Efficient Binary Decision Diagram Manipulation by Reducing the Number of Intermediate Nodes

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*—The complexity of hardware systems has increased significantly in recent decades. Due to increasing user requirements, there is a need to develop more efficient data structures and algorithms to guarantee the correct behavior of such systems. A *Reduced Ordered Binary Decision Diagram* (BDD) is a suitable data structure as it represents all Boolean functions canonically given a variable order as well as provides algorithms for efficient manipulation. However, BDDs also have challenges: practicability depends on their minimization and there is a large memory consumption for some complex functions.**

To address these issues, this work investigates the number of emerged intermediate nodes that are not used in the final BDD result and presents a novel approach for efficient BDD manipulation by reducing the number of such nodes. Experiments on BDD benchmarks show that peak BDD node sizes can be significantly reduced, leading to accelerated BDD manipulation.

***Index Terms*—Boolean functions, binary decision diagrams, software packages, formal verification, model checking**

I. Introduction

Moore's law describes that the number of transistors in *Integrated Circuits* (ICs) doubles every two years [1]. Due to technological progress, billions of transistors are nowadays present in *Very Large Scale Integration* (VLSI) circuits that can be found, i. a., in smartphones used by today's society. As ICs become more complex, VLSI design cannot be created without *Computer-Aided Design* (CAD), making it an essential part of the hardware design process known as *VLSI CAD* [2].

In order to meet time-to-market constraints and to guarantee the quality of VLSI CAD, continuous algorithmic improvements in the field of verification are necessary. Model checking is an important approach to assess the correctness of hardware systems through state exploration and property checking [3]. Traditionally, data structures have been implemented that explicitly consider system states [4]. Thus, only automata with at most 10^6 reachable states could be processed [5]. However, since real models consist of billions of states, they cannot be considered in a reasonable amount of time [6]. *Reduced Ordered Binary Decision Diagrams* (BDDs) [7] are suitable for this application as they can compactly encode Boolean functions and allow efficient algorithms, such as reachability analysis, leading to a breakthrough in this technique [8].

This work was supported by DFG within the Reinhart Koselleck Project PolyVer (DR 287/36-1).

979-8-3503-3277-3/23/$31.00 ©2023 IEEE

Therefore, extensive research has been conducted to improve verification techniques by developing components that are combined in BDD packages [9]. Components include, i. a., BDD manipulation by the *If-Then-Else* (ITE) algorithm, hash-based *Computed Tables* (CTs) for caching operations, *Unique Tables* (UTs) with collisions resolved by chaining to identify all existing nodes, and *Garbage Collection* (GC) methods to remove unused (dead) nodes [10].

Although a BDD is an efficient data structure for Boolean functions, there are existing challenges: practicability depends on their minimization and there is a large memory consumption for some complex functions like multipliers [11], [12]. Firstly, the final BDD size of many functions like adders depends on the variable order [13], [14]. Secondly, when logical operations are performed repeatedly during BDD manipulation, the number of UT nodes temporarily used for final BDD construction, called *Intermediate Nodes* (Inodes), can dramatically increase, leading to memory overflow, so that subsequent operations such as reachability analysis cannot be performed [15], [16], [17], [18], [19], [20].

To address these issues, in this paper we investigate the emergence of Inodes and their number during BDD manipulations. Based on these results, we propose a novel approach for efficient BDD manipulation by reducing the number of Inodes. Experiments confirm that using this approach can significantly reduce the greatest number of nodes in use at any point during the process lifetime, leading to accelerated BDD manipulation that is on average about 20 % faster compared to the single use of the ITE algorithm and related work.

In summary, the main contributions are as follows:

1) Investigation of the number of Inode emergences using Boolean functions;
2) Development of an approach to reduce Inodes and accelerate BDD manipulation;
3) Approach evaluation and comparison with related work.

This paper is organized as follows: We summarize the Boolean function fundamentals and related work in Section II. Section III investigates Inode emergences and describes the proposed approach based on observations. In Section IV, the efficiency of this approach is evaluated and experimental results are discussed. Finally, Section V concludes the paper.

II. Background

This section introduces important fundamentals in an attempt to keep this work self-contained. While Section II-A describes concepts for understanding Boolean functions, Section II-B briefly discusses related proposals for increasing the efficiency of memory use during function manipulations.

A. Preliminaries

In ICs, signals can be symbolized by variables $x_1, \ldots, x_n$ taking logical values from $\mathbb{B} := \{0, 1\}$. In statement logic, 0 (1) $\in \mathbb{B}$ is interpreted as `false` (`true`). Thus, outputs whose values are specified by inputs can be described by mathematical mappings like *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 [21], the basis for today's computer systems, allows computations with Boolean functions as well as their manipulation and defines *algebraic structures*.

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

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

is called the *Boolean algebra of functions*.

Based on Definition 2, properties can be derived including but not limited to *commutative*, *absorption*, *resolution*, and *annulment laws* [22]. Properties come in pairs, i. e. the dual of a *Boolean Expression* (BE) – a common representation of Boolean functions – is obtained by interchanging $+$ with $\cdot$ and 0 with 1 [23].

Definition 3. Let $X_n = \{x_1, x_2, \ldots, x_n\}$ be a variable set and $\Sigma = X_n \cup \{0, 1, +, \cdot, \overline{\ }, (,)\}$ be an alphabet. The set $BE(X_n)$ over X_n is the subset of Σ^* that is defined inductively:

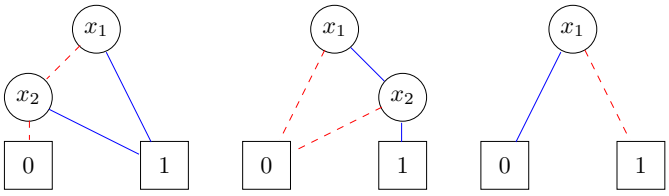
Identity elements 0 and 1, and variables are BEs.
If g and h are BEs, then the disjunction $g + h$, conjunction $g \cdot h$ (gh), and negation $\overline{g}$ are also BEs.
Nothing else is a BE.

If an IC is modular, a technique such as model checking can first compute single representations in order to subsequently combine them [12], enabled by *Shannon expansion* [24].

Definition 4. Let $f \in \mathcal{B}_n$ be a n-ary function. The partitioning f to x_i with $f_{x_i=1}(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, 1, \alpha_{i+1}, \alpha_{i+2}, \ldots, \alpha_n)$ and $f_{x_i=0}(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, 0, \alpha_{i+1}, \alpha_{i+2}, \ldots, \alpha_n)$ $\forall \alpha \in \mathbb{B}^n$ is called the *Shannon expansion*

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

If variables are successively decomposed using Definition 4 respecting a total order π and avoiding redundancies/isomorphisms by exploiting laws of Boolean algebra, a *BDD* results.



(a) $x_1 + x_2$ (b) $x_1 \cdot x_2$ (c) $\overline{x}_1$

Fig. 1: BDDs representing the Boolean basis functions

Definition 5. A *BDD* is a directed acyclic graph $G = (V, E)$ over variables $X_n := \{x_1, x_2, \ldots, x_n\}$ and a value set $\mathbb{B}$. Each node is assigned such a label where a Boolean function f is interpreted as follows:

If v is labeled with $b \in \mathbb{B}$, then the leaf represents the constant function.
If v is an inner node, it is labeled with $x_i \in X_n$, where the variable is decomposed by $x_i \cdot f_v + \overline{x}_i \cdot f_{\overline{v}} = (x_i, f_v, f_{\overline{v}})$ respecting a total order $\pi : x_1 < x_2 < \ldots < x_n$, where f_v is the high child and $f_{\overline{v}}$ is the low child referenced by the parent v. The edge set E contains all such references.

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

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

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

Applying Definition 4 successively top-down to build a BDD requires repeatedly performing an equivalence test to check whether subfunctions are already represented [2]. A more efficient way is to transform Definition 4 using Boolean algebra laws and combine nodes 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 traced back to

$$ITE(f, g, h) = f \cdot g + \overline{f} \cdot h$$

that computes *If* f *Then* g *Else* h such as $f + h = ITE(f, 1, h)$, which is compatible with Definition 4 because of

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

The resulting triple corresponds to Definition 5 and formulates the conventional divide-and-conquer Algorithm 1 [25] that constructs BDDs by a sequence of logical operations starting from single nodes. At first, terminal cases are checked in Line 1. Using a CT, Lines 2–4 check if an operand combination has already been computed. Otherwise, two cofactors are computed in Lines 5–7 decomposing according to the previously determined order of variables. Lines 8–10 check for isomorphism. In Line 11, canonicity is ensured by either finding or adding the computed triple into the UT ut.

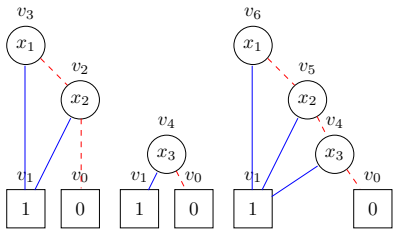
Algorithm 1: Conventional BDD manipulation operator ITE

Input: BDDs f, g, h
Output: Constructed BDD based on f, g, h

```
1  ...                                              ▷ terminal cases
2  if ct.has_entry(f, g, h) then
3  |   return ct(f, g, h)
4  end if
5  x ← top variable of f, g, h
6  t ← ITE(f_{x_i}, g_{x_i}, h_{x_i})
7  e ← ITE(f_{x̄_i}, g_{x̄_i}, h_{x̄_i})
8  if t = e then
9  |   return t
10 end if
11 r ← ut.find_or_add(x, t, e)
12 ct.insert(f, g, h, r)
13 return r
```

Node	Label	Then	Else
v_0	0	—	—
v_1	1	—	—
v_2	x_2	v_1	v_0
v_3	x_1	v_1	v_2
v_4	x_3	v_1	v_0
v_5	x_2	v_1	v_4
v_6	x_1	v_1	v_5

(a) UT



(b) f (c) h (d) $ITE(f, 1, h)$

Fig. 2: Performing ITE, where $f = x_1 + x_2$ and $h = x_3$

Depending on the current number of dead nodes or if there is not enough space to add new nodes, a GC is performed to periodically free unused memory with subsequent table expansions if necessary. Finally, Line 12 stores the result in the CT followed by the return of the constructed BDD.

Example 2. Let $\rho \in \mathcal{B}_3$ with $\rho(x_1, x_2, x_3) = x_1 + x_2 + x_3$, where $\pi : x_1 < x_2 < x_3$. Fig. 2 illustrates the UT (Fig. 2a) with BDD nodes after performing $ITE(f, 1, h)$ on the basis of $f = x_1 + x_2$ (Fig. 2b), $g = 1$, and $h = x_3$ (Fig. 2c), where Fig. 2d represents the constructed final BDD. It can be seen that during the BDD manipulation there are no more references to the nodes v_2 and v_3, i. e. they are dead.

Based on these components, Algorithm 1 can be carried out in a time almost linear to the number of BDD nodes assuming an ideal UT and CT, i. e. checking and storing nodes in constant time.

B. Related Work

In practical applications, logical operations such as Algorithm 1 are performed successively [5]. Beyond the theoretical point of view w. r. t. complexity described in the last section, this can dramatically increase the UT size which leads to a failure (or a significant slowdown) of the computation caused by a memory overflow.

Although optimizations for choosing a proper variable order have been developed in recent years to address this issue [26], the reduction of intermediate computations has not been investigated to this extent: While algorithms such as [15] create BDD nodes only when such are represented in the final result, there exist approaches like [16] that get final BDDs top-down and bottom-up. Below is a brief explanation of these methods.

The so-called *Multi-way* method [15] exploits implicit don't cares in a BE including a cube cofactorization by recursively applying Definition 4 to address wasteful intermediate computations. For example, when computing a BE like $fg + h$, each of f and g can be minimized using h as a don't care set before the product is formed. That is, for any subexpression where h is 1, the final result will also be 1 on that subexpression regardless of the logical value of f or g. Thus, the creation of Inodes is prevented and only the final BDD is constructed.

The second method, called *XTop* [16], first performs a topological analysis top-down to find good decomposition points for the IC. To this end, a cut-set is computed by reducing the number of compositions and dependent variables w. r. t. the outputs, where a cut-set is a set of gates such that any path from an IC input to an IC output has to cross through one of the gates in the IC. Based on selected decomposition points, single BDDs are then constructed bottom-up using Algorithm 1, which are finally composed.

Although the aforementioned methods have proven successful for various problems, such as formal design verification and graph-theoretic problems, they come with some drawbacks: While with *Multi-way* from [15] the explosion in the number of BE operations exists, *XTop* from [16] has problems with essential algorithms like dynamic variable ordering. Therefore, the main goal of this work is to overcome these limitations.

III. Reducing the Number of Intermediate Nodes

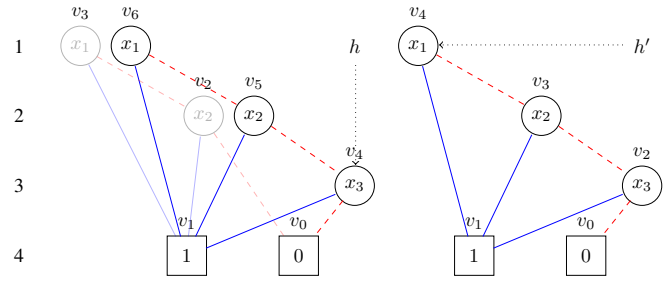
In this section, we describe the core development from our investigation to the approach of reducing the number of *Intermediate Nodes* (Inodes). First, the emergence of Inodes and their number are investigated in Section III-A. Second, in Section III-B, we present our approach to allow systematic reduction of the number of Inodes.

A. Investigation of the number of emerged Inodes

In BDD packages, complicated BEs in the form of BDDs are constructed by logically combining single BDD nodes using Algorithm 1. If combining is performed conventionally, this can lead to unnecessary memory and runtime overhead due to intermediate results that are not in the final BDD.

Considering Example 2, for the construction of $x_1 + x_2 + x_3$ first $f = x_1 + x_2$ is built, which is afterwards combined with $h = x_3$ via logical disjunction. However, the problem with this combination is that two Inodes (v_2, v_3) emerge since they are not used in the final BDD (Fig. 2d).

Remark. In the case of dead nodes, it is not clear during BDD manipulation whether they are finally intermediate, since they can be reactivated and thus also be in the final result [11]. Every Inode is dead, but not every dead node is intermediate.



(a) $ITE(f, 1, h)$ (b) $ITE(f', 1, h')$

Fig. 3: ITE constructions for $x_1+x_2+x_3$, where $f = x_1+x_2$, $h = x_3$, $f' = x_3 + x_2$, and $h' = x_1$

Via an ITE walkthrough, it can be observed why these Inodes emerge and how they can be prevented, which is illustrated in Fig. 3: Respecting the order $\pi : x_1 < x_2 < x_3$, it is shown in Fig. 3a that h must be "transported" to the intended level 3 in the BDD, causing references to change, and v_2 and v_3 to become intermediate. If, taking commutativity into account as shown in Fig. 3b, $f' = x_3 + x_2$ is constructed first and then combined with $h' = x_1$, no Inode results since there is no recursive descent and h' can be "docked" directly. This observation applies analogously to logical conjunction $ITE(f, h, 0)$ because of the principle of duality mentioned in Section II-A, i. e. only the edge redirections need to be swapped. For negation it is sufficient to swap only the 1-edges and 0-edges.

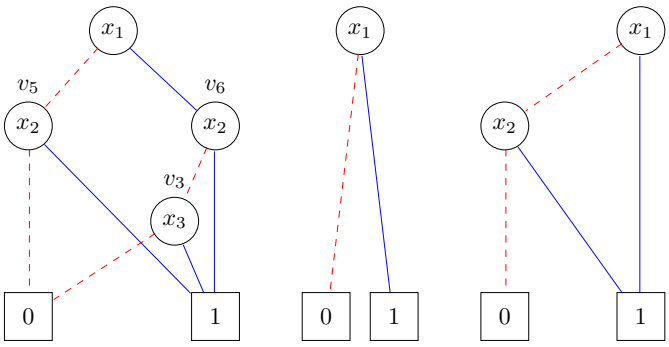
Remark. Binary operators, by themselves, do not perform intermediate computations nor create unnecessary nodes. Thus, it makes no difference for this investigation whether ITE or related algorithms such as $APPLY$ [2] construct BDDs.

Generally, the order in which the operands are processed can drastically affect the memory and thus the runtime since the ITE recursion terminates in a specific branch depending on the combination for an operand: for example, for conjunction the value 0 applies, for disjunction if an operand becomes 1.

Another weakness of a conventional BDD manipulation becomes apparent by possible simplifications using the Boolean algebra laws (Section II-A).

Example 3. Let $\rho \in \mathcal{B}_3$, $\rho(x_1, x_2, x_3) = x_1x_3 + x_2 + x_1$, and $\pi : x_1 < x_2 < x_3$. Anticipating the absorption law, ρ can be simplified to $x_2 + x_1$. Fig. 4 shows performing $ITE(f, 1, h)$ based on $f = x_1x_3 + x_2$ (Fig. 4a) and $h = x_1$ (Fig. 4b). During the recursion steps, it is detected that the node v_3 already contained in the UT can be absorbed making v_5 and v_6 isomorphic and thus mergeable as illustrated in Fig. 4c.

The findings from Example 3 can easily be transferred to other laws such as the resolution laws. An extreme case occurs when, e. g., a BE like $fg+h$ is constructed where $h = 1$. Then the construction of fg is wasted since the final BDD is only the 1-leaf. This obviously becomes worse as the BE grows longer. Analogous to the operand order, redundancies can also have negative effects on memory and runtime.



(a) f (b) h (c) $ITE(f, 1, h)$

Fig. 4: Absorption detection during $ITE(f, 1, h)$ manipulation, where $f = x_1x_3 + x_2$ and $h = x_1$

Algorithm 2: Preprocessing approach *Sortify* to reduce the number of Inodes during BDD manipulation

Input: BE f
Output: Sorted simplified BE based on f
1 *#pragma omp parallel*
2 *#pragma omp master*
3 $sort(f_{begin}, f_{end})$
4 $simplify(f)$

In summary, the operand order should already be observed before the BDD manipulation and redundancies should be discovered as early as possible in order to reduce the number of Inodes.

B. Proposed Approach To Reduce the Number of Inodes

BDDs are constructed on the basis of combining single nodes during BDD manipulation. Due to our observations from the last section, (unnecessary) Inodes emerge during BDD manipulation for two main reasons: 1) poorly selected operand order and 2) redundancies.

As explained in Section II-B, intermediate computations are completely prevented in [15]. However, there is usually a high rebirth rate using well-known techniques such as model checking [11], i. e. dead nodes that may be intermediate are often reactivated before a GC which can prevent deep recursive descents due to caching. In general, it is difficult to predict the "value" of a created node [19]. Therefore, Inodes are permitted up to a certain threshold value in [16]. However, due to the direct integration into the BDD manipulation there are problems with the dynamic variable ordering.

We propose a parallel preprocessing approach called *Sortify*, using w. l. o. g. OpenMP [27], as a heuristic to reduce the number of Inodes during BDD manipulation as shown in Algorithm 2. To this end, we are oriented to Quicksort [28], one of the fastest sorting algorithms. To address issue 1), a BE is sorted using a predefined sort key invoked by a master thread of a parallel region (Lines 1–3). To address issue 2), the sorted BE is searched neighbor by neighbor to simplify (Line 4).

Algorithm 3: Sorting method *sort* for Boolean subexpressions

Input: BE iterators f_{begin}, f_{end}
Output: Sorted BE f

```
1  ...                                                          ▷ terminal cases
2  p ← median(f_begin, f_begin + (f_end − f_begin)/2, f_end)
3  f_left ← f_begin
4  f_right ← f_end
5  i ← f_begin + 1
6  while i ≤ f_right do
7      if k_π(i, p) then
8          swap(f_left, i)
9          ++f_left
10         ++i
11     else if k_π(p, i) then
12         swap(f_right, i)
13         −−f_right
14     else ++i
15     end if
16 end while
17 if δ(f_begin, f_end) ≥ c then
18     #pragma omp taskgroup
19     {
20     #pragma omp task
21     if δ(f_begin, f_left) > 0 then
22         sort(f_begin, f_left − 1)
23     end if
24     #pragma omp task
25     if δ(f_right, f_end) > 0 then
26         sort(f_right + 1, f_end)
27     end if
28     }
29 else                                                          ▷ serial sorting
30 end if
```

The main idea (Algorithm 3) is to divide a BE into three subsequent partitions consisting of subexpressions that are less, equal to, or greater than a preselected subexpression p of the entire BE picked by the median of three based on the first, middle, and last subexpression (Lines 1–2). Primarily, there is now a single pass through each subexpression, from left to right (Lines 3–6). Each subexpression is compared to p, with the sort key k_π following the variable order π. There are three main cases that are handled: If a subexpression is less than p, this subexpression is swapped to the left partition (Lines 7–10), otherwise if a subexpression is greater than p, it is swapped to the right partition (Lines 11–13), else nothing is swapped (Lines 14–16). Afterwards, the left and right partitions are recursively sorted in the same way. Since the sorting of the left and right partitions is independent, it can be parallelized by two concurrent tasks. To reduce the number of parallel recursive tasks that are scheduled, the cut-off c (100,000 according to [27]) is introduced. Before concurrency it is checked if the current number of subexpressions δ to be sorted reaches c (Line 17). If this is the case, then the recursive descent is performed in parallel (Lines 18–28), otherwise serial (Lines 29–30).

To simplify the BE, its order is now exploited and, similar to finding prime implicants of the Quine McCluskey method [29], the subexpressions are compared neighbor by neighbor in one "round" to see if Boolean algebra laws can be applied.

Compared to Quicksort, Sortify allows to reduce the best-case complexity from linearithmic $\mathcal{O}(n \log n)$ to linear $\mathcal{O}(n)$, where n is the number of subexpressions. This is achieved by counteracting unbalanced partitions via the selection procedure and avoiding unnecessary recursive calls using three partitions. Since there is no logic minimization, but the subexpressions are only compared neighbor by neighbor after sorting, this effort is negligible. In addition, Sortify is parallelized and cache-coherent greatly affecting the CPU's cache pipeline.

IV. Experimental Results

This section summarizes the experiments conducted in order to empirically analyze our approach. While Section IV-A describes the setup used for the evaluations. Section IV-B presents the impact of our approach compared to related work.

A. Experimental Setup

Our preprocessing approach was implemented in C++20 as w. l. o. g. EDDY [30] was used as BDD package for performance evaluation. To allow a fair comparison, the methods discussed in Section II-B were directly integrated into EDDY. For representative purposes, IWLS-93 benchmark instances were taken from [31]. The initial UT (CT) size was set to 2^{20} (2^{18}) due to the complexity of the instances. Using these instances, peak node usage N – the greatest number of nodes in use at any point during the process lifetime – was measured during the BDD manipulation and effects on CPU time T (in sec) were recorded, both reported as node ratio NR and time ratio TR. The used variable order follows the order of appearance in the respective file. All evaluations were carried out on a Fedora 28 machine with an Intel Xeon E3-1270 v3 CPU with 3.5 GHz and 32 GB of main memory. For each instance, 10 runs were performed and the average was calculated. The *Time Out* (TO) was set to 10 min, whereas the *Memory Out* (MO) was configured to a node limit of 1 M.

B. Performance Evaluation

The experimental results can be seen in Table I and confirm that our approach meets the objectives of this work. Sortify is able to significantly reduce the number of Inodes for all considered benchmark instances thereby accelerating BDD manipulation due to a lower number of recursive descents, GC calls, UT/CT expansions, and hash collisions. As a preprocessing method for ITE, it results in peak node usage being reduced by about 29 % compared to single usage (Conventional). This accelerates the runtime by about 23 %. While minimally more nodes are needed compared to Multi-way, Sortify is about 19 times faster due to better cache performance which is a successful compromise. In addition, Sortify is stable in solving these instances: While XTop, e. g., has a higher peak node usage than Conventional to construct the BDD for c5315 due to poorly selected decomposition points, Sortify always uses a lower number of nodes at any point during the process lifetime.

TABLE I: Experimental comparison of Sortify and related work in terms of peak node usage and CPU time

Instance Name	Conventional		Multi-way		XTop		Sortify		Conv./Sortify		Multi-way/Sortify		XTop/Sortify	
	N	T	N	T	N	T	N	T	NR	TR	NR	TR	NR	TR
des	21,449	0.52	**13,455**	5.88	21,316	0.52	15,209	**0.45**	1.41	1.16	0.88	13.07	1.40	1.16
rot	14,038	0.43	**9,781**	4.60	13,805	0.35	10,228	**0.16**	1.37	2.69	0.96	28.75	1.35	2.19
c880	15,452	0.47	**10,757**	2.42	13,674	0.28	11,497	**0.19**	1.34	2.47	0.94	12.74	1.19	1.47
c1355	46,238	0.65	**32,768**	3.98	40,233	0.52	34,105	**0.37**	1.36	1.76	0.96	10.76	1.18	1.41
c1908	24,854	0.54	**18,201**	2.93	24,032	0.38	22,193	**0.24**	1.12	2.25	0.82	12.21	1.08	1.58
c2670	322,988	4.47	**167,241**	86.87	259,734	4.13	184,179	**3.61**	1.75	1.24	0.91	24.06	1.41	1.14
c3540	405,544	5.39	**292,834**	97.84	393,867	5.20	318,223	**4.52**	1.27	1.19	0.92	21.65	1.24	1.15
c5315	39,983	0.64	**30,992**	13.96	40,577	0.76	33,109	**0.55**	1.21	1.16	0.94	25.38	1.23	1.38
c6288-12	256,878	2.28	—	TO	208,887	2.13	**176,092**	**1.83**	1.46	1.25	—	—	1.19	1.16
c6288-13	732,667	9.27	—	TO	611,420	8.81	**428,977**	**7.02**	1.71	1.32	—	—	1.43	1.25
c6288-14	MO	—	—	TO	MO	—	**926,563**	**31.97**	—	—	—	—	—	—
Total (1–8)	890,546	13.11	576,029	218.48	807,238	12.14	628,743	10.09	1.40	1.65	0.92	18.58	1.27	1.39

N Peak node usage T CPU time in sec NR Node ratio TR Time ratio

V. Conclusion

This paper focused on investigating the number of Inode emergences using Boolean functions in order to reduce them and accelerate BDD manipulation. By observing the variable order and detecting redundancy, our developed approach can significantly reduce the number of Inodes and is on average about 20 % faster compared to the single use of ITE and related work as demonstrated by experiments.

In addition to the consideration of further benchmark instances and study of various parameters, future research will be directed towards integrating our approach directly into the BDD manipulation, e. g., to detect redundancies structurally in order to try to further improve the manipulation.

References

[1] G. Moore, "Cramming more components onto integrated circuits," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

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

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

[4] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[5] S. Chaki and A. Gurfinkel, *BDD-Based Symbolic Model Checking*. Springer, 2018.

[6] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[7] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[8] K. McMillan, *Symbolic Model Checking*. Springer, 1993.

[9] G. Janssen, "A consumer report on BDD packages," in *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design*. IEEE Computer Society, 2003, pp. 217–222.

[10] R. Drechsler and D. Sieling, "Binary decision diagrams in theory and practice," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, pp. 112–136, 2001.

[11] B. Yang, R. Bryant, D. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. Ranjan, and F. Somenzi, "A performance study of BDD-based model checking," in *Proceedings of the 2th International Conference on Formal Methods in Computer-Aided Design*. Springer, 1998, pp. 255–289.

[12] T. van Dijk, E. Hahn, D. Jansen, Y. Li, T. Neele, M. Stoelinga, A. Turrini, and L. Zhang, "A comparative study of BDD packages for probabilistic symbolic model checking," in *Proceedings of the First International Symposium on Dependable Software Engineering*, 2015, pp. 35–51.

[13] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *24th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE Computer Society, 2021, pp. 99–104.

[14] R. Drechsler and A. Mahzoon, "Polynomial formal verification: Ensuring correctness under resource constraints," in *Proceedings of the 41st IEEE/ACM International Conference on CAD*. ACM, 2022.

[15] T. Shiple, R. Brayton, and A. Sangiovanni-Vincentelli, "Computing Boolean expressions with OBDDs," UC Berkeley, Tech. Rep., 1993.

[16] J. Jain, A. Narayan, C. Coelho, S. Khatri, A. Sangiovanni-Vincentelli, R. Brayton, and M. Fujita, "Combining top-down and bottom-up approaches for ROBDD," UC Berkeley, Tech. Rep., 1996.

[17] A. Hett, R. Drechsler, and B. Becker, "Fast and efficient construction of BDDs by reordering based synthesis," in *Proceedings of the European Conference on Design and Test*. IEEE, 1997, pp. 168–175.

[18] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of the 34th Annual Design Automation Conference*. ACM, 1997, pp. 263–268.

[19] S. Minato, "Streaming BDD manipulation," *IEEE Transactions on Computers*, vol. 51, no. 5, pp. 474–485, 2002.

[20] S. Sølvsten, J. Pol, A. Jakobsen, and M. Thomasen, "Efficient binary decision diagram manipulation in external memory," *CoRR*, 2021. [Online]. Available: https://arxiv.org/abs/2104.12101

[21] G. Boole, "The calculus of logic," *The Cambridge and Dublin Mathematical Journal*, vol. 3, 1848.

[22] E. Huntington, "Boolean algebra. a correction," *Transactions of the American Mathematical Society*, 1933.

[23] T. Jenkyns and B. Stephenson, *Boolean Expressions, Logic, and Proof*. Springer, 2018.

[24] C. Shannon, "The synthesis of two-terminal switching circuits," *The Bell System Technical Journal*, vol. 28, no. 1, pp. 59–98, 1949.

[25] G. Janssen, "Design of a pointerless BDD package," in *International Workshop on Logic and Synthesis*, 2001, pp. 310–315.

[26] C. Jiang, J. Babar, G. Ciardo, A. Miner, and B. Smith, "Variable reordering in binary decision diagrams," *International Workshop on Logic and Synthesis*, 2018.

[27] T. Mattson, "Introduction to OpenMP," in *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM, 2006.

[28] G. Rahul, P. Sandeep, and Y. Latha, *Quicksort Algorithm—An Empirical Study*. Springer, 2020.

[29] A. Majumder, B. Chowdhury, A. Mondai, and K. Jain, "Investigation on Quine McCluskey method: A decimal manipulation based novel approach for the minimization of Boolean function," in *International Conference on Electronic Design, Computer Networks & Automated Verification (EDCAV)*, 2015, pp. 18–22.

[30] R. Krauss, M. Goli, and R. Drechsler, "EDDY: A multi-core BDD package with dynamic memory management and reduced fragmentation," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. ACM, 2023, pp. 423–428.

[31] P. Fišer and J. Schmidt, "A comprehensive set of logic synthesis and optimization examples," in *Proceedings of the 12th International Workshop on Boolean Problems*, 2016, pp. 151–158.