EDDY: A Multi-Core BDD Package With Dynamic Memory Management and Reduced Fragmentation

Rune Krauss
University of Bremen
Bremen, Germany
krauss@uni-bremen.de

Mehran Goli
University of Bremen
Bremen, Germany
mehran@uni-bremen.de

Rolf Drechsler
University of Bremen / DFKI
Bremen, Germany
drechsler@uni-bremen.de

ABSTRACT

In recent years, hardware systems have significantly grown in complexity. Due to the increasing complexity, there is a need to continuously improve the quality of the hardware design process. This leads designers to strive for more efficient data structures and algorithms operating on them to guarantee the correct behavior of such systems through verification techniques like model checking and meet time-to-market constraints. A *Binary Decision Diagram* (BDD) is a suitable data structure as it provides a canonical compact representation of Boolean functions, given variable ordering, and efficient algorithms for manipulating them. However, reduced ordered BDDs also have challenges: There is a large memory consumption for the BDD construction of some complex practical functions and the use of realizations in the form of BDD packages strongly depends on the application.

To address these issues, this paper presents a novel multi-core package called *Engineer Decision Diagrams Yourself* (EDDY) with dynamic memory management and reduced fragmentation. Experiments on BDD benchmarks of both combinational circuits and model checking show that using EDDY leads to a significantly performance boost compared to state-of-the-art packages.

CCS CONCEPTS

• **Applied computing → Computer-aided design**;
• **Hardware → Hardware validation**;
• **Mathematics of computing → Graph theory**;
• **Computing methodologies → Parallel algorithms**.

KEYWORDS

Boolean functions, binary decision diagrams, model checking, memory management, parallel computing

ACM Reference Format:
Rune Krauss, Mehran Goli, and Rolf Drechsler. 2023. EDDY: A Multi-Core BDD Package With Dynamic Memory Management and Reduced Fragmentation. In *28th Asia and South Pacific Design Automation Conference (ASPDAC '23), January 16–19, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3566097.3567913

1 INTRODUCTION

Very Large Scale Integration (VLSI) circuits with up to several billion transistors are present in many technical devices today. Technological progress has increased so that a multitude of daily tasks in our society are now supported by computer systems. Examples are the anti-lock braking system in cars and monitoring in medicine. Due to the increased complexity, a VLSI design is no longer possible without *Computer-Aided Design* (CAD), making it an essential part of the design process [17].

In order to meet user requirements, it is necessary to guarantee the quality of the increasingly complex hardware design process through continuous algorithmic improvements in the field of verification. Model checking [2] is an important approach to assessing the correctness of systems through state exploration and property checking. Algorithms and data structures for representing automata were originally implemented with consideration of explicit states [10]. Thus, only automata with at most 10^3 to 10^6 reachable states could be processed [8]. However, real models have billions of states, not all of which can be considered in a reasonable amount of time [7]. Therefore, *Reduced Ordered Binary Decision Diagrams* (BDDs), specified by R. Bryant [4] in 1986, are suitable as they can compactly encode Boolean functions and allow efficient algorithms such as And-Exist and reachability analysis, leading to a breakthrough in model checking [6, 16].

Subsequently, improvements have been researched, especially in terms of efficient implementation of BDDs [3]. These include, i. a., hash-based *Unique Tables* (UTs) with collisions resolved by chaining for storing nodes, hash-based *Computed Tables* (CTs) for caching nodes, *Garbage Collections* (GCs) for clearing dead nodes, and complemented edges for efficiently negating functions [12]. In summary, these concepts can be found in BDD packages [18, 22, 26] that are used in CAD tools such as NuSMV [9].

Although a BDD is an efficient data structure for Boolean functions and provides efficient algorithms for manipulating them, there are still some challenges to overcome: There is a large memory consumption for the BDD construction of some complex practical functions such as multipliers [5] and TCAS [9], as well as the respective package performance is strongly dependent on the problem domain such as a model. Studies such as [25] have shown that the model dependency of packages in terms of performance is mainly due to memory management: (1) locks have a negative impact on the speedup of parallel programs [19], (2) static CT size can have a negative impact on BDD applications [28], and (3) while GCs and variable reorderings can reduce the number of nodes existing at runtime, they are generally time-consuming [23]. In addition, pointer-based approaches complicate debugging and may even increase the node size depending on the architecture [14].

To address the aforementioned issues, in this paper, we present *Engineer Decision Diagrams Yourself* (EDDY), a novel index-based multi-core BDD package based on [13], with dynamic memory management and reduced fragmentation. Experiments on BDD benchmarks of both combinational multilevel circuits and model checking confirm that using EDDY leads to a significantly performance boost compared to state-of-the-art BDD packages. In this context, EDDY is on average about three times faster with overall lower memory usage.

In summary, the main contributions of this paper are as follows:

(1) Multithreading for parallel synthesis and concurrent access to BDDs;
(2) Lock-free CT with dynamic caching for faster computations;
(3) Delayed GC with fragmentation handling.

The rest of this paper is organized as follows: We summarize fundamentals on BDDs and give an overview of existing BDD packages in Section 2. Section 3 presents the proposed approaches that are realized in EDDY. In Section 4, the experimental setup and results are shown. Finally, Section 5 concludes the paper and discusses possible future work.

2 BACKGROUND

In this section, important fundamentals and formal notations are introduced in an attempt to keep this work self-contained. First, Section 2.1 presents basic concepts for understanding Boolean functions and BDDs as one of their representations. Then, in Section 2.2, state-of-the-art BDD packages are briefly discussed in terms of implementation techniques.

2.1 Preliminaries

In digital circuits, signals can be symbolized as variables $x_1, \dots, x_n$ that take logical values from $\mathbb{B} := \{0, 1\}$. In propositional logic [27], 0 $(1) \in \mathbb{B}$ is interpreted as `false` (`true`). Therefore, output signals whose values are uniquely specified by input signals can be described by *Boolean functions.*

Definition 1. A mapping $f : \mathbb{B}^n \to \mathbb{B}^m$ is called a *Boolean function*, where $n, m \in \mathbb{N}$. $\mathcal{B}_{n,m} := \{f \mid f : \mathbb{B}^n \to \mathbb{B}^m\}$ describes the set of Boolean functions, where $\mathcal{B}_n := \mathcal{B}_{n,1}$.

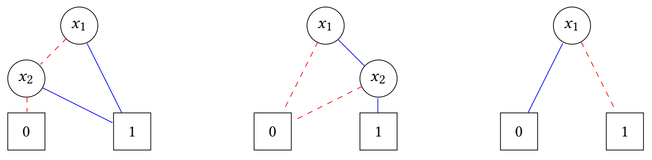
The *Boolean calculus* [20] and, i. a., methods developed for e. g. circuit analysis by C. Shannon [21] are the basis for today's digital computer systems and allow computations with Boolean functions as well as their manipulation.

Definition 2. The quadruple $(\mathcal{B}_{n,m}, +, \cdot, \overline{})$ with

$$f + g \in \mathcal{B}_{n,m} := (f + g)(\alpha) = f(\alpha) \vee g(\alpha) \forall \alpha \in \mathbb{B}^n$$
$$f \cdot g \in \mathcal{B}_{n,m} := (f \cdot g)(\alpha) = f(\alpha) \wedge g(\alpha) \forall \alpha \in \mathbb{B}^n$$
$$\overline{f} \in \mathcal{B}_{n,m} := \overline{f}(\alpha) = 1 \iff f(\alpha) = 0 \forall \alpha \in \mathbb{B}^n$$

is called the *Boolean algebra of functions.*

If a circuit is modular, an application such as model checking can first compute representations for modules in order to subsequently combine them. The basis for such computations with Boolean functions is provided by *Shannon expansion.*



(a) Disjunction **(b) Conjunction** **(c) Negation**

Figure 1: BDDs representing the Boolean basis functions

Definition 3. Let $f \in \mathcal{B}_n$ be a n-ary Boolean function. By partitioning f to x_i with $f_{x_i=1}(\alpha_1, \alpha_2, \dots, \alpha_{i-1}, 1, \alpha_{i+1}, \alpha_{i+2}, \dots, \alpha_n)$ and $f_{x_i=0}(\alpha_1, \alpha_2, \dots, \alpha_{i-1}, 0, \alpha_{i+1}, \alpha_{i+2}, \dots, \alpha_n)$ $\forall \alpha \in \mathbb{B}^n$:

$$f = x_i \cdot \underbrace{f_{x_i=1}}_{\text{positive cofactor}} + \overline{x_i} \cdot \underbrace{f_{x_i=0}}_{\text{negative cofactor}}.$$

If variables are successively decomposed using Definition 3 while respecting a total order π and avoiding redundancies/isomorphisms by exploiting laws of Boolean algebra, a *BDD* results.

Definition 4. A *BDD* is a directed acyclic graph $G = (V, E)$ over variables $X_n := \{x_1, \dots, x_n\}$ and a value set $\mathbb{B}$. Each node is assigned such a label, where a Boolean function is interpreted as follows:

If v is labeled with $b \in \mathbb{B}$, then the leaf describes the constant function that maps each argument to b.

If v is an inner node, it is labeled with $x_i \in X_n$, where the variable is decomposed using Definition 3, respecting a total order $\pi : x_1 < x_2 < \dots < x_n$.

The edge set E consists of all pairs (v, v'), with the child v' referenced by the parent v.

If, in addition, $(f_v)_{x_i} \neq (f_v)_{\overline{x_i}} \forall v \in V$ and no distinct nodes $v, w \in V$ exist which are labeled with the same variable and whose children are identical, then G is reduced.

Example 1. BDDs for the Boolean basis functions $+, \cdot, \overline{} \in \mathcal{B}_2$ are given in Figure 1: (1) disjunction (Figure 1a), (2) conjunction (Figure 1b), and (3) negation (Figure 1c).

Remark. The referencing is typically drawn using solid edges (1-edges) and dashed edges (0-edges).

The successive "top-down" use of Definition 3 requires repeatedly performing an equivalence test to check whether subfunctions are already represented [17]. A more efficient way is to consider nodes as decisions and combine (synthesize) them via

$$f \otimes g = x_i \cdot (f_{x_i=1} \otimes g_{x_i=1}) + \overline{x_i} \cdot (f_{x_i=0} \otimes g_{x_i=0}),$$

where $\otimes \in \mathcal{B}_2$. These operations can be generally traced back to the ternary operator

$$ite(f, g, h) = f \cdot g + \overline{f} \cdot h \tag{1}$$

such as $f + g = ite(f, 1, g)$, which is compatible with Definition 3 because of

$$f \cdot g + \overline{f} \cdot h = ite(x_i, ite(f_{x_i}, g_{x_i}, h_{x_i}), ite(f_{\overline{x_i}}, g_{\overline{x_i}}, h_{\overline{x_i}})).$$

Such a recursive formulation allows efficient storing/indexing of BDDs and forms the basic structure of the so-called *UT*, where each node is stored as a triple (variable, children).

2.2 Related Work

The efficiency of BDD syntheses and related algorithms, as well as data structures, can be increased using different techniques beyond the theoretical point of view described in the last section. Therefore, based on ideas in [3], a lot of research work has been conducted in BDD packages in the last decades. A comprehensive survey is available in [14]. In the following, we briefly explain the widely used state-of-the-art packages with regard to their differences.

CUDD [22] stands for "Colorado University Decision Diagram" and is programmed in C/C++. Nodes are referenced pointer-based respecting complemented edges. Nodes that are no longer needed are cleared by a GC using reference counting. In addition, numerous operations, including finding a "good" variable order, are supported, where they share a CT and can coexist.

BuDDy [18] is written in C/C++, where nodes are referenced index-based. Nodes can be cleared by a mark-and-sweep GC. The package provides the most used operations for BDD manipulation, including dynamic reordering. Several CTs are used for operations.

Sylvan [26] is a BDD library written in C that can perform basis BDD operations in parallel. Moreover, the most commonly used operations like And-Exist are supported here as well.

Although the aforementioned packages share some conceptual similarities with EDDY, they come with some drawbacks in terms of (1) sequential performing of operations or increased use of locks, (2) static caching, and (3) time-consuming GCs. Hence, the main goal of this work is to overcome these limitations.

Algorithm 1: Parallel BDD synthesis using *ite*

```
Input: BDDs $f, g, h$
Output: BDD for $ite(f, g, h)$
 1  if $f = 1 \vee g = h$ then
 2      return $g$
 3  end if
 4  if $f = 0$ then
 5      return $h$
 6  end if
 7  ...                                        ▷ other terminal cases
 8  if $ct.has\_entry(f, g, h)$ then
 9      return $ct(f, g, h)$
10  end if
11  $x \leftarrow$ top variable of $f, g, h$
12  $future\_t \leftarrow async(ite, f_{x_i}, g_{x_i}, h_{x_i})$
13  $t \leftarrow future\_t.get()$
14  $future\_e \leftarrow async(ite, f_{\overline{x_i}}, g_{\overline{x_i}}, h_{\overline{x_i}})$
15  $e \leftarrow future\_e.get()$
16  if $t = e$ then
17      return $t$
18  end if
19  $r \leftarrow ut.find\_or\_add(x, t, e)$
20  $ct.insert(f, g, h, r)$
21  return $r$
```

3 ENGINEER DECISION DIAGRAMS YOURSELF (EDDY)

In this section, we present our main novel approaches in detail, summarized as a BDD package called *EDDY* based on [13], which is pointerless because of easier debugging and a node size that is independent of the respective architecture. It supports common operations, including multiple-operand functions like And-Exist, and is therefore useful for model checking.

For this purpose, Section 3.1 forms the multi-core support for the memory management consisting of dynamic caching and delayed GC with fragmentation handling, which are then described in Section 3.2 and Section 3.3.

3.1 Multithreading for Parallel Synthesis

In general, BDD syntheses and related operations are performed sequentially in packages such as [22] and [18]. However, it was shown in [25] that a significant speedup is possible by parallel computation. Traditionally, concurrency issues such as data races are solved by locks, providing mutual exclusion but also potentially decelerating a system [11].

Thus, to address issue (1) listed in Section 2.2, the synthesis operator *ite* (Equation 1) is parallelized w. l. o. g., oriented to [26], but with minimization of locks using automatic thread management.

Algorithm 1 presents the pseudocode of the parallelized *ite* operation based on [3]. The function takes three arguments which are indices to a respective BDD. Terminal cases are checked in Lines 1–7. Using an operation cache, Lines 8–10 check for an already computed operand combination and – if existing – return it. Otherwise, two cofactors are computed in parallel in Lines 11–15, decomposing according to the previously determined variable of order. Since each call represents an independent task, both recursive function calls can be parallelized. Thus, multiple threads can be easily used: A thread is automatically started using *async* and waits until the result as a future object can be obtained by *get*. Both routines are high-level wrappers of the C++ standard library, i. e. the task is automatically decoupled from the result.

Remark. The described use of multiple threads is also possible for BDD operations such as sat_{count} and sat_{all} [24], where the traversal is similar to *ite*.

After computing both subproblems, Lines 16–18 check for isomorphism. In Line 19, canonicity is atomically ensured by either finding or adding the computed triple in the hash table *ut*, respecting complemented edges. To avoid undefined behavior, synchronization between the threads with respect to the BDDs contained globally in the shared UT is sufficient here. In addition, Line 20 stores the result in the CT, followed by the return of the index for accessing the computed BDD.

3.2 Lock-Free CT With Dynamic Caching

Generally, the CT size is statically adjusted in BDD packages. For example, in [22] there is a "reward-based" policy where the CT size is doubled when a large cache hit rate (30 %) is observed, unless a predefined maximum size has been reached. In [18], the CT size is doubled when the load factor of the UT equals 100 % and there are no more dead nodes than a maximum of 20 %. Reasons for these strategies are, i. a., a higher chance that valuable cache hits survive longer, especially to solve large problems, and, secondly, the effort to achieve optimal utilization of the CT in relation to

Algorithm 2: Caching of entries (*insert*) in the CT

```
Input: Subentries $a, b, c, r$
1 $index \leftarrow hash(a, b, c) \mod size$
2 $slot \leftarrow entries[index]$
3 $spinlock(slot)$
4 $assign(a, b, c, r)$
```

Algorithm 3: CT adjustment based on the hit rate

```
1 if $count \geq \delta$ then
2     if $ct.curr\_hit\_rate() \geq ct.old\_hit\_rate()$ then
3         $ct.expand()$
4     end if
5     $ct.old\_hit\_rate() \leftarrow ct.curr\_hit\_rate()$
6     $\delta \leftarrow count \cdot 2$
7 end if
```

Algorithm 4: CT adjustment in *find_or_add*

```
Input: Variable $x$ and BDDs $t, e$
Output: BDD found or added
1 $assert(t \neq e)$
2 ...                          ▷ search for BDD and return if found
3 if $\beta \geq \beta_{\max}$ then
4     ...                      ▷ UT adjustment
5     $ct.expand()$
6 end if
7 ...                          ▷ make node
8 return BDD added
```

the UT respecting GCs. However, since the so-called *large cache hit rate* is not dynamically adjusted at runtime or, respectively, the maximum load factor is not observed, such strategies can have a negative impact on BDD applications and there is also a higher collision probability [15, 28].

Therefore, to address issue (2) listed in Section 2.2, CTs for BDD operations are dynamically adjusted in EDDY. The CT as a data structure is developed using the lock-free `atomic_flag` data type of the C++ standard library. Critical sections in functions with a minimal amount of work, such as Algorithm 2 for caching entries, are developed with spinlocks using `atomic_flag`, since there are fewer context switches compared to mutexes [1].

To dynamically adjust the hit rate of CTs for operations, a threshold value δ (initial value corresponds to the predetermined CT size) is introduced in Algorithm 3 that triggers such an adjustment when the respective function call counter *count* reaches this value (Line 1). When the current hit rate reaches the old hit rate, the CT is doubled (Lines 2–4). Therefore, the hit rate is subsequently updated and δ is adjusted by $count \cdot 2$ (Lines 5–6).

Remark. It is sufficient to put Algorithm 3 at the beginning of a BDD operation such as *ite* and sat_{count}, which supports a CT for caching results.

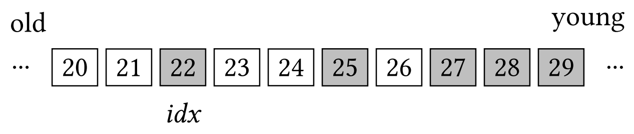


Figure 2: Node memory and the concept of GC

To reduce collisions and increase the chance for optimal utilization of CT in relation to UT, the maximum load factor $\beta_{\max}$ (70 % according to [15]) is introduced in Algorithm 4 for finding and adding BDDs. In this context, canonicity is first ensured (Line 1). It is checked whether a triple already exists and a BDD node is returned immediately if a corresponding triple is found in the UT (Line 2). Otherwise, when the current load factor β of the UT reaches the defined threshold value (Line 3), both the UT and CT are adjusted (Lines 4–6). A BDD node is then created, added to the UT, and returned (Lines 7–8).

3.3 Delayed GC With Fragmentation Handling

The GC type used in a BDD package is reference counting [22] or mark-and-sweep [18]. While more memory is required for nodes when using reference counting, as well as there is an overhead due to incrementing/decrementing the counters, a mark-and-sweep GC is more complicated to implement and can lead to increased fragmentation of available memory [13]. Regardless of type, GC is classically triggered based on the percentage of dead nodes. However, in applications such as model checking, a high rebirth rate usually exists [28].

To address issue (3) listed in Section 2.2, reducing performing of syntheses due to recently cleared nodes and therefore to reuse subresults, a delayed mark-and-sweep GC with fragmentation handling is proposed.

Due to the high rebirth rate mentioned, the GC should be delayed as long as possible, i. e. it is triggered automatically when the free physical memory is almost exhausted. In the mark phase, the living (reachable) BDD nodes are marked via a depth-first search. Bit manipulation is used for this by setting the most significant bit of the variable. In the sweep phase, any dead (not reached) node is cleared. For this purpose, the UT is iterated backward so that the index *idx* for choosing the next node slot points to the vacated location in the direction of the beginning of the node memory, which is exemplarily illustrated in Figure 2. Since *idx* is set to the next free available slot after occupancy and so on, this automatically overwrites possible present memory fragments with nodes in the further process.

4 EXPERIMENTAL RESULTS

This section summarizes the experiments conducted to empirically analyze EDDY and demonstrate the benefits of the approaches proposed in the last section. To this end, Section 4.1 discusses the configured system specification and benchmark instances used for the following performance evaluations. Section 4.2 investigates the impact of the approaches by comparing them against state-of-the-art BDD packages to test their suitability for applications such as model checking.

Table 1: Experimental comparison of EDDY and state-of-the-art BDD packages in terms of runtime and memory usage

Instance	CUDD		BuDDy		Sylvan		EDDY	
	T	M	T	M	T	M	T	M
c2670	12	297	11	252	11	271	**4**	241
c3540	9	504	8	403	9	446	**5**	308
c6288-14	37	1,637	67	4,109	38	1,441	**29**	1,295
c6288-15	127	4,753	265	12,569	134	4,477	**94**	3,412
c6288-16	405	13,605	—	MO	417	12,847	**334**	10,603
dpd75	492	176	1,008	308	402	152	**32**	482
ftp3	41	301	39	288	339	52	**13**	424
mmgt20	328	118	477	255	337	112	**12**	249
over12	89	295	65	198	79	232	**21**	327
tcas	298	4,850	TO	—	332	5,070	**172**	3,448

T	Runtime in sec
M	Memory usage in MB
TO	*Time Out*
MO	*Memory Out*

4.1 Experimental Setup

To evaluate the proposed approaches, they were implemented in C++20. For performance evaluation, EDDY is compared to state-of-the-art BDD packages discussed in Section 2.2 in terms of runtime T (in sec) and memory usage M (in MB). To demonstrate the effectiveness of memory management, EDDY was modified by replacing its memory management with the one of the index-based package BuDDy, which is called *EDDY**. In addition, to show the speedup $S_n = \frac{T_1}{T_n}$ due to multi-core support, we compare EDDY with Sylvan using n threads. To allow a fair comparison, the initial size of the CT (UT) was set to the same value 2^{18} (2^{20}) according to [18], taking into account the respective problem domain complexity. The used variable order follows the order of appearance in the respective file.

The used benchmark instances are taken from the standard *ISCAS-85*[1] and *SMV traces*[2] benchmark sets, as they are representative in the context of combinational multilevel circuits and model checking applications [28].

All evaluations were carried out on a Fedora 28 machine with an Intel Xeon E3-1270 v3 CPU with 3.5 GHz (up to 3.9 GHz boost) and 32 GB of main memory. For each instance considered, 10 runs were performed and then the average was calculated. The *Time Out* (TO) was set to 30 min, whereas the *Memory Out* (MO) was configured to 16 GB.

4.2 Performance Evaluation

First, EDDY is compared to the state-of-the-art BDD packages based on the experiments performed using the selected benchmark instances, which are shown in Table 1.

The experimental results show that EDDY is more efficient than the state-of-the-art packages. For all considered benchmark instances of ISCAS-85 and SMV traces, EDDY is faster. In addition,

[1]The instances are available at https://web.eecs.umich.edu/~jhayes/iscas.restore/benchmark.html (successfully accessed October 31, 2022).
[2]The instances are available at https://nusmv.fbk.eu/examples/examples.html (successfully accessed October 31, 2022).

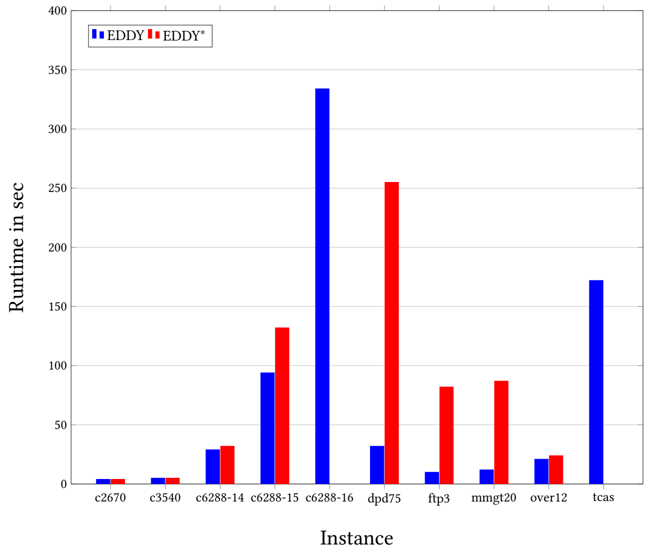


Figure 3: Performance comparison between EDDY and EDDY* using ISCAS-85 benchmarks and SMV traces

EDDY consumes less memory for each ISCAS-85 benchmark instance. While EDDY for e. g. *ftp3* uses minimally more memory compared to the other packages due to the delayed GC, it solves the problem significantly faster due to fewer GCs performed, making it a successful compromise. Moreover, EDDY is stable in solving problems, meaning *over12* is solved in the best case in 20 sec, and the worst case corresponds to 23 sec. For example, with CUDD, the best case is 72 sec, and the worst case corresponds to 101 sec, caused by increased hash collisions. Considering only the model checking instances, EDDY is on average about six times faster in comparison to the other BDD packages. Considering all benchmark instances, EDDY is on average about three times faster with overall lower memory usage.

The main reason for the experimental results shown for EDDY is its dynamic memory management, the effectiveness of which is demonstrated in Figure 3. For this purpose, EDDY is compared with EDDY*, which uses the memory management of BuDDy. It can be seen that EDDY is consistently better than EDDY* when considering the benchmarks of ISCAS-85 and SMV traces. Moreover, EDDY can solve any instance.[3]

The multi-core support in EDDY can further increase its performance, which is shown by comparing Figure 4 with Figure 5 considering the speedup achieved for the benchmark instances used. First, they show that larger circuits and models have higher speedups for both Sylvan and EDDY. However, due to the lock minimization in EDDY, its speedup is higher on average. In this context, the speedup increases with the number of threads. For example, if eight threads are considered, on average there is a speedup of 3.1 in EDDY, which is higher compared to 2.7 in Sylvan.

In summary, these results clearly confirm that the proposed approaches meet the objectives of this work.

[3]If no bar is specified, a TO or MO exists, respectively.

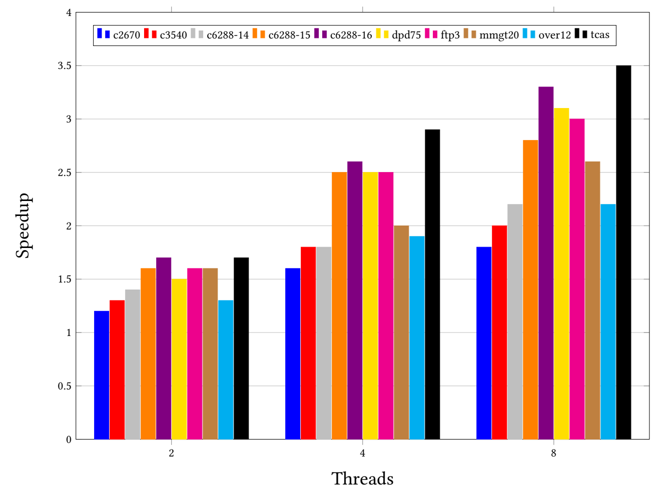


Figure 4: Parallel speedup of Sylvan based on problem complexity and number of threads

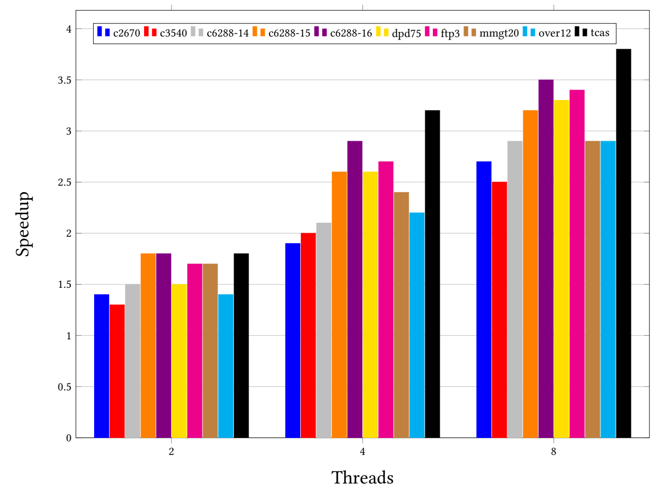


Figure 5: Parallel speedup of EDDY based on problem complexity and number of threads

5 CONCLUSION

In this paper, we presented EDDY, an index-based BDD package with novel approaches: (1) operations with automatic thread management, (2) lock-free CT with dynamic caching, and (3) delayed GC with fragmentation handling. Our experimental results demonstrated that EDDY outperforms the state-of-the-art packages due to its memory management. EDDY is on average about three times faster with overall lower memory usage and a possible average speedup of 3.1. Besides supported functions like And-Exist, EDDY allows verification tasks like model checking.

We believe that the efficiency of packages can be further improved. Therefore, parameters for the adjustment of hash tables will be investigated in the future. Furthermore, the proposed approaches can be integrated into other packages. Finally, various studies can be made using further benchmark instances.

REFERENCES

[1] T. Anderson. 1990. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1, 1 (1990), 6–16. https://doi.org/10.1109/71.80120

[2] C. Baier and J. Katoen. 2008. *Principles of Model Checking*. The MIT Press.

[3] K. Brace, R. Rudell, and R. Bryant. 1991. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. ACM, New York, NY, USA, 40–45. https://doi.org/10.1145/123186.123222

[4] R. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* 35, 8 (1986), 677–691.

[5] R. Bryant. 1991. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. Comput.* 40, 3 (1991), 205–213. https://doi.org/10.1109/12.73590

[6] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. 1994. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13, 4 (1994), 401–424. https://doi.org/10.1109/43.275352

[7] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. 1992. Symbolic model checking: 10^{20} States and beyond. *Information and Computation* 98, 2 (1992), 142–170. https://doi.org/10.1016/0890-5401(92)90017-A

[8] S. Chaki and A. Gurfinkel. 2018. *BDD-Based Symbolic Model Checking*. Springer. 219–245 pages.

[9] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. 2000. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 410–425. https://doi.org/10.1007/s100090050046

[10] E. Clarke, E. Emerson, and A. Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* 8, 2 (1986), 244–263. https://doi.org/10.1145/5397.5399

[11] G. Cong and D. Bader. 2004. Lock-Free Parallel Algorithms: An Experimental Study. In *Proceedings of the 11th International Conference on High Performance Computing*. Springer, Berlin, Heidelberg, 516–527. https://doi.org/10.1007/978-3-540-30474-6_54

[12] R. Drechsler and D. Sieling. 2001. Binary decision diagrams in theory and practice. *International Journal on Software Tools for Technology Transfer* 3, 2 (2001), 112–136. https://doi.org/10.1007/s100090100056

[13] G. Janssen. 2001. Design of a Pointerless BDD Package. In *International Workshop on Logic and Synthesis*. 310–315.

[14] G. Janssen. 2003. A consumer report on BDD packages. In *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design*. IEEE Computer Society, USA, 217–222. https://doi.org/10.1109/SBCCI.2003.1232832

[15] T. Mailund. 2019. *The Joys of Hashing*. Springer. 21–47 pages.

[16] K. McMillan. 1993. *Symbolic Model Checking*. Springer.

[17] C. Meinel and T. Theobald. 2012. *Algorithms and Data Structures in VLSI Design*. Springer.

[18] J. Nielsen. 1996. *BuDDy: A BDD package*. http://buddy.sourceforge.net/manual

[19] R. Rajwar and J. Goodman. 2002. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 5–17. https://doi.org/10.1145/605397.605399

[20] J. Seiffertt. 2017. *Boolean Algebra*. Springer. 11–24 pages.

[21] C. Shannon. 1938. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers* 57, 12 (1938), 713–723. https://doi.org/10.1109/T-AIEE.1938.5057767

[22] F. Somenzi. 1995. *CUDD: CU Decision Diagram Package*. http://web.mit.edu/sage/export/tmp/y/usr/share/doc/polybori/cudd

[23] F. Somenzi. 2001. Efficient manipulation of decision diagrams. *International Journal on Software Tools for Technology Transfer* 3, 2 (2001), 171–181. https://doi.org/10.1007/s100090100042

[24] T. Toda and K. Tsuda. 2015. BDD Construction for All Solutions SAT and Efficient Caching Mechanism. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, New York, NY, USA, 1880–1886. https://doi.org/10.1145/2695664.2695941

[25] T. van Dijk, E. Hahn, D. Jansen, Y. Li, T. Neele, M. Stoelinga, A. Turrini, and L. Zhang. 2015. A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking. In *Proceedings of the First International Symposium on Dependable Software Engineering*. Springer, Berlin, Heidelberg, 35–51. https://doi.org/10.1007/978-3-319-25942-0_3

[26] T. van Dijk and J. Pol. 2015. Sylvan: Multi-Core Decision Diagrams. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 677–691.

[27] J. Wang and W. Tepfenhart. 2019. *Propositional Logic*. Taylor & Francis. 83–114 pages.

[28] B. Yang, R. Bryant, D. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. Ranjan, and F. Somenzi. 1998. A Performance Study of BDD-Based Model Checking. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*. Springer, Berlin, Heidelberg, 255–289.