based on knowing the number of internal nodes in the input and deriving the trivial worst-case size of the output.

- **1-level:** Extends #nodes with Theorem 2. The i-level cuts are given by computing the 1-level cut with the proof of Proposition 1 as described in Sect. 3.4 and then applying Lemma 1 to obtain a bound on the 2-level cut.
- **2-level:** Extends the 1-level variant by computing 2-level cuts directly with the algorithm based on the proof of Proposition 2 in Sect. 3.4.

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

4.1 Benchmarks

We have evaluated the quality of our modifications on the four benchmarks below that are publicly available at [32]. These were also used to measure the performance of Adiar 1.0 (BDDs) and 1.1 (ZDDs) in [34,38]. The first benchmark is a circuit verification problem and the others are combinatorial problems.

- ***EPFL* Combinational Benchmark Suite** [2]**.** The task is to check equivalence between an original hardware circuit (specification) and an optimised circuit (implementation). We construct BDDs for all output gates in both circuits, and check if they are equivalent. We focus on the 23 out of the 46 optimised circuits that Adiar could verify in [38]. Input gates are encoded as a single variable, x_i, with a maximum 2-level cut of size 2.
- **Knight's Tour.** On an $N_r \times N_c$ chessboard, the set of all paths of a Knight is created by intersecting the valid transitions for each of the $N_r N_c$ time steps. The cut of each such ZDD constraint is $\sim 8 N_r N_c$. Then, each Hamiltonian constraint with cut size 4 is imposed onto this set [34].
- N**-Queens.** On an $N \times N$ chessboard, the constraints on placing queens are combined per row, based on a base case for each cell. Each row constraint is finally accumulated into the complete solution [21]. For BDDs, each basic cell constraint has a cut size of $\sim 3N$, while for ZDDs it is only 3.
- **Tic-Tac-Toe.** Initially, a BDD or ZDD with cut size $\sim N$ is created to represent that N crosses have been set within a $4 \times 4 \times 4$ cube. Then for each of the 76 lines, a constraint is added to exclude any *non-draw* states [21]. Each such line constraint has a cut size of 4 with BDDs and 6 with ZDDs.

4.2 Tradeoff Between Precision and Running Time

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

All combinatorial benchmarks use a unary operation at the end to count the number of solutions. Table 1 shows the average ratio between the predicted and actual maximum size of this operation's priority queue. As instances grow larger, the quality of the #nodes heuristic deteriorates for BDDs. On the other

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

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

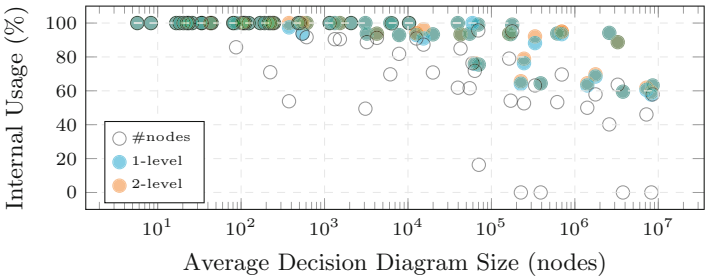


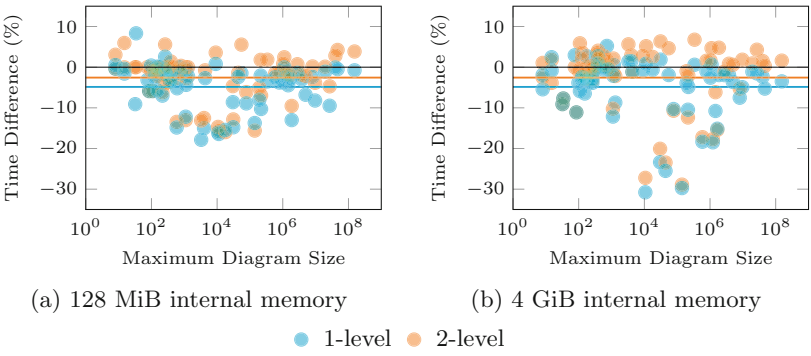
Fig. 9. *Internal* vs. *external* memory usage for product constructions (128 MiB).

hand, the 1 and 2-level cut heuristics are at most off by a factor of 2. Hence, since the priority queue's maximum size is some 2-level cut, the algorithms in Sect. 3.4 are only over-approximating the actual maximum 2-level cut by a factor of 2. The result of this is that i-level cuts can safely identify that a BDD with $5.2 \cdot 10^7$ nodes (1.1 GiB) can be processed purely within 128 MiB of internal memory available. The precision of i-level cuts are worse for ZDDs, but still allow processing a ZDD with $4.3 \cdot 10^7$ nodes (978 MiB) with 128 MiB of memory.

This difference in precision affects the product construction algorithms, e.g. the Apply operation. Figure 9 shows the amount of product constructions that each heuristic enables to run with internal memory data structures. Even when the average BDD was 10^7 nodes (229 MiB) or larger, with i-level cuts at least 59.5% of all algorithms were run purely in 128 MiB of memory, whereas with #nodes sometimes none of them were. Yet, while there is a major difference between #nodes and 1-level cuts, going further to 2-level cuts only has a minor effect.

How often internal memory could be used is also reflected in Adiar's performance. Figure 10 shows the difference in the running time between using i-level cuts and only using #nodes. All benchmarks runs were interleaved and repeated at least 8 times. The minimum measured running time is reported as it minimises any noise due to hardware and the operating system [12]. Since the #nodes version also includes the computation for the 1-level cuts but does not use them, any performance decrease in Fig. 10 for 1-level cuts is due to noise.

Using the geometric mean, 1-level cuts provide a 4.9% improvement over #nodes. Considering the 1.0% overhead for computing the 1-level cuts, this



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

Fig. 10. Adiar with i-level cuts compared to #nodes (lower is better). Horizontal lines show the average difference in performance.

is a net improvement of 3.9%. More importantly, in a considerable amount of benchmarks, using i-level cuts improves the performance by more than 10%, sometimes by 30%. These are the benchmark instances where only i-level cuts can guarantee that all auxiliary data structures can fit within internal memory, yet the instances are still so small that there is a major overhead in initialising TPIE's external memory data structures.

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

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

4.3 Impact of Introducing Cuts on Adiar's Running Time

In [34,38] we measured the performance of Adiar 1.0 and 1.1 against the conventional BDD packages CUDD 3.0 [39] and Sylvan 1.5 [16]. In those experiments [33,35], Sylvan was not using multi-threading and all experiments were run on machines with 384 GiB of RAM of which 300 GiB was given to the BDD package. To gauge the impact of using cuts, we now compare our previous measurements without cuts to new ones with cuts on the exact same hardware and settings. The results of our new measurements are available at [36].

With 300 GiB internal memory available, all three modified versions of Adiar essentially behave the same. Hence, in Fig. 1 (cf. Sect. 1) we show the best performance for all three versions on top of the data reported in [38]. Even on the largest benchmarks we see a performance increase by exploiting cuts. Most important is the increase in performance for the moderate-size instances where

the initialisation of TPIE's external memory data structures are costly, e.g. N-Queens with $N < 11$ and Tic-Tac-Toe with $N < 19$. Based on the data in [34,38] these instances of the combinatorial benchmarks are the ones where the largest constructed BDD or ZDD is smaller than $4.9 \cdot 10^6$ nodes (113 MiB).

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

Verifying the EPFL benchmarks involves constructing a few BDDs that are larger than the 113 MiB bound mentioned above, but most BDDs are much smaller. For the 15 EPFL circuits that only generate BDDs smaller than 113 MiB, using cuts decreases the computation time on average by 92% (with median 92%). While Adiar v1.0 still took 56.5 h to verify these 15 circuits, now with Adiar 1.2 it only takes 4.0 h to do the same. These 52.5 h are primarily saved within one of the 15 circuits. Specifically, using cuts has decreased the time to verify the `sin` circuit optimised for depth by 52.1 h. Here, the average BDD size is 2.9 KiB, the largest BDD constructed is 25.5 MiB in size, and up to 42,462 BDDs are in use concurrently.

Despite this massive performance improvement with Adiar 1.2 due to our new technique, there is still a significant gap of 3.7 h with CUDD and Sylvan on these 15 circuits. We attribute this to the fact that these benchmarks also include many computations on really tiny BDDs. Although we keep the auxiliary data structures in internal memory, the resulting BDDs are still stored on disk, even when they consist of only a few nodes.

5 Conclusion

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

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

5.1 Related and Future Work

Many approaches tried to achieve large-scale BDD manipulation with distributed memory algorithms, some based on breadth-first algorithms, e.g. [21,25,40,42]. Yet, none of these approaches obtained a satisfactory performance. The speedup obtained by a multicore implementation [16] relies on parallel depth-first algorithms using concurrent hash tables, which doesn't scale to external memory.

CAL [31] (based on a breadth-first approach [6,29]) is to the best of our knowledge the only other BDD package designed to process large BDDs on a single machine. CAL is I/O efficient, assuming that a single BDD level fits into main memory; the I/O efficiency of Adiar does not depend on this assumption. Similar to Adiar, CAL suffers from bad performance for small instances. To deal with this, CAL switches to the classical recursive depth-first algorithms when all the given input BDDs contain fewer than 2^{19} nodes (15 MiB). As far as we can tell, CAL's threshold is purely based on experimental results of performance and without any guarantees of soundness. That is, the output may potentially exceed main memory despite all inputs being smaller than 2^{19} nodes, which would slow it down significantly due to random-access. For BDDs smaller than CAL's threshold of 2^{19} nodes, Adiar 1.2 with i-level cuts could run almost all of our experiments with auxiliary data structures purely in internal memory.

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

5.2 Applicability Beyond Decision Diagrams

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

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

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

References

1. 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. Amarú, L., Gaillardon, P.E., De Micheli, G.: The EPFL combinational benchmark suite. In: 24th International Workshop on Logic & Synthesis (2015)
3. Amparore, E.G., Donatelli, S., Gallà, F.: starMC: an automata based CTL* model checker. PeerJ Comput. Sci. **8**, e823 (2022)
4. Arge, L.: The I/O-complexity of ordered binary-decision diagram manipulation. In: Staples, J., Eades, P., Katoh, N., Moffat, A. (eds.) ISAAC 1995. LNCS, vol. 1004, pp. 82–91. Springer, Heidelberg (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. 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
8. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
9. Bryant, R.E., Biere, A., Heule, M.J.H.: Clausal proofs for pseudo-Boolean reasoning. In: TACAS 2022. LNCS, vol. 13243, pp. 443–461. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_25
10. Bryant, R.E., Heule, M.J.H.: Dual proof generation for quantified Boolean formulas with a BDD-based solver. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 433–449. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_25
11. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: TACAS 2021. LNCS, vol. 12651, pp. 76–93. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_5
12. Chen, J., Revels, J.: Robust benchmarking in noisy environments. arXiv (2016). https://arxiv.org/abs/1608.04295
13. 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 1995, Society for Industrial and Applied Mathematics (1995)
14. Ciardo, G., Miner, A.S., Wan, M.: Advanced features in SMART: the stochastic model checking analyzer for reliability and timing. SIGMETRICS Perform. Evaluation Rev. **36**(4), 58–63 (2009)
15. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. Int. J. Softw. Tools Technol. 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. Int. J. Softw. Tools Technol. 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: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 479–483. Springer, 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. Hellings, J., Fletcher, G.H., Haverkort, H.: Efficient external-memory bisimulation on DAGs. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 553–564. SIGMOD 2012, Association for Computing Machinery (2012). https://doi.org/10.1145/2213836.2213899, https://doi.org/10.1145/2213836.2213899
20. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
21. 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
22. Lampis, M., Kaouri, G., Mitsou, V.: On the algorithmic effectiveness of digraph decompositions and complexity measures. Discrete Optim. **8**(1), 129–138 (2011). https://doi.org/10.1016/j.disopt.2010.03.010. parameterized Complexity of Discrete Optimization
23. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. Int. J. Softw. Tools Technol. Transfer **19**(1), 9–30 (2015). https://doi.org/10.1007/s10009-015-0378-x
24. Meyer, U., Sanders, P., Sibeyn, J.: Algorithms for Memory Hierarchies: Advanced Lectures. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36574-5
25. Milvang-Jensen, K., Hu, A.J.: BDDNOW: a parallel BDD package. In: Gopalakrishnan, G., Windley, P. (eds.) FMCAD 1998. LNCS, vol. 1522, pp. 501–507. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49519-3_32
26. Minato, S.I.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: 30th Design Automation Conference (DAC), pp. 272–277. Association for Computing Machinery (1993). https://doi.org/10.1145/157485.164890
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. Mølhave, T.: Using TPIE for processing massive data sets in C++. Duke University, Durham, NC, Technical report (2012)
29. 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
30. Papadimitriou, C.H., Yannakakis, M.: Optimization, approximation, and complexity classes. J. Comput. Syst. Sci. **43**(3), 425–440 (1991). https://doi.org/10.1016/0022-0000(91)90023-X
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.: BDD Benchmark. Zenodo (2023). https://doi.org/10.5281/zenodo.7040263
33. Sølvsten, S.C., van de Pol, J.: Adiar 1.0.1 : Experiment data (11 2021). https://doi.org/10.5281/zenodo.5638551
34. Sølvsten, S.C., van de Pol, J.: Adiar 1.1: zero-suppressed decision diagrams in external memory. In: Rozier, K.Y., Chaudhuri, S. (eds.) NFM 2023. LNCS, vol. 13903, pp. 464–471. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-33170-1_28
35. Sølvsten, S.C., van de Pol, J.: Adiar 1.1.0 : experiment data (2023). https://doi.org/10.5281/zenodo.7709134
36. Sølvsten, S.C., van de Pol, J.: Adiar 1.2.0 : experiment data (2023). https://doi.org/10.5281/zenodo.8124120
37. Sølvsten, S.C., van de Pol, J.: Predicting memory demands of BDD operations using maximum graph cuts (extended paper) (2023)
38. 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
39. Somenzi, F.: CUDD: CU decision diagram package, 3.0. Technical report, University of Colorado at Boulder (2015)
40. Stornetta, T., Brewer, F.: Implementation of an efficient parallel BDD package. In: Design Automation Conference Proceedings, vol. 33, pp. 641–644 (1996). https://doi.org/10.1109/DAC.1996.545653
41. Vengroff, D.E.: A transparent parallel I/O environment. In: In Proceedings of 1994 DAGS Symposium on Parallel Computation, pp. 117–134 (1994)
42. Yang, B., O'Hallaron, D.R.: Parallel breadth-first BDD construction. SIGPLAN Not. **32**(7), 145–156 (1997). https://doi.org/10.1145/263767.263784