Formal Methods in Computer-Aided Design 2023

Binary Decision Diagrams on Modern Hardware

Samuel Pastva and Thomas Henzinger
Institute of Science and Technology Austria
Klosterneuburg, 3400 Austria
Email: samuel.pastva@ist.ac.at,tah@ist.ac.at

***Abstract*—Binary decision diagrams (BDDs) are one of the fundamental data structures in formal methods and computer science in general. However, the performance of BDD-based algorithms greatly depends on memory latency due to the reliance on large hash tables and thus, by extension, on the speed of random memory access. This hinders the full utilisation of resources available on modern CPUs, since the absolute memory latency has not improved significantly for at least a decade.**

In this paper, we explore several implementation techniques that improve the performance of BDD manipulation either through enhanced memory locality or by partially eliminating random memory access. On a benchmark suite of 600+ BDDs derived from real-world applications, we demonstrate runtime that is comparable or better than parallelising the same operations on eight CPU cores.

***Index Terms*—binary decision diagram, symbolic algorithm, hash table, cache.**

I. Introduction

Binary decision diagrams (BDDs) [9] (or more specifically, reduced ordered binary decision diagrams (ROBDDs)) are one of the fundamental data structures in computer science. They are directed acyclic graphs representing Boolean functions, often exponentially more succinct compared to Boolean expressions or function tables [34].

They have a wide range of applications in formal verification [1], [10], [12], [14], [20], [43], satisfiability checking [22], [24], hardware design [26], [27], [44], test design [37], dialectical frameworks [16], and optimisation [7]. They are the building blocks for the so-called *symbolic algorithms* that are, among other applications, used for exploration of large graphs suffering from exponential state-space blow-up [3], [5], [6], [28], [45]. Many extensions of BDDs exist that attempt to improve their succinctness, typically at the cost of more complex manipulation algorithms. One example are zero-suppressed decision diagrams [32], but a more exhaustive summary of known BDD variants can be found in [2].

There are implementations of BDDs that rely on shared-memory [41] and distributed-memory [35] parallelism, external memory [38] and even GPUs [42]. Furthermore, variable ordering within the BDD has a strong impact on its succinctness and has been an intense subject of optimisation [18], [21].

In this paper, we tackle another important aspect of BDD implementation. In general, it is known that operations on BDDs are bottlenecked by memory latency due to their extensive use of large hash tables [8].

This is an unfortunate bottleneck on modern hardware, since the absolute memory latency has not improved for at least the last 15 years [13]. Memory *capacity*, memory *bandwidth*, the number of CPU cores, as well as their *width* and *frequency* has grown significantly. However, the memory *latency* on a CPU bought today (i.e. 2023) is essentially the same as on the one bought in 2006 [13].

In this paper, we demonstrate that as a result, a *modern* CPU (2020) is in fact *worse* at BDD manipulation compared to its *legacy* (2014) counterpart once the problem size grows beyond the last-level cache (typically L3 cache). To address this problem, we propose an alternative data structure that replaces one of the underlying hash tables (node uniqueness table). We also devise additional criteria to reduce the amount of memory accesses performed during BDD manipulation. In the end, we observe that our approach to BDD manipulation indeed improves the performance on a modern CPU significantly, to an extent comparable with parallelisation on 8 CPU cores.

Finally, note that this is not the first attempt to design a more cache friendly BDD implementation. In [15], the authors propose to use a more cache-friendly hashing scheme called Hopscotch hashing. However, the paper also proposes fundamentally different BDD manipulation algorithms (based on BFS, not DFS), incorporates parallelism, a novel GC algorithm, and a number of other experimental optimizations. The work demonstrates an improvement over existing BDD packages, but does not show whether the improvement is due to improved cache friendliness or due to the fundamentally different algorithm. Meanwhile, [29] proposes to order the BDD nodes chronologically: i.e. the BDD node must appear in memory *after* both of its child nodes (this is trivially satisfied by the DFS post-order in which nodes are typically generated). This assumption then partially eliminates memory lookups that would be necessary with arbitrary node order. An improvement in runtime is demonstrated, but we are not aware of any modern work that uses this technique (the original paper is now 25 years old). Furthermore, we are not aware of any work that would directly *measure* the extent to which BDD operations are bottlenecked by memory or attempt to control for this aspect in the measurements.

A. Paper structure

First, Section II recalls the definition of ROBDDs and of the APPLY algorithm which ROBDDs use to perform logical transformations. Section III then describes our benchmark scenario involving a *modern* and a *legacy* CPU, together with the set of tested BDD operations and packages.

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_20

This article is licensed under a Creative Commons Attribution 4.0 International License

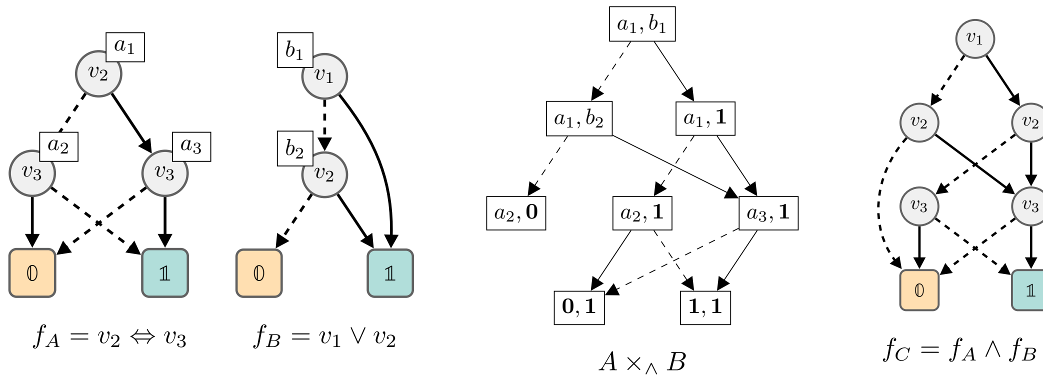

Fig. 1. Illustration of ROBDDs over the set $\mathcal{V} = \{v_1, v_2, v_3\}$. Solid edges represent the *high* successor while dashed edges represent the *low* successor. Left to right: ROBDDs of two simple functions $f_A = v_2 \Leftrightarrow v_3$ and $f_B = v_1 \vee v_2$. The product graph for the operation $\text{APPLY}_\wedge(a_1, b_1)$. The resulting ROBDD for the function $f_C = f_A \wedge f_B$.

Subsequently, Section IV-A demonstrates that for sufficiently large problem instances, the performance of a typical BDD package is in fact worse on the modern CPU. In Section IV-B, we then observe that BDD nodes with in-degree one are significantly over-represented in a typical BDD.

Using this observation, we define a new node table with improved memory locality (Section IV-C) and formulate a simple rule that eliminates a portion of redundant accesses to the operations cache (Section IV-D). Finally, we test the performance of this new approach in Section IV-E.

Due to the nature of this work, we also provide a reproducibility artefact[1] which contains all benchmark data and code, as well as the raw results for each experiment. To keep the main paper concise, some of the more low-level parts of the methodology and results are only available within the artefact.

II. Preliminaries

Since there are already many excellent texts describing BDDs in detail, we only introduce the terminology and notation relevant for this paper. An interested reader is further referred for example to Chapter 7 of [14].

Notation: We assume $\mathcal{V}$ is a finite set of Boolean variable symbols. We also use 1 and 0 interchangeably with *true* and *false* when appropriate.

A. Binary decision diagrams

A *binary decision diagram* (BDD) B is a directed acyclic graph with a single *root* node (denoted $root(B)$) and two *terminal* nodes $\mathbf{0}$ and $\mathbf{1}$. We write B to mean either the BDD itself or the set of its nodes when the distinction is clear from context. Each non-terminal node $x \in B$ is labelled with a variable $var(x) \in \mathcal{V}$. Furthermore, each such $x \in B$ has exactly two successor nodes denoted $low(x)$ and $high(x)$ (corresponding to the choice of $var(x) = 0$ and $var(x) = 1$ respectively). A simple example is given in Fig. 1, left.

We assign a Boolean function $f_x : \{0,1\}^{\mathcal{V}} \to \{0,1\}$ to each $x \in B$ s.t. $f_x = (var(x) \wedge f_{high(x)}) \vee (\neg var(x) \wedge f_{low(x)})$, with $f_{\mathbf{0}} = \mathit{false}$ and $f_{\mathbf{1}} = \mathit{true}$. In other words, every valuation of variables from $\mathcal{V}$ determines a path from x to either $\mathbf{0}$ or $\mathbf{1}$, corresponding to the output of the function f_x.

Now, let us assume there is *some* total ordering on the variables $\mathcal{V}$. We say that a BDD B is *ordered* (OBDD) when $var(low(x)) > var(x)$ and $var(high(x)) > var(x)$ for every non-terminal $x \in B$ (we also assume $var(\mathbf{0})$ and $var(\mathbf{1})$ are values greater than any $v \in \mathcal{V}$). For example, in Fig. 1, the variable ordering is $v_1 < v_2 < v_3$.

Finally, we say that B is *reduced* (ROBDD) when: (a) there is no vertex x such that $low(x) = high(x)$, and (b) there are no two vertices x and y such that $var(x) = var(y)$, $low(x) = low(y)$, and $high(x) = high(y)$. These two requirements can be also interpreted as "reduction rules" that describe how to transform an OBDD into an ROBDD. In the following, we assume all BDDs are ordered and reduced, we thus use the terms BDD and ROBDD interchangeably.

B. The APPLY algorithm

Given a fixed ordering of variables $\mathcal{V}$, each Boolean function $f : \{0,1\}^{\mathcal{V}} \to \{0,1\}$ has a unique corresponding ROBDD [9]. We also have an algorithm that, given two ROBDDs A and B, computes ROBDD C of the function $f_C = f_A \star f_B$ where $\star$ is some binary Boolean operator. In the worst case, this APPLY algorithm operates in $\mathcal{O}(|A| \cdot |B|)$ time. However, the complexity for practical BDDs is typically much smaller than this upper bound.

In Algorithm 1, we give a recursive formulation of this APPLY procedure. In practice, one often replaces the recursion with a loop and an explicit stack to avoid overflow and to eliminate function call overhead. The APPLY algorithm also relies on two core data structures which are typically implemented using hash tables.

First is the *node table* (also called *unique table*) accessed using the $\text{ENSURE_NODE}(v, l, h)$ procedure. This function searches the node table for a node x with $var(x) = v$, $low(x) = l$, and $high(x) = h$. If such node is found, its identifier x is returned. When no such node exists, a new node is created and its identifier is returned.

Second is the *cache table* responsible for memorisation of already computed results. This table is often implemented as a

[1] https://doi.org/10.5281/zenodo.7958052

```
1  Function APPLY_⋆(x_A ∈ A, x_B ∈ B)
2  |  if x_A ⋆ x_B ∈ {0, 1} then return x_A ⋆ x_B;
3  |  if let x_C ← CACHE(x_A, x_B) then return x_C;
4  |  v ← min(var(x_A), var(x_B));
5  |  (l_A, h_A) ← (x_A, x_A);
6  |  if v = var(x_A) then
7  |  |  (l_A, h_A) ← (low(x_A), high(x_A));
8  |  (l_B, h_B) ← (x_B, x_B);
9  |  if v = var(x_B) then
10 |  |  (l_B, h_B) ← (low(x_B), high(x_B));
11 |  l ← APPLY_⋆(l_A, l_B);
12 |  h ← APPLY_⋆(h_A, h_B);
13 |  x_C ← ENSURE_NODE(v, l, h) if l ≠ h else l;
14 |  CACHE(x_A, x_B) ← x_C;
15 |  return x_C;
```

Algorithm 1: BDD APPLY algorithm parametrised by a binary Boolean operator $\star$.

leaky hash table which overwrites values when hash collision occurs. Such implementation is correct (if a value is missing, it is simply recomputed), but depending on the number of collisions, it may exceed the $\mathcal{O}(|A| \cdot |B|)$ time complexity. This introduces a possible trade-off between running time and memory consumption.

Finally, let us observe that for every operation $\text{APPLY}_\star(root(A), root(B))$, there is a tighter complexity metric given by the number of unique (x_A, x_B) pairs reachable from $(root(A), root(B))$ by $\text{APPLY}_\star$. We call this set of tuples the *product graph* of $\text{APPLY}_\star(root(A), root(B))$ and denote it $A \times_\star B$. Note that the size of this product graph depends on $\star$, because some operations can short-circuit the condition on Line 2 even when one of the arguments is not a terminal node (e.g. $x_A \wedge \mathbf{0} = \mathbf{0}$). We then observe that the complexity of the APPLY algorithm is $c \cdot |A \times_\star B|$ for some constant c, assuming the calls to CACHE and ENSURE_NODE are $\mathcal{O}(1)$ and that CACHE is not leaky. Observe that for the example in Fig. 1, we have $|A| \cdot |B| = 20$, but $|A \times_\wedge B| = 8$.

III. Benchmark Methodology and Hardware

Due to the practical nature of this paper, we must thoroughly disclose what benchmarks are performed and how we measure the performance of BDD packages on our hardware.

A. Benchmark BDDs

Many authors test the performance of BDDs on pathological worst case scenarios like the multiplier circuit or the n-queens problem [11], [33]. While this certainly reveals some performance characteristics of the implementation, it is susceptible to over-fitting of a particular pattern of BDD operations. To mitigate this issue, we derive a large benchmark dataset based on real-world problems from model verification in systems biology, scaling from simple BDDs to millions of nodes.

Specifically, we use the tool AEON [4] which performs exhaustive formal analysis of Boolean networks, simple logical models of asynchronous biological processes. We then take the 20 largest models from the Biodivine Boolean Models (BBM) dataset [36], ranging from 100 to 300 variables (and consequently, $2^{100-300}$ states).

We use AEON to compute the BDD representations of a set of network fixed-points and a set of reachable states based on a predefined initial state for each model. These are tasks that are also commonly performed by formal methods tools in computer science and are not specific to Boolean networks or systems biology. For each computation, we save every intermediate BDD smaller than ten million nodes into a separate file. If two different BDDs of equivalent size are encountered, we only retain the latest BDD. This generates a large dataset of realistic BDDs of increasing size.

Now, our goal is to define a set of benchmarks which cover the space of admissible BDD operations over these real-world BDDs as uniformly as possible. Each BDD B can be assigned a *bucket* $b(B) = log_{10}(|B|)$ (the "order of magnitude" of the size of B). We then sample pairs of BDDs A, B (w.l.o.g. we assume $|A| \geq |B|$) and compute the BDD $C = A \wedge B$. Such *benchmark triple* (A, B, C) is then assigned into a bucket triple $(b(A), b(B), b(C))$. For each bucket triple, we save the first five unique benchmarks. The sampling stops once no new viable benchmark is found in the last 100 samples.

In our case, this process yields 629 benchmark instances using 963 unique BDDs. The final number of triples for each combination of buckets is summarized in Fig. 2. While the result does not cover every theoretically admissible combination of BDD sizes, it still covers a wide range of possible BDD operations. When presenting results for individual benchmarks, these are typically sorted by the size of the product graph $A \times_\wedge B$, as this gives a good approximation of the expected complexity of the BDD operation.

Finally, note that for the sake of simplicity, our tests only cover the conjunction ($\wedge$) operator. However, we have no reason to believe that there are significant differences in performance compared to other Boolean operators once the size of the product graph is taken into account. As such, we prioritize a wider coverage of different BDD sizes to testing more Boolean operators.

B. Hardware configuration

To compare "modern" and "legacy" CPUs, we consider the following two platforms:

- 4-core Intel i7-4790 (released in 2014) with 32GB of DDR3-1600 memory at CAS latency of 9 cycles.
- 8-core AMD Ryzen 5800X (year 2020) with 128GB of DDR4-3200 memory at CAS latency of 18 cycles.

These are both very common CPUs from their respective generations. They are paired with the maximum amount of memory available on that platform at the top speed officially supported by the manufacturer[2]. Furthermore, notice that the

[2]The data rate is maximal supported on both CPUs. However, in terms of latency, the DDR3 configuration is slightly worse than the best official JEDEC configuration (9 cycles instead of 8 for DDR3-1600), while the DDR4 configuration is slightly better than the best official JEDEC configuration (18 cycles instead of 20 for DDR4-3200). Hence the modern system even has a *small* advantage compared to the officially claimed "best" configurations.

$\lvert A\rvert$	$\lvert B\rvert$	$\lvert A \wedge B\rvert$								
		10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
10^1	10^1	5	5	–	–	–	–	–	–	–
10^2	10^1	1	5	5	–	–	–	–	–	–
10^2	10^2	0	5	5	5	–	–	–	–	–
10^3	10^1	0	0	5	5	–	–	–	–	–
10^3	10^2	5	0	5	5	1	–	–	–	–
10^3	10^3	5	0	5	5	0	0	–	–	–
10^4	10^1	0	0	1	5	5	–	–	–	–
10^4	10^2	4	0	5	5	5	1	–	–	–
10^4	10^3	5	0	5	5	4	0	0	–	–
10^4	10^4	5	0	5	5	5	3	0	0	–
10^5	10^1	0	0	0	0	5	4	–	–	–
10^5	10^2	0	0	5	5	5	5	1	–	–
10^5	10^3	5	0	5	5	5	5	0	0	–
10^5	10^4	5	0	5	5	5	5	4	0	0
10^5	10^5	5	0	2	5	5	5	5	1	0
10^6	10^1	0	0	0	0	5	5	5	–	–
10^6	10^2	1	0	5	5	5	5	5	0	–
10^6	10^3	5	0	5	5	5	5	4	1	0
10^6	10^4	5	0	5	5	5	5	5	0	0
10^6	10^5	5	0	1	5	5	5	5	1	1
10^6	10^6	5	0	4	4	5	5	5	5	0
10^7	10^1	0	0	0	0	0	1	5	2	–
10^7	10^2	5	0	5	5	5	5	5	0	0
10^7	10^3	5	0	5	5	5	5	5	1	1
10^7	10^4	5	0	5	5	5	5	5	1	1
10^7	10^5	5	0	0	5	5	5	5	5	0
10^7	10^6	5	0	0	0	5	5	5	5	5
10^7	10^7	5	0	0	1	3	5	5	5	5

Fig. 2. The distribution of the 629 benchmarks within buckets of exponentially increasing size. Dashes indicate combinations that are provably impossible.

effective latency of both memory configurations is the same: the DDR4 configuration has twice the CAS latency, but also twice the data rate of the DDR3 configuration[3].

All automated overclocking features were disabled on both CPUs to improve consistency between runs and we did not observe any thermal throttling. Furthermore, we assume that no other programs were using a significant amount of resources during measurements. This is critical due to the fact that multiple CPU cores compete for the shared L3 cache.

Finally, as a sanity check, some tests were also repeated on a similar server hardware (Intel Xeon E7-8860; released in 2011, and AMD EPYC 7713; released in 2021) yielding comparable results. However, since we did not have exclusive access to these machines and thus could not prevent measurement noise caused by sharing resources with other software, we focus on the numbers obtained for the "desktop" platforms.

C. BDD packages and the benchmark harness

Our implementation is built using the Rust programming language. In several instances, we use *unsafe* operations in Rust to remove unnecessary array bounds check in the core algorithm. Aside from these instances, the memory safety of the implementation has been validated by the Rust compiler. Testing was performed using Debian 12 with `gcc 12.2.0` and `rustc 1.71.0`. Each measurement was performed over at least three runs. When we observed standard deviation higher than 5% of the average, we repeated the experiment up to ten times to improve reliability. However, this was only rarely necessary. Due to this low run-to-run variance, we only report the average runtime for each experiment.

Aside from our purpose-built implementation, we consider the following three BDD packages:

- `cudd 3.0` [39] as one of the best known BDD packages. While originally designed over 25 years ago, CUDD is still one of the most widely used BDD packages.
- `sylvan` [41] is one of the first BDD packages to demonstrate practical multi-core scalability. When presenting results, we suffix its name with the number of employed cores (e.g. `sylvan-4`).
- `lib-bdd` of the tool AEON [4] is an example of a "naive" BDD implementation: it uses hash tables provided by Rust's standard library and generally does not implement any advanced features like variable reordering.

To compare performance, we focus on individual BDD operations, i.e. on the individual runs of the APPLY algorithm with the conjunction ($\wedge$) operator. For each package we prepared a test harness where the runtime of the operation is isolated from other overhead such as package initialization and loading of test BDDs into memory. Where applicable, we disable garbage collection or dynamic variable reordering, as it is not relevant for our testing.

Furthermore, the performance of BDD packages often strongly depends on the initial size of the node and cache table [40]. While the packages can grow these data structures dynamically, the combination of the problem size and the growth rate can influence runtime significantly [40].

To reduce the impact of this variable on the final runtime, we allow each package to pre-allocate as much memory as possible during initialization (not counted towards the total runtime)[4]. While this can impact performance on smaller BDDs, it seems necessary to allow each package to reach its full potential on larger benchmarks and is outright mandatory in some instances.[5] Furthermore, this models the situation where our single benchmarked BDD operation is part of a larger symbolic computation which amortizes the allocation of the necessary data structures across many operations.

Finally, when reporting the *average* value for additive metrics (like runtime), this stands for the standard arithmetic mean. Meanwhile, for multiplicative metrics (like speed-up), this corresponds to the geometric mean which is more appropriate in such instances. We also use internal CPU performance counters to measure executed instructions per clock and L3

[3]Currently, the fastest officially supported memory configuration in consumer CPUs is roughly DDR5-5600 CL32 (faster configurations do exist but require CPU overclocking and are typically not feasible with high amounts of memory). Unfortunately, we did not have access to such configuration. However, its effective latency is again very similar to our tested scenarios.

[4]Package `lib-bdd` cannot perform any pre-allocation and thus its memory allocation counts towards the total runtime. Also note that while each package is allowed to *reserve and initialize* as much memory as needed, it can still choose to initially use a smaller portion and grow the hash tables gradually. In particular, `sylvan` grows the hash table utilization gradually, while `cudd` always uses the whole table from the start.

[5]In `cudd`, the growth is extremely slow beyond the first few gigabytes of memory, making it practically unusable unless we explicitly override the table sizes to reserve as much memory as necessary beforehand.

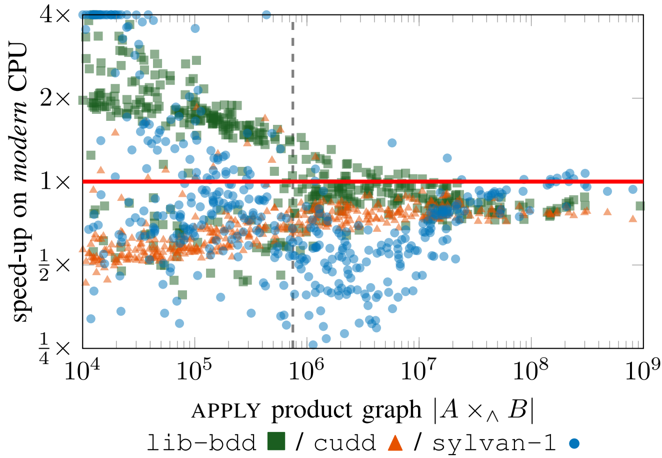

Fig. 3. The speed-up in runtime of the *modern* vs. the *legacy* system. Speed-up greater than 4× is truncated to 4×. Points below the red line represent slow-down instead of speed-up. Points to the right of the vertical line represent 200 largest benchmark instances. Note that both axis are logarithmic.

cache miss ratio. However, their usage has negligible impact on runtime.[6]

IV. Algorithms and Results

We structure this section as follows: First, we present several empirical findings about BDDs and the current APPLY implementations. Based on these findings, we then propose a variant of the node and cache table which should help improve performance on modern CPUs. Finally, we evaluate these claims empirically on our benchmark dataset.

A. Comparing modern and legacy systems

While it is generally *known* that the performance of BDD operations is bottlenecked by the memory latency, it is useful to demonstrate the extent of this problem. To do so, we benchmarked each BDD package on both the *legacy* and the *modern* system, calculating the relative *speed-up* achieved by the modern system. Note that for this test, we used the same pre-allocation settings on both systems and excluded six benchmarks that we could not reliably complete on the legacy system due to insufficient memory. The results of this analysis are presented in Fig. 3.

If we consider the full benchmark suite, the results appear to be largely positive: For `lib-bdd` and `sylvan-1`, two packages that gradually increase their table sizes with benchmark size, we see 1.52× and 1.78× average improvement, respectively. Only for `cudd`, which was instructed to use the maximal table size for each benchmark due to issues with table growth, we actually observe a slow-down of 0.69×. However, once we focus on the 200 largest benchmarks, every package is actually slower compared to the legacy system: a `lib-bdd` operation is 0.88× slower on a modern system, `cudd` operation is 0.77× slower, and a `sylvan-1` operation is even 0.61× slower.

From Fig. 3, it is clear that small benchmarks benefit the most from the modern CPU (except for `cudd`), which aligns with our assumption that additional L3 cache on the modern CPU improves BDD performance. In case of `cudd`, the reason for the poor performance on the smaller problem instances is the table growth setting: When the hash tables grow gradually, they maintain high density and thus utilise the available cache lines well. However, when the table is set to its maximal size from the start, it is very sparse for small problems. As such, each cache line will typically store only one element, making it much less effective. Intuitively, for this `cudd` configuration, every benchmark behaves like a "large" benchmark and it further amplifies the difference in practical latency of the modern and legacy system.

Also note that the relatively good results of `lib-bdd` compared to the other packages can be primarily attributed to its naive architecture: Each BDD operation in `lib-bdd` performs *more* instructions to accomplish the same task compared to the other packages (evidence for this is given in the subsequent text), which leaves the CPU more headroom for optimization and reordering while waiting for memory. In other words, `lib-bdd` achieves the best speed-up on large problems because it is the slowest, most inefficient package with the most space to improve.

To support these claims experimentally, we use CPU performance counters to measure the number of CPU cycles, executed instructions, L3 cache references and L3 cache misses for each benchmark. Interestingly, both `cudd` and `lib-bdd` seem to perform approx. 22 L3 cache references for each node of the APPLY product graph on average. This number is also independent on the size of the product graph. Similarly, the absolute number of executed instructions per product graph node is also largely constant, but here `cudd` is consistently almost 2× better than `lib-bdd`. Both implementations thus perform as expected: their complexity in terms of instructions is in fact $c \cdot |A \times_\wedge B|$, with $c_{\texttt{cudd}} < c_{\texttt{lib-bdd}}$.

However, for the 200 largest problem instances, the average IPC (instructions per cycle) of both implementations is very low: 0.35 for `cudd` and 0.85 for `lib-bdd`[7]. This is because both implementations miss almost 40% of the L3 cache requests while an L3 cache reference occurs on average every 14 (`cudd`) and 32 (`lib-bdd`) instructions.

In theory, the modern CPU can achieve up to 4 IPC, and we actually observed 3.49 IPC on some small problem instances. At the same time, the worst IPC observed on the large benchmarks was just 0.18. Intuitively, this means that for the larger problem instances, the CPU is utilizing *less than 10%* of its available compute resources. Furthermore, the results indicate that any speed-up between the modern and the legacy system occurs for problem instances that fit into (or are close to) the available L3 cache.

[6]Performance counters are thread-local and thus could not be reliably used for the multithreaded BDD package `sylvan`, even with one worker.

[7]Note that higher IPC generally does not guarantee higher performance. While `lib-bdd` achieves higher IPC, it also performs more instructions and is overall 0.84× slower than `cudd` in these tests.

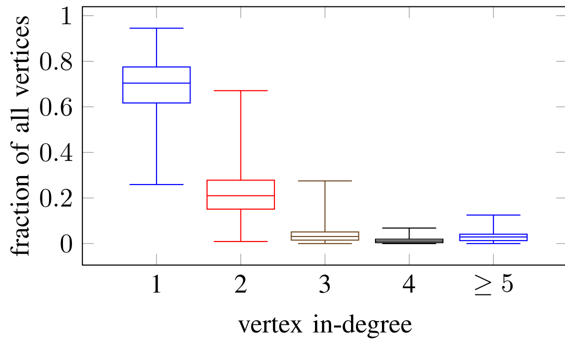

Fig. 4. The in-degree distribution within the BDDs of our benchmark dataset larger than 1000 nodes. Each box plot summarises the proportion of vertices with the corresponding in-degree across all available benchmark BDDs. Last box plot covers all in-degrees greater than or equal to five.

B. In-degree distribution in BDDs

Next we state a very simple but powerful observation: Due to each non-terminal node having exactly two outgoing edges, a BDD B has $2\cdot(|B|-2)$ edges. Furthermore, each node aside from root has at least one *incoming* edge. As such, for every node with in-degree $k > 2$, we expect to have $k-2$ nodes with only one incoming edge. This leads us to an intuitive hypothesis that single-parent nodes are greatly overrepresented among BDD nodes.

We can easily verify that this property holds in our benchmark dataset. In Fig. 4, we show the portion of nodes for individual in-degrees, cutting off at ≥ 5 incoming edges. As expected, our observations strongly resemble an exponential distribution: if we draw a random BDD node, there is roughly a 70% chance the node has a single parent, 20% chance it has two parents, 3% chance of it having three parents, and so on.

C. Parent-local node table design

We can now use the previous claim to design a new node table which is simpler and more cache friendly than current implementations. For simplicity, we assume that the node table stores entries in a continuous array, with new entries simply added to the end of the node list. This mirrors the assumption form [29], but as we later show, the node ordering is not a strict requirement for our approach. To resolve an ENSURE_NODE query, each node maintains a pointer structure akin to a *prefix trie* consisting of a subset of its parent nodes.

We know that due to the in-degree distribution, the vast majority of such tries will only store a handful of elements. Furthermore, due to the recursive nature of APPLY, whenever ENSURE_NODE(v,l,h) is executed, it is likely that the entries for nodes l and h are still in the CPU cache.

The pseudocode for this approach is given in Algorithm 2. Here, CREATE allocates a new table slot for the given node and returns the slot identifier. MSB stands for *most significant bit*, and ROTATE_ONE is a simple single-bit left rotation. Expression $node(t)$ is a shorthand for $(var(t), low(t), high(t))$.

Finally, for each node x, we introduce three additional identifiers which are all initialised to nil and stored together with the node data in the node table: $parent(x)$, $next_0(x)$, and $next_1(x)$. Here, $parent(x)$ is the reference to the root of the prefix trie. This trie then references all nodes where x is the maximal child (see Lines 2-4). Subsequently, $next_0(x)$ and $next_1(x)$ reference the "successor" nodes within the trie. Which successor is taken depends on the MSB prefix of the node hash (see Lines 11-12).

```
1  Function ENSURE_NODE(v, l, h)
2    (c_max, c_min) ← (max(l, h), min(l, h));
3    hash ← HASH(v, c_min);
4    t ← parent(c_max);
5    if t is nil then
6      x ← CREATE(v, l, h);
7      parent(c_max) ← x;
8      return x;
9    loop
10     if (v, l, h) = node(t) then return t;
11     msb ← MSB(hash) ;            // msb ∈ {0,1}
12     hash ← ROTATE_ONE(hash);
13     if next_msb(t) is nil then
14       x ← CREATE(v, l, h);
15       next_msb(t) ← x;
16       return x;
17     else
18       t ← next_msb(t);
```

Algorithm 2: The ENSURE_NODE procedure that resolves node duplicates through a parent-local trie.

We'd like to highlight several properties of Algorithm 2:

- We choose c_{max} to store the node because it avoids terminal nodes, which typically have very high in-degrees. However, other suitable conditions could be considered, as long as they only depend on the values (v,l,h).
- We do not include c_{max} in the node hash. Furthermore, the hash need not be truncated to a specific node table length, which further simplifies the hash function[8].
- Consequently, our HASH function is a simple multiplicative hash $(v \textbf{ xor } c_{min})\cdot p$ where p is a large prime number and the multiplication is natively truncated to 64 bits.
- If the table grows such that it needs to be completely reallocated, the existing nodes can be simply copied: there is no need to relocate nodes or recompute hashes.

To test the viability of this approach in practice, we prepare a simple benchmark that recreates all of our test BDDs one-by-one: first in a standard hash map with quadratic probing and a fast industry-standard hash function, and then in our parent-local node table. The nodes are created in DFS post-order, i.e. in the same order as within the APPLY algorithm. Averaging all BDDs, the parent-local table is 2.38× faster. However, on BDDs with at least one million nodes, the parent-

[8]Note that internally, a $x \textbf{ mod } y$ operation (except for $x \textbf{ mod } 2^k$) translates to division, which is still a relatively costly operation even on modern CPUs. A single division instruction may require as many CPU cycles as the whole ENSURE_NODE method if the required data is present in cache.

local approach is even $4.08\times$ faster than the standard hash table, with an L3 cache miss rate of just 18% vs. 27% for the standard hash table. Keep in mind that these results do not account for the rest of the APPLY algorithm. We will revisit this aspect in further experiments.

D. *Excluding single-parent tasks from* CACHE

Unfortunately, we cannot apply the same principle to the CACHE table, because its entries (i.e. the nodes of the product graph $A \times_\star B$) are queried in DFS pre-order, unlike the output BDD nodes which are queried in DFS post-order. This means that when we first query a particular entry (x_A, x_B), we do not have any information about its child nodes yet.

However, the observation about the number of single parent nodes nevertheless applies to the product graph as well. That is, for the majority of (x_A, x_B) pairs explored by the APPLY algorithm, there is a single parent (y_A, y_B) through which (x_A, x_B) is discovered.

Proposition 1: Let (x_A, x_B) be a node of the product graph $A \times_\star B$ such that it has a single parent node (y_A, y_B). Then it is unnecessary to store the result for (x_A, x_B) in CACHE as long as (y_A, y_B) is saved properly.

Intuitively, every time (x_A, x_B) is visited by the APPLY algorithm, it is through the node (y_A, y_B). As such, once the result for (y_A, y_B) is saved in the cache, (x_A, x_B) is never visited again. Consequently, due to the distribution of node in-degrees, we know that the majority of nodes in the product graph do not actually need to be stored in CACHE.

However, this proposition does not tell us how to detect these "redundant" cache entries. Furthermore, detecting all such entries appears to be a hard problem: when a product graph node is first visited, we do not know if it *can* be visited again from some other source. If we mistakenly exclude it from CACHE, it could result in re-computation of a non-trivial portion of the product graph.

Proposition 2: Assume x_A has a single parent in BDD A and x_B has a single parent in BDD B. Then (x_A, x_B) has a single parent in any product graph $A \times_\star B$.

While this certainly does not cover *all* redundant entries, such observation gives us a way of eliminating at least *some* of the redundant work. To implement it, we only need to maintain a simple 2-bit $\{0, 1, \texttt{many}\}$ parent counter for each BDD node. It is then easy to verify the conditions of our proposition during each CACHE access and to skip any unnecessary queries.

In our benchmark dataset, this criterion leads to an average 22% reduction in the number of product graph nodes stored in CACHE. However, we should note that this method is quite uneven: the eliminated node ratio ranges from more than 90% to less than 1% depending on the BDD. Nevertheless, due to its low overhead, we still consider it a worthwhile improvement.

In the future, it is possible to explore additional heuristics that eliminate a higher percentage of single-parent nodes from the operations cache.

Other cache table considerations: Aside from the aforementioned improvements, we use a relatively standard cache table design: (1) we overwrite results on collision; (2) we grow the table in exponents of two once the number of insertions exceeds the current `length`; (3) to determine the cache slot, we use log_2(`length`) most significant bits of the same prime multiplicative hash as for the node table, but the input is a single integer concatenation of x_A and x_B.

Note that when this hash is truncated to the same bit-length as the input, it is in fact *perfect*: it maps each input to a unique output [31][9]. As such, we only need to save the hash of (x_A, x_B) instead of the whole key. This does not necessarily reduce memory consumption, but there is no need to recompute the hash during table growth. Furthermore, when the table grows, this hash has a very predictable node placement since the new slot is a single-bit extension of the previous slot. Consequently, similar to the node table, growing the cache is essentially a memory copy operation.

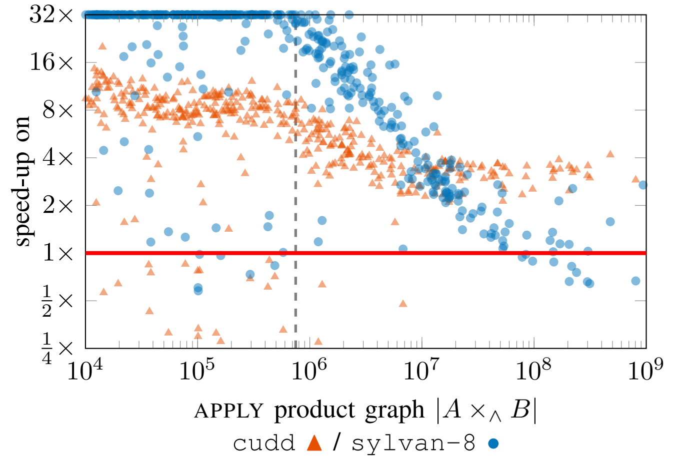

Fig. 5. The speed-up in runtime of our implementation compared to `cudd` and `sylvan-8`, truncated to $32\times$. Points below the red line represent slow-down instead of speed-up. Points to the right of the vertical line represent 200 largest benchmark instances. Note that both axis are logarithmic.

E. *Performance evaluation*

1) State-of-the-art BDD packages: To evaluate the performance of this approach, we compare the implementation to the `cudd` and `sylvan-8` runtime on our modern CPU (we omit `lib-bdd` as it is largely superseded by at least one of the methods on every benchmark). The effective speed-up is shown in Fig. 5. If we focus on the 200 largest benchmarks, our implementation achieves on average a $3.70\times$ speed-up compared to `cudd` and $5.69\times$ speed-up compared to `sylvan-8`. However, we see that especially for `sylvan`, the improvements are diminishing as the benchmark size grows.

Therefore, we also investigate the 10 largest benchmarks. Here, `sylvan-8` is sometimes faster than our implementation, however our method is still on average $1.03\times$ as fast as `sylvan-8`. Note that these are tasks that all consume at least 16 GBs and sometimes more than 32 or 64 GBs

[9]This hash is *similar* and often confused with the notorious *Knuth multiplicative hash* [25] and the so-called *binary multiplicative hash* [17]. However, these do not use a prime as the multiplier and are consequently not perfect [30].

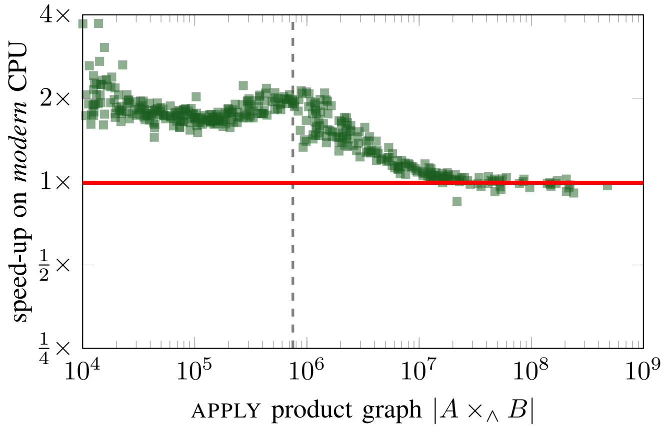

Fig. 6. The speed-up in runtime on the *modern* vs. the *legacy* system for our implementation. Points below the red line represent slow-down instead of speed-up. Points to the right of the vertical line represent 200 largest benchmark instances. Note that both axis are logarithmic.

of system memory over hundreds of millions of product graph nodes. This means that our implementation is better on small and medium tasks while achieving roughly comparable performance on very large tasks.

Furthermore, note that our implementation is always faster than `sylvan-4` (not shown in the figure), and on average $1.5\times$ faster on the 10 largest tasks. We also note that in our testing, `sylvan-4` was on average $2.27\times$ (up to $3.9\times$) faster and `sylvan-8` on average $3.22\times$ (up to $6.3\times$) faster than `sylvan-1` for the 200 largest benchmarks. While this is not perfect scaling, it appears to be roughly in line with previous reported results for `sylvan` [41].

We also investigated the performance counters to explain this improvement. For the 200 largest instances, we see average IPC of 1.26 (as opposed to `cudd`'s 0.35), while the L3 cache miss rate is only 27% compared to `cudd`'s 40%. Furthermore, our implementation needs on average only 12 L3 cache references per product graph node compared to `cudd`'s 21, and the number of instructions executed between such references increased to 42 from 14 on average. In other words, our implementation performs more work with fewer requests to the main memory, just as we were trying to achieve. For the top 10 benchmarks, these numbers are less ideal, with just 0.73 IPC and 32% cache miss rate, but this is still quite enough to achieve a sizeable improvement over `cudd`.

2) Modern hardware: Next, we investigate whether we achieved our initial goal. That is, whether we improved the scalability of the BDD APPLY algorithm on modern hardware. As such, we repeat the experiment from Section IV-A with our new implementation. The results are shown in Fig. 6, which is directly comparable to Fig. 3.

Here, we see that the speed-up is again diminishing with growing BDD size. However, we also see that the overall improvement is greater and more predictable compared to the other implementations. In particular, while we see a slight slow-down in some of the experiments, it is generally within the 5% measurement noise tolerance established earlier. For the whole dataset, we see an average speed-up of $1.72\times$, which is comparable to `lib-bdd` and `sylvan-1`. However, for the 200 largest benchmarks, we still have a speed-up of $1.26\times$, as opposed to slow-down for `lib-bdd` ($0.88\times$) and `sylvan-1` ($0.61\times$). Finally, zooming in on the 10 largest benchmarks, we have a small average slow-down of $0.95\times$, which is again (barely) within our 5% measurement tolerance.

3) Memory consumption and memory layout: Due to the prototype nature of our implementation, we have not thoroughly evaluated the memory consumption yet. However, we observed no significant differences between the packages when the memory was limited to 32GB. Specifically, each package ran out of memory on roughly the same handful of largest benchmark problems. We thus do not consider out method to be significant advantaged nor disadvantaged in this regard.

Finally, we should stress that in the presented setting, our method supersedes [29]. It benefits from the temporal ordering of BDD nodes, but further reduces the number of necessary memory accesses and collisions through the use of the prefix trie. However, while our method also *benefits* when the in-memory ordering of BDD nodes matches the temporal ordering, the in-memory ordering is not strictly required: as opposed to [29], the critical aspects of the temporal ordering are essentially stored in the prefix tries.

This begs the question: How important is the in-memory node ordering for our method? To test this, we prepared an experiment where we compare our method on BDDs preprocessed with three possible in-memory orderings: DFS preorder, DFS post-order, and randomly shuffled.

We find that when comparing pre-order and post-order, preorder is slightly faster, but the difference is $< 5\%$ and diminishes with increasing benchmark size. However, comparing post-order and shuffled node ordering, we observe that postorder is on average 26% faster, and almost 40% faster for the 200 largest benchmarks. However, this lead then diminishes for the largest 10 queries, where it is less than 10%, suggesting that the ordering is the most important for medium-sized BDDs that are "close" to the L3 cache capacity.

Hence we see that the improvements of our method are not completely dependent on the in-memory layout of the BDD nodes. The method is still better than `sylvan-4`, but not as good as `sylvan-8` for the largest benchmarks. However, the choice of memory ordering does measurably influence the outcome. Consequently, this information can inform the implementation of garbage collection algorithms for BDDs. It is not uncommon for garbage collection methods to reorder objects to improve memory locality [23]. Our results thus show this to be an important consideration for BDDs.

V. CONCLUSION

In this paper, we demonstrated the impact of memory latency on the performance of BDD packages. Specifically, we show that a more "modern" CPU does not necessarily guarantee improved performance once the size of the problem no longer fits into the L3 cache of the CPU.

We then proposed improvements to the node and cache table used within the APPLY algorithm with the goal of increasing locality and reducing the number of memory accesses overall. We demonstrate that with these improvements, our implementation significantly outperforms classical BDD packages like `cudd`, and is better or comparable to parallelization of the same task to 8 CPU cores using the package `sylvan`. Importantly, we also show that it is the only implementation in our testing that exhibits a consistent improvement in performance when comparing a modern and a legacy CPU.

However, we should stress that the results of this paper do not argue *against* parallelization of BDD operations. We simply use parallelism as a meaningful comparison that gives an alternative way of speeding up BDD operations. In fact, we believe that similar performance benefits can also translate to parallel BDD algorithms if appropriate data structures are developed based on our observations.

Additionally, we should note that the presented approach does not in any way prevent dynamic variable reordering: to swap two adjacent variables, we can swap the BDD nodes in-place, such as in [19]. We then update the parent-local trie for each affected node accordingly. This requires a delete operation on the trie structure, but such operation does not differ from deletion on normal tries.

Finally, we observe that latest hardware developments open new interesting propositions for improving BDD performance. First, several new CPUs utilize silicon die stacking or other advanced packaging techniques to significantly increase the L3 cache capacity or add another layer of L4 cache.

Second, we are currently experiencing a resurgence of task-specific hardware in the field of statistical machine learning and a great increase in capabilities of field programmable gate arrays (FPGAs). Perhaps an in-hardware APPLY implementation tuned for out-of-order exploration of the product graph can be designed such that it sufficiently hides the large latency of modern random access memory.

ACKNOWLEDGMENTS

This work was supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 101034413 and the "VAMOS" grant ERC-2020-AdG 101020093.

REFERENCES

[1] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 37–44, 2010.

[2] Junaid Babar, Gianfranco Ciardo, and Andrew Miner. CESRBDDs: binary decision diagrams with complemented edges and edge-specified reductions. *International Journal on Software Tools for Technology Transfer*, 24(1):89–109, 2022.

[3] Jiří Barnat, Jakub Chaloupka, and Jaco Van De Pol. Distributed algorithms for SCC decomposition. *Journal of Logic and Computation*, 21(1):23–44, 2011.

[4] Nikola Beneš, Luboš Brim, Jakub Kadlecaj, Samuel Pastva, and David Šafránek. AEON: attractor bifurcation analysis of parametrised Boolean networks. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*, pages 569–581. Springer, 2020.

[5] Nikola Beneš, Luboš Brim, Samuel Pastva, and David Šafránek. Computing bottom SCCs symbolically using transition guided reduction. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, pages 505–528. Springer, 2021.

[6] Nikola Beneš, Luboš Brim, Samuel Pastva, and David Šafránek. Symbolic coloured SCC decomposition. In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27*, pages 64–83. Springer, 2021.

[7] David Bergman, Andre A Cire, Willem-Jan Van Hoeve, and John Hooker. *Decision diagrams for optimization*, volume 1. Springer, 2016.

[8] Karl S Brace, Richard L Rudell, and Randal E Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45, 1991.

[9] Randal E Bryant. Graph-based algorithms for Boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[10] Randal E Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 236–243. IEEE, 1995.

[11] Luigi Capogrosso, Luca Geretti, Marco Cristani, Franco Fummi, and Tiziano Villa. HermesBDD: A multi-core and multi-platform binary decision diagram package. *arXiv preprint arXiv:2305.00039*, 2023.

[12] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Alexander Svozil. Symbolic time and space tradeoffs for probabilistic verification. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2021.

[13] ChipsAndCheese. AMD's 7950X3D: Zen 4 gets VCache. https://chipsandcheese.com/2023/04/23/amds-7950x3d-zen-4-gets-vcache/, Apr 2023. Accessed: 2023-05-01.

[14] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.

[15] Mahmoud Elbayoumi, Michael S Hsiao, and Mustafa ElNainay. A novel concurrent cache-friendly binary decision diagram construction for multi-core platforms. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1427–1430. IEEE, 2013.

[16] Stefan Ellmauthaler, Sarah Alice Gaggl, Dominik Rusovac, and Johannes P Wallner. Representing abstract dialectical frameworks with binary decision diagrams. In *Logic Programming and Nonmonotonic Reasoning: 16th International Conference, LPNMR 2022, Genova, Italy, September 5–9, 2022, Proceedings*, pages 177–189. Springer, 2022.

[17] Jeff Erickson. *Algorithms*. 2019.

[18] Eric Felt, Gary York, Robert Brayton, and Alberto Sangiovanni-Vincentelli. Dynamic variable reordering for BDD minimization. In *Proceedings of EURO-DAC 93 and EURO-VHDL 93-European Design Automation Conference*, pages 130–135. IEEE, 1993.

[19] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation.*, pages 50–54. IEEE, 1991.

[20] Daochuan Ge, Meng Lin, Yanhua Yang, Ruoxing Zhang, and Qiang Chou. Quantitative analysis of dynamic fault trees using improved sequential binary decision diagrams. *Reliability Engineering & System Safety*, 142:289–299, 2015.

[21] Justin E Harlow III and Franc Brglez. Design of experiments in BDD variable ordering: Lessons learned. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 646–652, 1998.

[22] Tobias Heß, Chico Sundermann, and Thomas Thüm. On the scalability of building binary decision diagrams for current feature models. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A*, pages 131–135, 2021.

[23] Xianglong Huang, Stephen M Blackburn, Kathryn S McKinley, J Eliot B Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. *ACM SIGPLAN Notices*, 39(10):69–80, 2004.

[24] Martin Jonáš and Jan Strejček. Solving quantified bit-vector formulas using binary decision diagrams. In *Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings 19*, pages 267–283. Springer, 2016.

[25] Donald Ervin Knuth. *The art of computer programming, volume 3: Sorting and searching*, volume 3. Pearson Education India, 1973.
[26] Lukas Kohutka and Peter Pistek. Faster synthesis of combinational logic based on multiplexer trees and binary decision diagrams. In *2014 IEEE 12th IEEE International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 239–244. IEEE, 2014.
[27] Jitendra Kumar, Yukio Miyasaka, Asutosh Srivastava, and Masahiro Fujita. Formal verification of integer multiplier circuits using binary decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.
[28] Casper Abild Larsen, Simon Meldahl Schmidt, Jesper Steensgaard, Anna Blume Jakobsen, Jaco van de Pol, and Andreas Pavlogiannis. A truly symbolic linear-time algorithm for SCC decomposition. In *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023*, pages 353–371. Springer, 2023.
[29] David E Long. The design of a cache-friendly BDD library. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 639–645, 1998.
[30] Memotut. Multiplicative hash is not perfect. https://memotut.com/en/aeefa085417b7134f793/.
[31] Mercari. Knuth multiplicative hash is the least complete hash function. https://engineering.mercari.com/blog/entry/2017-08-29-115047/, Oct 2017. Accessed: 2023-05-01.
[32] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, pages 272–277, 1993.
[33] AMY Miyasaka and M Fujita. A simple BDD package without variable reordering and its application to logic optimization with permissible functions. In *Proc. Int. Workshop Log. Synth*, pages 1–8, 2019.
[34] Jim Newton and Didier Verna. A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams. *ACM Transactions on Computational Logic (TOCL)*, 20(1):1–36, 2019.
[35] Wytse Oortwijn, Tom van Dijk, and Jaco van de Pol. Distributed binary decision diagrams for symbolic reachability. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 21–30, 2017.
[36] Samuel Pastva, David Safranek, Nikola Benes, Lubos Brim, and Thomas Henzinger. Repository of logically consistent real-world boolean network models. *bioRxiv*, pages 2023–06, 2023.
[37] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 254–264, 2011.
[38] Steffan Christ Sølvsten, Jaco van de Pol, Anna Blume Jakobsen, and Mathias Weller Berg Thomasen. Adiar binary decision diagrams in external memory. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022*, pages 295–313. Springer, 2022.
[39] Fabio Somenzi. CUDD: CU decision diagram package release 3.0.0. *URL: http://vlsi. colorado. edu/fabio/CUDD*, 4(3), 2015.
[40] Tom van Dijk, Ernst Moritz Hahn, David N Jansen, Yong Li, Thomas Neele, Mariëlle Stoelinga, Andrea Turrini, and Lijun Zhang. A comparative study of BDD packages for probabilistic symbolic model checking. In *Dependable Software Engineering: Theories, Tools, and Applications: First International Symposium, SETTA 2015, Nanjing, China, November 4-6, 2015, Proceedings 1*, pages 35–51. Springer, 2015.
[41] Tom Van Dijk and Jaco Van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19:675–696, 2017.
[42] Miroslav N Velev and Ping Gao. Efficient parallel gpu algorithms for BDD manipulation. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 750–755. IEEE, 2014.
[43] Alexander Von Rhein, Sven Apel, and Franco Raimondi. Introducing binary decision diagrams in the explicit-state verification of Java code. In *Proc. Java Pathfinder Workshop*, volume 82, page 2, 2011.
[44] Liudong Xing, Ola Tannous, and Joanne Bechta Dugan. Reliability analysis of nonrepairable cold-standby systems using sequential binary decision diagrams. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 42(3):715–726, 2011.
[45] Yang Zhao and Gianfranco Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. *Innovations in Systems and Software Engineering*, 7:141–150, 2011.