Dynamic variable reordering for Binary Decision Diagrams

Master Thesis by

Andrej Pištek

Graduation committee:
dr. T. van Dijk
dr.ir. M. van Keulen
dr.ing E.M. Hahn

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Formal Methods and Tools (FMT)
Enschede, The Netherlands

August 15, 2023

Abstract

Binary Decision Diagrams (BDDs) are data structures that represent and manipulate Boolean functions efficiently. Variable ordering in BDDs determines the sequence in which variables are assigned, impacting their compactness and performance. In this thesis, we have researched, implemented and evaluated the dynamic variable reordering in the multi-core BDD package Sylvan. We cover how Rudell's sifting algorithm with parallel variable swap enables dynamic reordering in Sylvan. Also, we show why hash maps with chaining, reference-based garbage collection, and roaring bitmaps emerge as optimal strategies for efficient variable swaps, with the potential for future enhancements outlined. Optimal values for tuning parameters to improve the reordering performance are identified, with scaling effects observed in parallelisation. Finally, we evaluate the results with a series of safety game benchmarks, among which Sylvan reordering outperforms the state of the art CUDD BDD package for sufficiently large samples.

Contents

List of Abbreviations

ADD	**A**lgebraic **D**ecision **D**iagram
BDD	**B**inary **D**ecision **D**iagram
CUDD	**C**olorado **U**niversity **D**ecision **D**iagram
MTDD	**M**ulti-**T**erminal **D**ecision **D**iagram
OBDD	**O**rdered **B**inary **D**ecision **D**iagram
ROBDD	**R**educed **O**rdered **B**inary **D**ecision **D**iagram

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In the current fast-paced world, many crucial areas and infrastructures such as traffic systems, railways, healthcare and others rely on increasingly more complex automated systems. This demands safety and reliability of such systems more than ever before. Among others, model checking is a technique that helps to ensure the safety and reliability of such complex automated systems by formally verifying the properties of the system's behaviour. It is done by formalizing mathematical models of the system to check whether certain properties, such as safety and liveness, hold true. This is done via exploring all possible states and transitions of the system, which is known as the state space [36, 37]. Besides verifying desired properties of a system, it is also important to ensure that the logic circuits of the build system are designed correctly. It can be done by involving combinational circuit analysis to examine and evaluate logic circuits composed of interconnected gates. However, when the complexity of a system grows, the state space capturing the system properties grows as well. Therefore, it is vital to store and manipulate the states efficiently.

To help store and manipulate the states of a system or logic of a circuit, a fundamental data structure called Binary Decision Diagram (BDD) is commonly used. It is a data structure employed to handle large-scale state spaces and operations on Boolean functions in a scalable and efficient way. This makes BDDs versatile data structures for tackling complex computational problems. The variable ordering in a BDD determines the order in which the variables are considered when traversing a BDD. The choice of variable ordering can greatly affect the size of a BDD, with certain orderings resulting in much smaller BDDs than others. As a consequence, variable ordering can greatly affect the performance and scalability of BDD operations, which makes it an important topic to study [7].

To find an optimal static variable ordering for BDDs is NP-Hard [4]. An alternative approach to finding a good variable order is dynamic variable reordering, which reorders the variables on an existing BDD as the operations proceed. A commonly used approach to dynamic variable reordering is the sifting algorithm proposed by Rudell [28]. It performs a series of swap operations that swaps x_i and x_{i+1} in the variable order. Sifting variables has been shown to largely improve memory peek performance

which is the bottleneck of BDDs [24]. Therefore, in this thesis, we will research, implement and evaluate the sifting algorithm in the multi-core BDD package Sylvan. Moreover, the fine-tuned algorithm will be compared with the state of art Colorado University Decision Diagram (CUDD) package. CUDD package is the default choice for most of the symbolic model checkers [10]. It has well-tuned heuristics for controlling memory allocation and sifting [29]. Hence, the central research question (CRQ) is formulated as follows:

How to maximize the performance of the dynamic variable reordering for binary decision diagrams in the Sylvan BDD package?

The central research question covers several aspects namely, how the sifting algorithm can be implemented in Sylvan, how can different tuning parameters improve the sifting algorithm, and lastly, how the comparison between Sylvan with the sifting algorithm and other state of art BDD packages such as CUDD can be made fair. Therefore, the sub-research questions (RQs) are derived from the central research question to split the individual aspects into the following separate narrowed RQs:

RQ1: *How to implement dynamic variable reordering using the sifting algorithm in Sylvan?*

RQ2: *How to tune different parameters to maximize the performance of the dynamic variable reordering in Sylvan?*

RQ3: *How can a fair comparison of the dynamic variable reordering performance be made between Sylvan and CUDD?*

RQ4: *How does the Sylvan hash map with chaining collision avoidance affect the performance compared to linear probing collision avoidance w.r.t dynamic variable reordering?*

Before this thesis, the initial prototype implementation of dynamic variable reordering in Sylvan was provided by the thesis supervisor, dr. T. van Dijk. The implementation contained a variable swap, a variation of the Sylvan hash map to support single-item deletion. The remainder of this report is structured as follows. Chapter 2 introduces the fundamentals of Binary Decision Diagrams, the sifting algorithm, Lace, Sylvan and CUDD BDD packages, and BDD use cases. Chapter 3 provides an in-depth review of the related literature, discussing various existing variable reordering techniques. Chapter 4 presents the methodology, detailing the approach to answering the RQs and evaluation of the results. Chapter 5 compares hash map collision avoidance techniques w.r.t dynamic variable reordering. Chapter 6 details the implementation of a single adjacent variable swap, a core operation of the sifting algorithm detailed in Chapter 7. Chapter 8 presents the experimental evaluation, including the selection of benchmark models, performance metrics, and comparative analysis of the results. Chapter 9 provides a guide to help Sylvan users get the most out of the dynamic variable reordering. Chapter 10 discusses the research findings, highlighting the strengths, limitations, and potential future work. Finally, Chapter 11 concludes the thesis, summarizing its contributions.

Chapter 2

Preliminaries

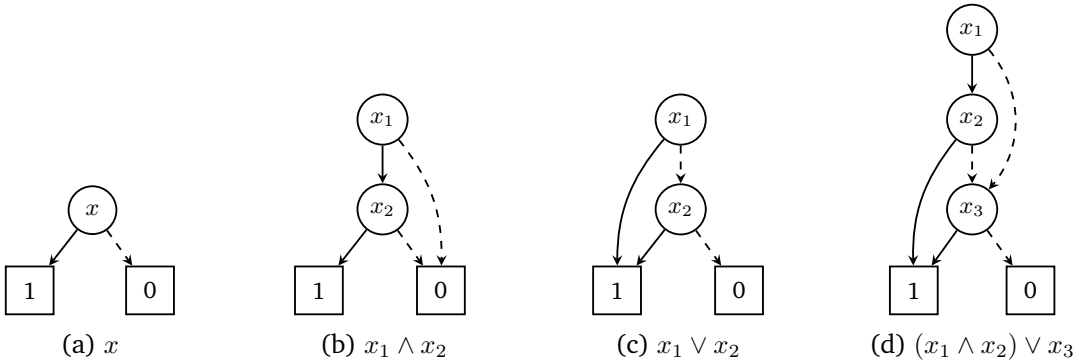
This chapter explains the basics of Binary Decision Diagrams, Sylvan and CUDD packages as well as the safety game. Firstly, BDDs are formally introduced together with BDD reduction and variable ordering. Then, the sifting algorithm proposed by Rudell is explained. Next, work stealing is introduced together with the framework Lace. Furthermore, two BDD packages are introduced, namely Sylvan and Colorado University Decision Diagram (CUDD). Lastly, use cases of BDDs and the safety game are introduced.

2.1 Binary Decision Diagrams

In the field of computer science, a Binary Decision Diagrams (BDD) is a fundamental data structure representing and manipulating Boolean functions denoted as $f : \mathbf{B^n} \rightarrow \mathbf{B}$ with n inputs where $n \in \mathbb{N}$. BDDs have demonstrated their efficiency as a data structure in various logic synthesis algorithms, including logic optimization or verification of both combinational circuits [1][31][28]. The concept of BDDs was initially introduced by Akers [1] and further developed by Bryant [6]. BDD is a directed acyclic graph that represents a Boolean function through the use of Shannon decomposition [30]. The definitions of BDD and Shannon decomposition are provided below.

Definition 2.1.1 (Binary Decision Diagram)**.** *Let $G = (V, E)$ be defined as a single-rooted, directed, and acyclic graph. Also, let G contain two types of vertices namely, terminal and non-terminal. A terminal vertex $v \subseteq V$ has an attribute $value(v) \in \{0, 1\}$. A non-terminal vertex $v \subseteq V$ has an attribute $level(v) \in \{1, ..., n\}$ and two children vertices $low(v)$, $high(v) \in V$ to which low and high edges of v labelled as 0 and 1 are pointing respectively. Then, G is called a binary decision diagram.*

In this thesis, we use circles to denote non-terminal vertices containing variables with index i and squares to denote terminal vertices containing values 1 or 0. The 1-edge is drawn using an arrow with a solid line and the 0-edge is drawn using a arrow with a dashed line. In Figure 2.1 four Boolean formulas are graphically depicted as BDDs. The Boolean formulas contain boolean algebra operations such as conjunction known as AND ($x \wedge y$), and disjunction known as OR ($x \vee y$).



(a) x (b) $x_1 \wedge x_2$ (c) $x_1 \vee x_2$ (d) $(x_1 \wedge x_2) \vee x_3$

Figure 2.1: Binary Decision Diagrams depicting four Boolean formulas.

Definition 2.1.2 (Shannon cofactors)**.** *Let F be a Boolean function on $X = \{x_i, ..., x_n\}$. The positive cofactor $F_{x_i=1}$ and the negative cofactor $F_{x_i=0}$ are defined as follows:*

$$F(x_1, ..., x_i, ..., x_n)_{x_i=1} \equiv F(x_1, ..., 1, ..., x_n)$$
$$F(x_1, ..., x_i, ..., x_n)_{x_i=0} \equiv F(x_1, ..., 0, ..., x_n)$$

Shannon cofactors are a fundamental concept in Boolean function manipulation. Given a Boolean function represented as a BDD, Shannon cofactors are obtained by splitting the BDD based on the value of a selected variable. The positive cofactor represents the part of the BDD where x_i is assigned the value 1. It is obtained by removing all nodes in the BDD where x_i is assigned 0. The negative cofactor represents the part of the BDD where x_i is assigned the value 0. It is obtained by removing all nodes in the BDD where x_i is assigned 1. See Figure 2.2 for a graphical representation provided by the courtesy of van Dijk [8].

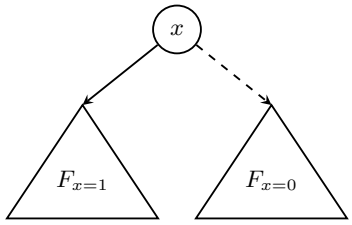


Figure 2.2: Shannon cofactors as sub-trees

Theorem 2.1.1 (Shannon decomposition)**.** *Let F be a Boolean function on $X = \{x_i, ..., x_n\}$. The following identity is Shannon expansion or decomposition of F with respect to x_i:*

$$F \equiv (x_i \wedge F_{x_i=1}) \vee (\overline{x_i} \wedge F_{x_i=0})$$

In 1938, Shannon proposed in [30] the Shannon decomposition or also called expansion. Any Boolean function can be represented by a BDD by applying the Shannon decomposition recursively. Example 2.1.1 shows the decomposition of a single variable.

Example 2.1.1. *Let $f(x, y, z)$ be a function with three inputs x, y, z. By applying the Shannon decomposition to variable x, we get $f(x, y, z) \equiv (x \vee f(1, y, z)) \vee (\overline{x} \wedge f(0, y, z))$. After the decomposition, x is not required as an input for function f anymore. By applying the decomposition recursively, variables y and z can be removed as well leaving us with only ones and zeros. In more general terms, if f takes n inputs after a single decomposition of one of its input variables, we need $n - 1$ inputs for f.*

An Ordered Binary Decision Diagram (OBDD) is a BDD where variables are ordered, and every path from the root to the leaf visits the variables in ascending order. The bottleneck of BDDs is the need to order the variables [28]. The choice of variable ordering can greatly affect the size of the BDD, with certain orderings resulting in much smaller BDDs than others. This can greatly affect the performance and scalability of such BDD.

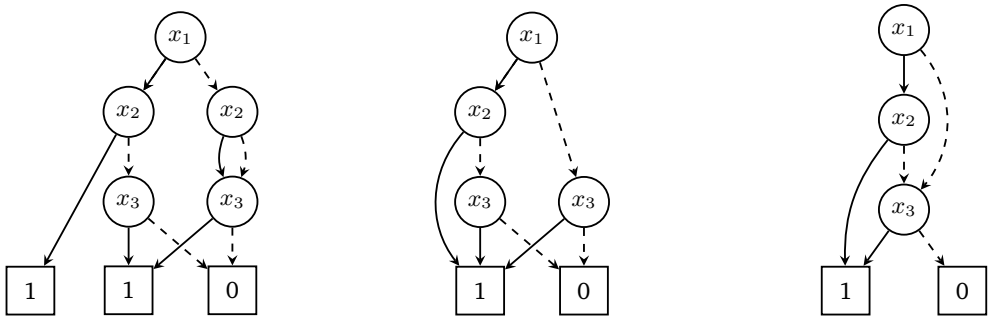
The variables can be ordered either statically or dynamically. Static variable ordering is typically used before the BDD is constructed and the optimal ordering is determined based on a use case. The dynamic variable reordering is used on an existing BDD as the operations proceed. In this thesis, we will be concerned with the dynamic variable reordering. Consider Figure 2.3a, the variable order in this BDD is $x_1 \prec x_2 \prec x_3$ and the Boolean function is $F = (x_1 \wedge x_3) \vee x_2$. By reordering the variables $x_1 \prec x_3 \prec x_2$ as shown in Figure 2.3b, the size of the BDD is reduced while preserving the same Boolean function $F = (x_1 \wedge x_3) \vee x_2$. Note that the Boolean formula in Figure 2.3b is different from the formula in Figure 2.1d, although the graphical representation of the BDD resembles the BDD in Figure 2.3b.

(a) BDD with $F = (x_1 \wedge x_3) \vee x_2$ (b) BDD with $F = (x_1 \wedge x_3) \vee x_2$

Figure 2.3: Binary decision diagram variable ordering.

A reduced-ordered binary decision diagram (ROBDD) usually simply called BDD, is a BDD that contains no redundant nodes and it does not contain duplicate sub-graphs. A redundant node is a node that has two identical child nodes. Any BDD can be reduced by following the reduction rules:

1. Merge equivalent nodes
2. Eliminate isomorphic sub-graphs by sharing sub-graphs



(a) Node redundancy (b) Duplicate sub-graphs (c) ROBDD

Figure 2.4: Reduction process of a BDD.

Consider Figure 2.4a, it is a BDD but not ROBDD. By applying the reduction rules we delete the redundant terminal node with value 1 and the redundant variable x_2. Then, in Figure 2.4b both variables x_3 point to two isomorphic sub-graphs which can be merged by applying the reduction rule 2. Finally, in Figure 2.4c we have reached ROBDD.

2.2 Sifting algorithm

In 1993, a paradigm for maintaining variable orders in a BDD called the sifting algorithm was proposed in [28] by Rudell. It is a dynamic variable reordering algorithm, meaning the reordering is performed on an existing BDD as the operations proceed. The sifting algorithm uses adjacent variable swap which affects only the BDD variables at the two levels x_i and x_{i+1}; all other variables remain unchanged. This is possible due to Theorem 2.2.1 [33].

Theorem 2.2.1. *Let F be a BDD over X_n and x be a variable at level i for which we assume the natural ordering π with $\pi(i) = x_i (1 \leq i \leq n)$. Then, moving x down the order (from i to $i+1$) has no effect on nodes at levels $< i$ (the part of the BDD above x is not affected by the reordering). Similarly, moving x up the variable order (from i to $i-1$) has no effect on nodes at levels $> i$ (the part of the BDD below x is not affected by the reordering).*

Figure 2.5 depicts Theorem 2.2.1 graphically. Level $i-1$ is closer to the root of the BDD, and level $i+1$ is closer to the terminal nodes of the BDD. Theorem 2.2.1 implies that no information is required about the upper part of the BDD (an upper grey area) when sifting down and the lower part (a lower grey area) when sifting up. Sifting up or down refers to a series of adjacent variable swaps using which any given variable can be moved in ordering π. After introducing the concept of sifting, we now introduce an adjacent variable swap, a core operation on which the sifting algorithm relies.

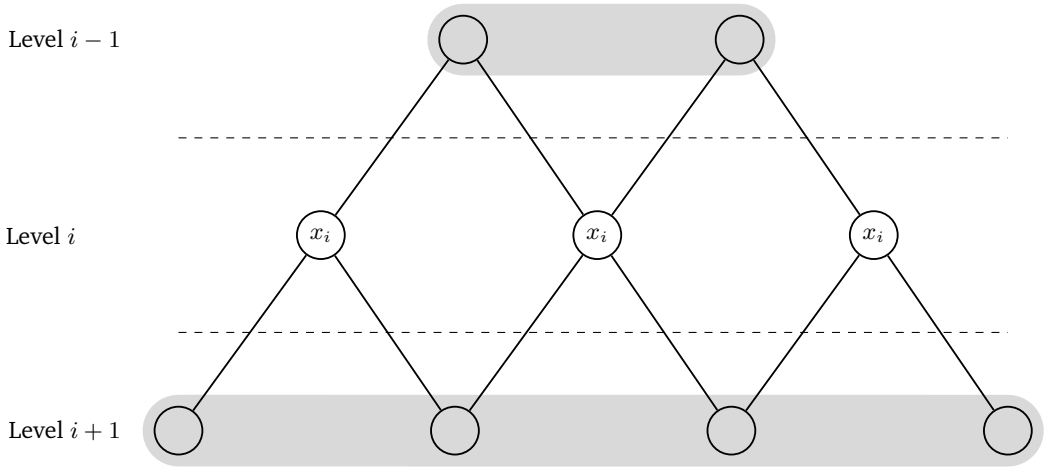


Figure 2.5: A BDD with variable x at level i

Due to Theorem 2.2.1, performing an adjacent variable swap between level i and $i+1$ is considered to be a local operation on any BDD. However, several implications arise from swapping a variable in a BDD namely, new nodes may be created and some nodes become dead. A dead node is a node which has no internal references (coming from other nodes in the forest) and no external references (coming from a user). Moreover, since the variable order is no longer preserved, to refer to a particular variable it is necessary to maintain a mapping from the original variable ordering to the ordering resulting from the swap. Example 2.2.1 details adjacent variable swaps between variables x and y.

Example 2.2.1. *Let $F = (x, F_1, F_0)$ with a BDD variable x, cofactors F_1 and F_0 and let F_{11} and F_{10} be the two cofactors of F_1 with respect to x and F_{01} and F_{00} be the two cofactors of F_0 with respect to y. Then, using the formula expansion we get $F = (x, (y, F_{11}, F_{10}), (y, F_{01}, F_{00}))$. By swapping variable x with y, F becomes $(y, (x, F_{11}, F_{01}), (x, F_{10}, F_{00}))$. As a result of the swap, cofactors F_{10} and F_{01} are swapped as well. The resulting formula expansion shows that the order of variables is swapped and the function of F is preserved. The swap is graphically depicted as follows:*

Figure 2.6: Variable swap between x and y

Consider Example 2.2.1, after the swap, cofactors F_1 and F_0 of x can be freed if they become dead since they are no longer referenced by x. However, cofactors F_{11}, F_{01}, F_{10},

F_{00} are referenced after the swap by the newly created variables x and can not be freed. In more general terms, when a variable is swapped at level i, variables at level $i+1$ can be deleted if their only reference was from level i. This observation can be used in the incremental garbage collection during the variable swap [28]. Although incremental garbage collection maybe is employed, a reference count is required for each node in the forest. Depending on the situation, a variable swap might result in adding more nodes, removing nodes or staying at the same number. Particular situation depends on cofactors F_{11}, F_{01}, F_{10}, F_{00}. In Example 2.2.2, variable swap between x and y results in more nodes.

Example 2.2.2. *Let $F = (x, F_1, F_0)$ with a BDD variable x, cofactors F_1 and F_0, and let F_{11} and F_{10} be the two cofactors of F_1 with respect to y. Then, using the formula expansion we get $F = (x, (y, F_{11}, F_{10}), F_0)$. By swapping variable x with y, F becomes $(y, (x, F_{11}, F_{01}), (x, F_{10}, F_{00}))$. The resulting formula expansion shows that a new node was created while preserving function F. Moreover, $F_{01}$and F_{10} now point to the same function. The swap is graphically depicted as follows:*

Figure 2.7: Variable swap between x_1 and x_2 with left children resulting in more nodes

Example 2.2.3. *Let $F = (x, F_1, F_0)$ with a BDD variable x, cofactors F_1 and F_0, and let F_{11} and F_{10} be the two cofactors of F_1 with respect to y. Then, using the formula expansion we get $F = (x, (y, F_{11}, F_{10}), F_0)$. By swapping variable x with y, F becomes $(y, F_1, (x, F_{10}, F_{00}))$. The resulting formula expansion shows that a new node was created while preserving function F. Moreover, $F_1$and F_{00} now point to the same function. The swap is graphically depicted as follows:*

Figure 2.8: Variable swap between x_1 and x_2 resulting in the same number of nodes

Consider Example 2.2.2, after the swap, an additional node for variable x is created. The resulting BDD depicted in Figure 2.7b contains F_{01} and F_{00} which both point to the same children node. The case when an adjacent variable swap reduces the number of nodes in a BDD is in fact the exact opposite of Example 2.2.2. The reverse swap can be obtained as follows; first, consider the BDD depicted in Figure 2.7b and swap variable x with y, then we get a BDD with where cofactor F_0 has both its cofactors F_{01} and F_{00} pointing to the same function. By applying the BDD reduction rules we eliminate the node redundancy and are left with only F_0 as seen in Figure 2.7a. In Example 2.2.3, swapping variables x and y results in the same number of nodes which follows the same node reduction rules. In both, Example 2.2.2 and 2.2.3, the mirrored situation where variable x would have the right children would result in more nodes and the same number of nodes, respectively.

Lastly, when the concept of the sitting algorithm and an adjacent variable swap are introduced, we introduce the sifting algorithm procedure. The algorithm is based on finding the optimum position for a variable in a forest to minimize the size of a BDD, assuming all variables are fixed. If there are n variables excluding the terminal nodes, then there are n potential positions for a variable, including its current position. The algorithm determines the optimum position for a variable by brute force enumeration as follows; first, the variable is swapped with its successor until it is next to the last (terminal) node. In other words, the variable is sifted down to the bottom of the BDD. Then, the variable is swapped back with its predecessor until there is no predecessor to swap with. In other words, the variable is sifted up to the top of the BDD. The smallest BDD size is remembered during this search and then is restored by sifting the variable down to the optimum position [28].

The asymptotic time complexity of Rudell's sifting algorithm is as follows. Let n be the number of variables in a BDD excluding the terminal nodes. In the optimal case sifting a single variable takes n swaps assuming it is at one of the boundaries of the BDD. In the worst case, the variable is in the middle of the BDD and first needs to be swapped $n/2$ times to one of its boundaries and then n times to the opposite boundary resulting in $n+\frac{n}{2}$ swaps. Moreover, if the optimal position was on the opposite boundary after we finished the search, another n swaps are required to move the variable to its optimum position. The other case is when the search ends right at the optimum position in which case no further swaps are required. Therefore, sifting a variable in a BDD results in $\omega(n)$ and $O(2n + \frac{n}{2})$ swaps. Finally, sifting each variable in the BDD results in an asymptotic time complexity with $O(n^2)$ swaps. To control the worst-case complexity, sifting a single variable is abandoned as soon as the BDD reaches a certain size threshold.

Example 2.2.4. *Let $x_1 \prec x_2 \prec x_3 \prec x_4$, be the initial ordering of variables in a BDD. Suppose the optimal ordering of the variables is $x_1 \prec x_2 \prec x_4 \prec x_3$ and we apply the sifting algorithm to variable x_3. Then, the algorithm performs four permutations to search for the optimum variable position and three more permutations to move the variable to the remembered optimum position. This is in fact the worst-case. The graphical representation of the permutations is depicted in Figure 2.9*

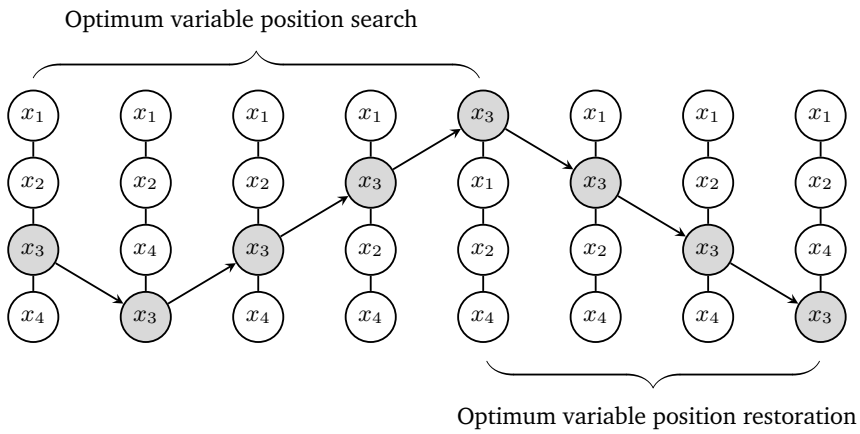


Figure 2.9: Sifting algorithm example

2.3 Lace

In this thesis, we will use a task-based work-balancing method called work-stealing used in the Sylvan BDD package. Work stealing is an efficient method to implement parallelism for small-sized tasks. In particular, we will work with the framework Lace introduced by van Dijk et al. in [9] also used in Sylvan. Lace is an implementation of work-stealing using concurrent deques based on the task split queue. The design of the dequeue includes a dynamic split point between the shared and the private portions of the dequeue. Due to the split point, memory fences are required on when shrinking the shared portion of the dequeue. Processors that are idle become thieves by stealing tasks from the queues of busy processors called victims.

Lace is implemented using C programming language, and its interface is built using C macros. In this thesis, we will use Lace to parallelize dynamic variable reordering. Therefore, the basic API is explained. Firstly, Lace tasks have to be defined using macro `TASK_n` where n is a number of parameters accepted by the task. In order to split declaration from implementation, Lace provides `TASK_DECL_n` for a declaration and `TASK_IMPL_n` for an implementation. Lace task can be then invoked by calling it with macro `CALL` from an existing Lace task or using `RUN` from outside of Lace. The tasks follow the fork-join paradigm. By calling `SPAWN` on a Lace task definition, a task is created. To obtain the result (if stolen) or execute the task (if not stolen), invoke `SYNC` on the same task. With `CALL` a task si pushed to a stack and by calling `SYNC`, a task is then matched form the stack back. An example of a recursively parallelized Fibonacci sequence may be defined as follows :

```
1 TASK_1(int, fibonacci, int, n) {
2     if(n < 2) return n;
3     SPAWN(fibonacci, n-1);
4     int a = CALL(fibonacci, n-2);
5     int b = SYNC(fibonacci);
6     return a+b;
7 }
```

Before invoking a Lace task, first Lace needs to be initialized with the number of workers and the size of the dequeue. Then, macro `RUN` needs to be used to invoke the task from outside of Lace. An example may be defined as follows:

```
1 int main(int argc, char **argv) {
2     int n_workers = 4;
3     lace_start(n_workers, 0);
4     int result = RUN(fibonacci, 42);
5     printf("fibonacci(42) = %d\n", result);
6     lace_stop();
7 }
```

Lastly, we introduce two macros which both work similarly namely, `NEWFRAME` and `TOGETHER`. The `NEWFRAME` macro is used the same way as `RUN` and `CALL`, however, `NEWFRAME` starts one new task and all other workers help execute this task in parallel while `TOGETHER` macro executes a task with local copy given to each worker. For more in-depth information on how Lace works we refer to [9].

2.4 Sylvan

In 2012, van Dijk et al. [11] introduced a parallel (multi-core) multi-terminal binary decision diagram (MTBDD) package called Sylvan written in C programming language. MTBDD is a superset of BDD with arbitrary terminal nodes, mapping from the Boolean space $\mathbf{B^n}$ onto any set. The Sylvan package is implemented using lock-less data structures and the work-stealing framework Lace [9]. In Sylvan, most BDD operations are implemented as recursive tasks, calculated in parallel, e.g., operation on x_i in a BDD is performed in parallel for both sub-graphs $F_{x_i=0}$ and $F_{x_i=1}$ and the final result is computed using a hash table.

The explanation of Sylvan's internal implementation is with emphasis on parts relevant to our implementation of dynamic variable reordering. Sylvan relies on two central data structures established by Somenzi [33], namely the unique table(or nodes table) and the computed table(or operation cache). In Sylvan, the unique table is a hash table which supports garbage collection, it stores the BDD nodes and is used for operations such as a find-or-insert node or delete node to perform the garbage collection. The computed table is a simplified hash table that stores the results of BDD operations and is used as the only shared operation cache to minimize interaction between workers. Both the unique table and the computed table are implemented using a specialized lock-free hash table. The garbage collection is implemented using a mark-and-sweep approach, where the marked nodes are kept during the garbage collection.

The concurrent hashmap implemented in Sylvan uses one table in which all nodes are stored. The table is split into regions and each region can be claimed by a separate Lace worker. Each worker gets assigned its region using a global thread local variable so it knows which region to operate on. Splitting the hash map into the regions allows parallelization of the hash map. Sylvan uses bitmaps to keep track of unique node indices as well as to keep track of claimed buckets. During the garbage collection, the bitmap holding the indices to each unique node is cleared and all externally referenced nodes

are depth-wise traversed and its node indices are reinserted into the bitmap again. After each garbage collection, the global thread-local variables are reset and each worked claims a new region.

In this thesis, we will be only concerned with MTBDDs and consecutively BDDs due to the limitations of the scope. An MTBDD is a 64-bit unsigned integer. The low 40 bits are an index in the unique table. The highest 1 bit is the complement edge, indicating negation. The definition is as follows:

```
typedef uint64_t MTBDD;
```

Sylvan uses two 64-bit unsigned integers to store internal information about each node named `a` and `b`. The definition is as follows:

```
typedef struct __attribute__((packed)) mtbddnode {
    uint64_t a, b;
} *mtbddnode_t;
```

The internal mtbddnode contains the following information. Complement mark (1 bit, first MSB), mark flag (1 bit, second MSB), leaf flag (1 bit, third MSB), map node flag (1 bit, fourth MSB), variable label (24 bits) and unique table node index to high and low children nodes (both 40 bits). Each MTBDD node is labelled with a 24-bit variable label which will be changed during the dynamic variable reordering. See Figure 2.10

Figure 2.10: Internal MTBDD node structure

The bitmaps for tracking nodes and regions are wrapped using C11 atomic semantics which provides the atomic operations such as `store, load, fetch_and_add`, or `compare_and_swap`. Moreover, several memory ordering can be set such as `relaxed, consume, acquire, release, acq_rel` or `seq_cst`. Each memory ordering defines different thread safety. For instance, memory order `relaxed` works as if no thread memory ordering measures are in place, whereas memory order `seq_cst` provides the highest sequential consistency. For more information, we refer to C reference documentation [3]. The structure containing unique table nodes, hashes, ownership bitmap and the bitmap with node indices containing data is defined as follows:

```
typedef struct llmsset {
    _Atomic(uint64_t)* table;    // table with hashes
    uint8_t*           data;     // table with values
    _Atomic(uint64_t)* bitmap1;  // ownership bitmap (per 512 buckets)
    _Atomic(uint64_t)* bitmap2;  // bitmap for "contains data"
    // --snip--
} *llmsset_t;
```

Common BDD algorithms supported by the Sylvan package are `ite, exists, constrain, compose, satcount` and `relprod`. The Sylvan package also provides the functionality to draw DOT graphs and supports BDD file input-outputs [12].

2.5 CUDD

In order to compare and benchmark dynamic variable reordering in Sylvan, a state of art BDD package CUDD is introduced. The Colorado University Decision Diagram Package (CUDD) developed by Somenzi [32] is a package designated for the efficient manipulation of decision diagrams. It supports binary decision diagrams (BDDs), algebraic decision diagrams (ADDs), and zero-suppressed binary decision diagrams (ZDDs). It is a well-recognized package that provides a wide range of functions for creating, and manipulating BDDs, ADDs, and ZDDs, including functions for Boolean operations, quantification, and composition. Among other applications, the library is widely used in formal verification, model checking and model counting.

The BDD operations are implemented as recursive tasks, calculated serially. The CUDD package uses the unique table and the computed table. The unique table contains as many hash tables as there are variables. These hash tables are called unique subtables. The computed table stores the result of the BDD operations.

CUDD supports several dynamic variable reordering algorithms such as the sifting algorithm, simulated annealing, genetic algorithm for variable ordering, or minimization of BDDs based on exchanges of variables. Both the genetic algorithm for variable ordering and the minimization of BDDs based on exchanges of variables are described as potentially slow by Somenzi in [33]. Algorithms such as the group sifting or sifting algorithm combined with the detection of symmetric variables called symmetric sifting have been specifically developed for the CUDD package. The package allows fixing orders of one or more variables. Besides the dynamic variable reordering, the CUDD package also implements asynchronous ordering, which is the reordering triggered automatically either by the increase of the number of variables above a given threshold or when a new internal node is created. After every reordering, the threshold is adjusted. When an operation is interrupted due to the dynamic variable reordering, it is aborted and tried again.

2.6 Usecases of BDDs

BDDs are widely used in, e.g., model checking, model counting, safety synthesis and many other areas [36, 14, 19]. In the model checking, BDDs are used to store sets of states and transitions that are represented by Boolean functions. Model counting is a computational method to count the number of satisfying assignments of a logical formula. BDDs are used in model counting to represent the formula in question, and then algorithms are used to count the number of satisfying assignments. The satisfying assignments are defined as the possible inputs to the formula that make it true. The efficiency of model counting using BDDs depends on the efficiency of the BDDs. Furthermore, Dudek et. al in [14], present ADDMC as a Weighted Model Counting with Algebraic Decision Diagrams. ADDs are supported by Sylvan and the improvements obtained by dynamic variable reordering are directly affecting ADDs as well. Safety synthesis is a computer-aided method for designing safety-critical systems. The goal of safety synthesis is to guarantee that the system behaves safely under all possible scenarios and that it satisfies a set of safety specifications. The BDDs provide a compact

and efficient way to represent the system state space and the transitions between states, which allows for efficient analysis and optimization. The safety synthesis process typically involves using model-checking algorithms, such as the fix point algorithm, that traverse the BDD and check for consistency with the safety specifications.

2.7 Safety games

The benchmark used to evaluate and compare the performance of the dynamic variable reordering is a safety game solver. Therefore, we provide a more detailed introduction to safe games. Reactive Synthesis Competition[1] (SYNTCOMP) is an academic competition where participants submit solvers to solve a specific problem. The solvers are then executed to determine which one can solve the maximum number of problems within a designated time frame. The primary goal is to synthesize a controller or ascertain the existence of one in the realizability track for a finite state safety game. In the Reactive Synthesis Competition, participants submit solvers that aim to synthesize a controller or determine the realizability of a controller for the safety game. The solvers are evaluated based on their ability to solve the maximum number of safety games within a given time limit, ensuring that the safety objectives are satisfied. For the formal definition of a safety game, see Definition 2.7.1

Definition 2.7.1 (Safety game)**.** *A safety game is formulated as a two-player turn-based game with a safety objective expressed symbolically. A game is a 5-tuple* $G = \langle L, X_u, X_c, (f_l)_{l \in L}, \texttt{BAD} \rangle$ *where [5]:*

- L is a finite set of Boolean variables representing latches.
- X_u is a finite set of Boolean variables representing uncontrollable inputs.
- X_c is a finite set of Boolean variables representing controllable inputs.
- $(f_l)_{l \in L}$ is the state transition function, where each $f_l : L \times X_u \times X_c \to L$ defines the next state given the current state, uncontrollable, and controllable actions.
- $\texttt{BAD} \subseteq L$ represents the set of bad states that must be avoided.

Synthesis problem The synthesis problem of the safety games can be described using a sequential digital circuit such as the circuit shown in Figure 2.11. The objective is to synthesize a controller or determine if a controller exists such that the game does not reach any state in the error set BAD. The controller should select appropriate controllable inputs based on L and X_u such that transition function $(f_l)_{l \in L}$ of the system does not transition to an error state, meaning it is safe. The goal is to maintain the system safe throughout the entire game. Note how the safety condition differs from the liveness condition where eventually, the desired property should be true [5, 35].

Safety games The synthesis problem is seen as a game since the controller plays against the environment on each tick of the global clock. The environment sets X_u inputs, and the controller responds by setting X_c such that the system remains safe. The game is solved by an iterative algorithm which computes the set of states from which the environment can force the game into unsafe states to determine whether the environment

[1] http://www.syntcomp.org/

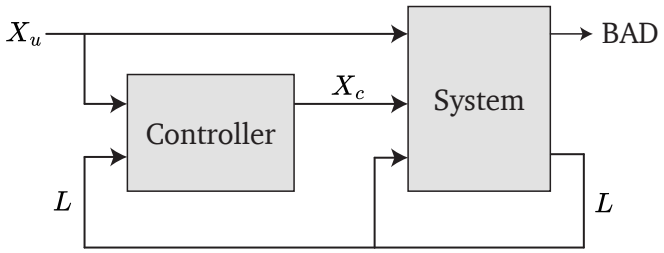


Figure 2.11: Synthesis problem with Controller (unknown) and System

has a strategy that guarantees its victory. If we observe that the environment can force the game to enter the BAD state during the computation, we say the game is unrealizable. Otherwise, the initial state must be in the WIN region as depicted in Figure 2.12, making the game realizable.

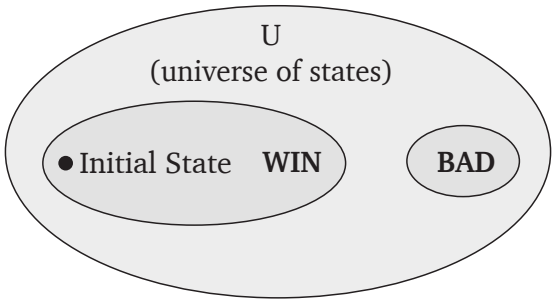


Figure 2.12: Example of when a safety game is solved [35]

Symbolic solving The states of the game can be represented symbolically by a Boolean formula. In the context of the safety games, we represent the winning set using a propositional formula over the game state that is true iff the state is in the winning set. The symbolic states can be encoded using a BDD, which provides a compact and efficient representation of the game state. In this thesis, we focus on games represented by sequential synchronous circuits encoded as And-Inverter Graphs (AIGs) defined in Definition 2.7.2 and represented in the AIGER format. The AIGER models representing safety games are loaded and converted into BDDs and are then used as input data for the safety game solver. For a description of the AIGER format, we refer to [16].

Definition 2.7.2 (AIG)**.** *And-Inverter Graph (AIG) is a directed, acyclic graph that represents a structural implementation of the logical functionality of a circuit or network. It is composed of two input AND gates and inverters. Terminal nodes represent inputs and constants. The AND gates are represented by nodes with two inputs and one output. A complemented edge indicates inversion.*

Chapter 3

Related work

In this chapter, existing dynamic variable heuristics are described. Firstly, the Symmetric and Group sifting designed for CUDD are described. Then, the dynamic lower bounds are introduced which aim to reduce the dynamic variable reordering. Lastly, the evaluation of BDD reordering heuristics and typical use cases of BDDs are presented.

3.1 Symmetry sifting

The effectiveness of sifting depends on the efficiency with which adjacent variables can be swapped. The sifting algorithm can get trapped in local minima because it primarily considers the immediate impact of swapping adjacent variables on the size of the BDD, rather than the long-term effect on the overall structure of the BDD. Additionally, the algorithm does not take into account the relations between variables when determining the optimal ordering, which can also contribute to the problem of getting trapped in local minima. Moller et al. [25] have found that symmetric variables tend to be adjacent in optimum orders for BDDs without complement arcs for most functions with up to five variables. However, Panda et al. in [27] argue that there is a linear gap between optimal orders and the best symmetric orders for cases with and without complement arcs.

Despite the linear gap, Panda et al. proposed in [27] symmetry-sifting algorithm. The algorithm is similar to the sifting algorithm proposed in [28] by Rudell. However, there are a few differences. The symmetry algorithm tests for symmetry every time the adjacency swap is performed. Once two variables are identified as symmetric, they are locked together and their relative position never changes. This leads to sifting groups of variables symmetric to each other. To find the locally optimum group of symmetric variables, they may have to be sifted twice instead of once. The reason is that during a single variable sifting, the additional variable can be locked to the sifted variable which invalidates the best-known position and one additional sifting is needed. The symmetry check is based on Theorem 3.1.1 by Panda et al. [27].

Theorem 3.1.1. *Let F be the BDD for f under order π. Let x_i and x_j be adjacent variables. Then x_i and x_j are symmetric in f if and only if:*

- *For all nodes labeled x_i, the condition $g_{x_i' x_j} = g_{x_i x_j'}$ is verified, where g is the function rooted at the node labeled x_i*

- *All arcs into nodes labeled x_j come from nodes labelled x_i*

The symmetry sifting algorithm requires no additional data structures, however, extra bytes are required per variable to keep track of the symmetry information. The experimental results in [27] show, that the symmetry sifting algorithm improves the effectiveness of the sifting algorithm by making BDDs smaller and keeping the overhead negligible when no symmetries are present.

3.2 Group sifting

The symmetry algorithm explained in Section 3.1 provides a limited advantage over the sifting algorithm proposed by Rudell [28]. Panda et al. argue in [27] that symmetric variables are not often found in practical circuits, and therefore, proposed an extension to symmetric sifting which is a more general version of the algorithm called group sifting. Among the goals of the group sifting algorithm is to be consisted in formulating a dependable criterion for the grouping of variables. Furthermore, Panda et al. propose an aggregation criterion which is based on letting sifting itself identify variables with strong attraction, and on applying a relaxation of the symmetry check called *extended symmetry check* [26].

Group sifting Group sifting is an extension of the symmetry sifting algorithm. A group of variables are moved at the time, where one variable is a special case of a group. Panda et al. define two types of groups, namely *Hard group* and *Soft group*. Hard groups are passed to the reordering procedure by the caller and may contain suggestions on the structure of the ordering. Moreover, hard groups can be nested and *fixed*. The sub-groups of a fixed group do not move relative to its super-group throughout the reordering. Soft groups are groups created by the reordering algorithm when it identifies strong affinity among the variables. The lifetime of a soft group ends in between the successive invocations of the reorderings. Hard groups are structured as a tree and they initially contain no soft groups. The group reordering procedure traverses the tree in a post-order fashion and sifts the children of each non-terminal non-fixed node. The soft groups are only created while reordering a set of individual variables contrary to the reordering of a set of groups.

Variable Aggregation An important aspect of the group sifting algorithm is the aggregation criteria. Panda et al. introduced two criteria, namely *extended symmetry* and the method of the second difference. Also, filtering condition based on variable interaction is applied after some other aggregation criterion. If two variables do not appear in the support of the same output, they are said to be *non-interacting*. All non-interacting variables are not aggregated.

The extended symmetry check builds on the top of Theorem 3.1.1 which shows that all edges going into a variable at level i should come from level $i-1$, and the condition $g_{x_i' x_j} = g_{x_i x_j'}$ must be true for positive symmetry, or $g_{x_i x_j} = g_{x_i' x_j'}$ must be true for negative symmetry. The extended symmetry check is augmented by allowing mixed cases of positive and negative symmetry and a fixed percentage of violations is allowed.

The method of the second difference measures the number of re-combinations occurring between two variables when they are adjacent in a manner that is relatively independent of their position in the BDD. The method of the second difference is defined in Definition 3.2.1.

Definition 3.2.1 (The method of the second difference)**.** *Let $N(i)$ be the number of the nodes labeled i. Let $n-1$ be the largest variable index appearing in the BDD. Let i be such that $0 < i < n-1$. Then, let $S(i) = \frac{N(i+1)}{N(i)} - \frac{N(i)}{N(i-1)}$ where $S(i)$ is a quantity related to the second difference of the sequence $N(i)$.*

Consider Definition 3.2.1, a negative $S(i)$ indicates a lot of re-combinations for given x_i and x_{i+1}. Hence, when aggregating variables, x_i and x_{i+1} should be kept together.

Reordering schemes Besides the aggregation criteria, the Relative Absolute Position (RAP) method is proposed by Panda et al. in [26], which is composed of two phases. In the first phase, a variable that has not been sifted yet is chosen. This variable is shifted up and down while checking for extended symmetry. If an extended symmetry is found, a group is formed. At the end, the variable is returned to the best-known local minima position. During the return, if the best know size was achieved before the group was formed, the group is dissolved. If an extended symmetry has been found, the next variable is chosen for the sifting, otherwise the second phase starts. In the second phase, the variable is checked against the two adjacent variables, to see if a group should be formed. The aggregation test is the method of the second difference followed by the variable interaction check. If the test succeeds, a soft group is formed with a size of up to three and the group goes through the second phase of sifting. If this sifting brings the soft group to a new position, the new group is dissolved.

3.3 Lower bounds in dynamic variable reordering

When using variants of the sifting algorithm that add up in complexity, for instance, the symmetry sifting or the group sifting, the run-time of those algorithms is generally increased. To balance the run-time increase, Drechsler et al. in [13] proposed lower bounds in dynamic variable ordering called lb-sifting. The lower bounds restrict the minimum size of the BDD to which it can fall after reordering the BDD variables. The goal is to stop early while sifting a variable up or down the BDD. The sifting can be stopped if the lower bound already exceeds the smallest recorded BDD size. In such case, we can continue with sifting the next variable without changing the yielded results [13]. Theorem 3.3.1 and Theorem 3.3.2 are described in [15] where $X_n := \{x_1, ..., x_n\}$ are a Boolean variables bounded to value $\mathbf{B} := \{0, 1\}$. The kth element of variable x_i is written as $\pi(k) = x_i$. A BDD node v is labelled with some variable $x_i = var(v)$. The nodes labelled with the same variable are referred to as $nodes(F, x_i) = \{v | v \in V, var(v) = x_i$ where F is a graph $(V, E)\}$. Let, $I \subseteq X_n$ and for a BDD F and $x_i \in X_n$, let $label(F, x_i) = |nodes(F, x_i)|$ and $label(F, I) = |nodes(F, I)|$. BDDs are defined for multi-output functions $f : \mathbf{B^n} \rightarrow \mathbf{B^m}$, using a graph for each of the m single-output functions $f_i^{(n)} 1 \leq i \leq m$ for the shared BDD representation. Given a set O of output nodes of a BDD, notation O_j^i refers to the set of output nodes at levels $i, ...j$. Every root

node is an output node. A set of variables in X_n interacting with $x_i \in X_n$ are denoted with $\mathcal{I}_{i,n}$. d For a detailed explanation and looking at the theory behind the Theorems 3.3.1 and 3.3.2, we refer to [15].

Theorem 3.3.1. *Let F be a BDD over X_n, for which we assume the natural ordering π with $\pi(i) = x_i (1 \leq i \leq n)$. Let $|F'_j|$ denote the size of the BDD after moving variable x_i to position j. When moving down a variable $x_i \in X_n$ as a lower bound on the size of resulting BDD F', we have*

$$
\begin{aligned}
&lb^{\downarrow}(F, x_i) \\
&\quad = label(F, X_{i-1}^1) \\
&\qquad + \max\Big\{label(F, X_n^{i+1} \setminus \mathcal{I}_{i,n}) + 1 + \tfrac{1}{2} label(F, X_n^{i+1} \cap \mathcal{I}_{i,n}), label(F, x_i)\Big\}
\end{aligned}
$$

When moving up a variable $x_i \in X_n$ as a lower bound on the size of resulting BDD F', we have

$$
lb^{\uparrow}(F, x_i) = label(F, X_{i-1}^1 \setminus \mathcal{I}_{i,n}) + |X_{i-1}^1 \cap |\mathcal{I}_{i,n}|\} + \tfrac{label(F,x_i)}{2|X_{i-1}^1 \cap \mathcal{I}_{i,n}|} + label(F, X_n^{i+1}).
$$

Later in 2006, Ebendt et al. in [15], proposed an enhanced method with tightened lower bounds called elb-sifting which combines the lb-sifting with the new tighten bounds. To use the new lower bounds together with bounds in Theorem 3.3.1, it is crucial to avoid counting nodes twice, as this would destroy the soundness of the new combined bound. The bounds while moving up are specified in Theorem 3.3.2 [15].

Theorem 3.3.2. *Let F be a BDD over X_n, with the set of output nodes O, for which we assume the natural ordering π with $\pi(i) = x_i (1 \leq i \leq n)$. When moving up a variable $x_i \in X_n$, as a lower bound on the size of resulting BDD F', we have*

$$
\begin{aligned}
&lb^{\uparrow}(F, x_i) \\
&\quad = \max\Big\{label(F, X_{i-1}^1 \setminus \mathcal{I}_{i,n}) \\
&\qquad + \max\Big\{|X_i^2 - 1 \cap \mathcal{I}_{i,n}| + label(F, \{x_1\} \cap \mathcal{I}_{i,n}), |X_i^1 - 1 \cap \mathcal{I}_{i,n}| \tfrac{1}{2|X_{i-1}^1 \cap \mathcal{I}_{i,n}|} label(F, x_i)\Big\}, \\
&\qquad\quad label(F, X_n^{i+1}) - |O|\Big\} \\
&\qquad + label(F, X_n^{i+1})
\end{aligned}
$$

As shown in [15], the experimental results suggest the lower bounds have a reduction of run-time up to 89.2%. The average improvement is 74.1%. The lower bounds do not impact the quality of the sifting, instead, they stop early the reordering process when no further optimization is possible.

3.4 Dynamic variable reordering scheduling

There are many ways to schedule a variable swap during dynamic variable reordering in a BDD. Jiang et al. in [21] proposed and compared several scheduling heuristics to dynamically reorder variables. The goal was to consider greedy, no lookahead, heuristics which perform a minimum number of swaps. The scheduling heuristics are based on swappable inversion between two variables in the BDD.

Definition 3.4.1 (Swappable inversion)**.** *Let F be a BDD over X_n where π_t denotes the target ordering, π_n the current ordering and let π_{n+1} denote an ordering resulting from swapping two adjacent variables in π_n. Then, we say v_i and v_j form an inversion if their relative orderings are different in π_n and π_t. The inversion is swappable if v_i and v_j are adjacent in π_n. Lastly, let $I(\pi_n, \pi_t)$ denote the total number of inversions between π_n and π_t:*

$$I(\pi_n, \pi_t) = \sum_{1 \leq i,j \leq L, v_i \prec_{\pi_n} v_j} I_{i,j}(\pi_n, \pi_t)$$

where $I(\pi_n, \pi_t) = 1$ if v_i and v_j form an inversion, 0 otherwise.

The following are the scheduling heuristics proposed in [21]:

- Random (RAN): Randomly choose a swappable inversion
- Bring Up (BU): Choose the swappable inversion with the highest variable in π not yet in its final position
- Sink Down (SD): Choose the swappable inversion with the lowest variable in π not yet in its final position
- Lowest Inversion (LI): Choose the lowest swappable inversion
- Highest Inversion (HI): Choose the highest swappable inversion
- Lowest Cost (LC): Choose the swappable inversion where the variable on top has the fewest associated nodes
- Lowest Memory (LM): Choose the swappable inversion that will result in the smallest BDD next

The comparison between the heuristics was done on the largest difference before and after the reordering; the peak number of nodes during reordering, and the required number of swaps. The experimental result in [21] shows, that BU, SD, LI and HI are all similarly superior and reliable in performance. LC did not perform comparably to the before mentioned heuristics, Jiang et al. argue in [21] that instead of the time complexity of a single swap, reordering should focus on the influence of the swaps on the number of nodes. Moreover, Jiang et al. are suggesting that a "good" heuristic mostly requires a low peak memory.

3.5 Evaluation of BDD ordering heuristics

The optimal variable reordering problem for OBDD is NP-Hard [4]. Since there is no algorithm that could solve the optimal variable ordering in polynomial time, unless $P = NP$, polynomial-time heuristics are used. The polynomial-time heuristics generally provide no guarantee of the quality of the solution. To study such heuristics,

fundamental principles of experimental design, randomization, replication, and organization to reduce error are used [17]. In [17], Harlow et al. present a methodology for Computer-aided design (CAD) experiments. The methodology consists of three abstractions, namely *circuit equivalence classes*, *treatments*, and *evaluation*. Moreover, Harlow et al. demonstrated experimental evaluation designs for the dynamic variable reordering heuristics, comparison of the performance of two different BDD packages and others.

The circuit equivalence classes abstraction is necessary to create classes of circuits which are invariant in desired properties, for instance logically equivalent isomorphism equivalence classes. The treatment abstraction involves the application of the evaluated algorithm to each member of the circuit equivalence class, with the goal to minimize the desired cost function. Lastly, the evaluation abstraction covers the evaluation of the results of an experiment consisting of calculating values for the desired cost functions and examining their frequency distributions. Cost functions are defined depending on an application. Harlow et al. in [17] used the final BDD sizes for each instance in a circuit equivalence class.

Chapter 4

Research Methodology

This chapter describes the research problem together with the type of data necessary to conduct the research. Based on the research problem, the methods describing the analysis process of the data are proposed and described. Lastly, the methodological choices are evaluated and justified.

The problem statement described in the introduction leads to the research questions, which aim to bring more knowledge into dynamic variable ordering in the Sylvan BDD package, fine-tuning and comparing different heuristics described in Chapter 3. To provide knowledge on how Sylvan can benefit from the dynamic variable reordering and how the Sylvan user should configure it.

To answer the CRQ, implementation of the dynamic variable ordering in the Sylvan is necessary, as well as evaluation of the reordering performance. For the comparison and evaluation of the reordering, quantitative data in the form of the synthesised Binary Decision Diagram will be used. In particular, the safety synthesis process of automatically constructing a BDD from a given safety specification to check if the system satisfies the specification can be used. In this process, the BDD is used as a compact representation of the state space of the system, which enables efficient verification of the safety properties. In the context of safety synthesis, a BDD solver is used to check the satisfaction of the safety specification with respect to the system's state space, represented by a BDD. An example of a simple BDD solver by Walker was published in The Reactive Synthesis Competition [19].

The dynamic variable ordering will be first implemented using Rudell's sifting algorithm. Then, variable reordering heuristics will be tested with the sifting algorithms, such as the dynamic lower bounds. Implementation of the dynamic reordering and the fine-tuning with Sylvan will answer RQ 1. Once the dynamic variable reordering is implemented and the safety synthesis pipeline is prepared, the results will be evaluated. Equivalence classes will be derived together with the cost functions described in Section 3.5 proposed by Harlow et al. in [17]. The quality of the reordering will be evaluated, meaning the lower the resulting BDD size the better the quality. Also, the runtime of the heuristics will be evaluated, and suggestions will be derived based on the benchmarks on which heuristics improve the reordering and when. Finally, a comparison between the state of art package CUDD will be made under the fairest circumstances. This will ultimately answer RQ 2 and RQ 3. Once all sub-research questions are answered, the

answer to the CRQ can be provided.

Figure 4.1 depicts the workflow and individual phases using which the RQs 1, 2, and 3 will be answered and will lead to the CRQ answer.

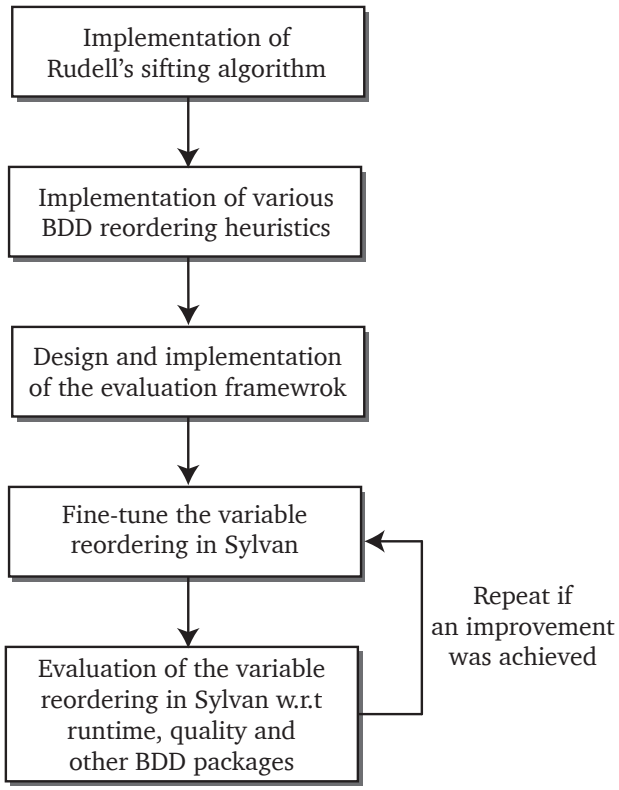


Figure 4.1: Research methodology workflow

The expected challenge is to design an evaluation framework in a way that it will be able to equivalently evaluate parallel BDD package Sylvan as well as non-parallel package CUDD w.r.t.dynamic variable reordering. Moreover, when using the BDD solver as a performance benchmark, it will be necessary to make the Sylvan BDD solver as close as possible to the CUDD solver, which will be necessary to evaluate performance fairly.

The expected work is split into work packages (WP) planned ahead. WP is a group of related tasks in a project. Hence, a list of tasks (T) is provided for each WP. A task is a set of specific actions necessary to complete the WP. The following WPs are defined chronologically in this thesis:

- **Work Package 1**: Implementation of Rudell's sifting algorithm
  - T1.1: Get familiar with the Sylvan code base
  - T1.2: Verify the existing Sylvan adjacent variable swap implementation
  - T1.3: Implement Rudell's sifting algorithm in Sylvan
  - T1.4: Implement tests to verify the correctness of the dynamic variable ordering in Sylvan

  - T1.5: Compare Sylvan hash map with chaining to linear probing
- **Work Package 2**: Implementation of various BDD reordering heuristics
  - T2.1: Select criteria based on which to decide which heuristics to implement
  - T2.2: Implement the chosen heuristics
- **Work Package 3**: Design and implementation of the evaluation framework
  - T3.1: Select criteria for the evaluation framework
  - T3.2: Implement the heuristics evaluation benchmark
  - T3.3: Implement the evaluation benchmark using the Sylvan and CUDD BDD solvers
- **Work Package 4**: Evaluation and fine-tuning of the dynamic variable reordering
  - T4.1: Evaluate and fine-tune the dynamic variable reordering

Chapter 5

Sylvan hash table

This chapter introduces a new variant of Sylvan hash table implementation. A hash table also known as a hash map, is a data structure which implements an associative array or dictionary, mapping keys to a particular value. It uses a hash function using which, a hash (key) is computed given a value. However, it might happen that for a different value the same key is computed, this is called a hash collision. Two common approaches to solving a hash collision are open addressing and separate chaining. The current Sylvan hash map implementation uses open addressing. We will show the limitations of the current Sylvan hash table implementation w.r.t dynamic variable reordering and introduce a hash map variant using separate chaining collision avoidance. The hash map variant was provided as a part of the initial prototype implementation. Furthermore, we will also show the limitations of the mark-and-sweep garbage collection technique w.r.t variable reordering and we will show why reference counting is a more optimal alternative.

Firstly, the current hash map implementation provided in Sylvan will be briefly introduced. Then, the Sylvan hash map implementation with separate chaining will be introduced and described. Furthermore, the mark-and-sweep and reference counting garbage collection techniques will be compared and explained. Both open addressing and chaining approaches will be compared as well, and conclusions will be drawn w.r.t dynamic variable reordering.

5.1 Sylvan hash map

In [11], van Dijk et al. presented the following three variants of Sylvan hash maps 1) variant with reference counter and tombstones, 2) variant with independent locations, and 3) variant with bit arrays to manage the data array. As described in Section 2.4, Sylvan is using a variant with bit arrays also known as bitmaps to manage the data array. The hash map splits the hashes and the data content into two arrays, `table` for storing the hashes and `data` for storing the values. Furthermore, `bitmap1` is used to maintain the ownership of the buckets claimed by a worker, where for instance $n - th$ bit set to 1 means the $n - th$ bucket is claimed by some worker. The ownership is not given back until the garbage collection. After the garbage collection `bitmap1` (ownership) is reset and all workers claim new buckets. The second bitmap maintaining indices

to existing data is stored in `bitmap2`. If $n - th$ bit is set to 1 it means the $n - th$ element in the `data` array contains some valid data. Sylvan uses the *mtbddnode_t* with the size of 16 bytes as a structure stored in the `data` table. The current Sylvan hash table does not provide a function to delete a single entry from the table. Instead, the mark-and-sweep garbage collection technique is used. To implement efficient dynamic variable reordering, we need to be able to delete a single entry from the table. Sylvan hash table implements open addressing collision resolution technique called probing. When a new entry is inserted the buckets are examined from the entry to which the hash points. If it is occupied, we follow the next entry defined by the probe sequence until an unoccupied slot is found. To handle element removal, simply removing the element breaks the probing sequence, causing subsequent elements to be misplaced during future insertions and searches. This issue arises because the probing sequence relies on the consecutive arrangement of elements for correct indexing. One approach to element removal is to utilize tombstones. With tombstone deletion, an element is replaced by a marker called a tombstone, indicating that an element used to be present but has been removed. When performing a lookup, the same procedure is followed: navigate to the hashed location and continue moving forward until an empty spot is found. The concept behind tombstone deletion is that a tombstone is not considered an empty spot, so it is skipped during the search process to locate the desired element. Another approach to deleting a single element is using separate chaining through the use of linked lists. In separate chaining, each element in the hash map maintains a linked list of elements that have collided at that particular hash location. To remove an element, it is necessary to locate it within the linked list and adjust the pointers accordingly. In the new hash map variant, we use spate chaining introduced in Section 5.2.

5.2 Chaining

To support the removal of a single element in the Sylvan hash map, chaining collision avoidance is introduced. The prototype implementation was provided at the beginning of the thesis, since, the current version of Sylvan v1.8.0 only contains a hash table with probing and without the function to delete a single element. The implementation of the hash map with chaining uses the same `sylvan_table.h` header with a macro `SYLVAN_USE_LINEAR_PROBING`. In case the linear probing macro is set to 0, the function `clear_one_hash` which removes a single entry from the hash table and `clear_one_data` which removes a single entry from the data table are included. Two implementation files are provided `sylvan_table.c` which contains the original hash table with probing and `sylvan_table_chaining.c`. Using CMake option `SYLVAN_USE_LINEAR_PROBING`, Sylvan user can select at compile time which implementation will be included.

The structure of the new hash table is depicted in Figure 5.1. Each table slot is 128 bits long where the first 64 most significant bits (MSB) are used as a chain head and a lock when performing an operation on a particular chain. When -1 is set (in hexadecimal `0xffffffffffffffff`) it signals the chain is locked in case another worker tries to modify the chain while some other worker has in progress operation on it. In such a case, new workers accessing the chain wait until the chain is free again using

spinlock. Otherwise, the first 24 MSBs are unused and only the next 40 bits are utilized to store the index to the head of the chain. Then, the next 24 bits are used to store a hash of the entry and the 40 least significant bits are used to store the index of the next item in the chain.

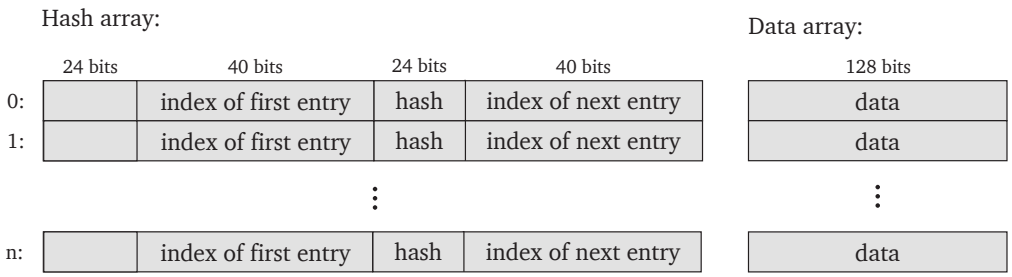


Figure 5.1: Hash and data table designs with chaining

Example 5.2.1. *Let U be the universe of all keys and $K \subseteq U$. Suppose K contains the following set of keys $\{k_1, k_2, k_3, k_4, k_5\}$. Moreover, let hash function h map the keys as follows $h(k_1) = 2$, $h(k_2) = 3$, $h(k_3) = 3$, $h(k_4) = 3$, $h(k_5) = 4$. Then, we insert k_1 at position 2, and k_2 at 3. No next item in the chain is set for both keys, and the head points to themselves. Now, we would like to insert k_3 which should be inserted at position 3, however, it is already occupied by k_2. Instead, we will search for an empty slot in the table which is at position 2. At this point, the head index at position 3 of the chain changes from 1 to 2 and k_3 starts pointing to position 1 where k_2 is. Similarly, k_4 should be inserted at position 3. The next empty slot is at position 3. Hence, it is inserted at position 3. The head index at position 3 is updated from 2 to 3 and k_4 starts pointing to position 2 where k_3 is. Lastly, $k5$ is inserted at position 4 since it is an empty non-conflicting slot. The end result is depicted in Figure 5.2.*

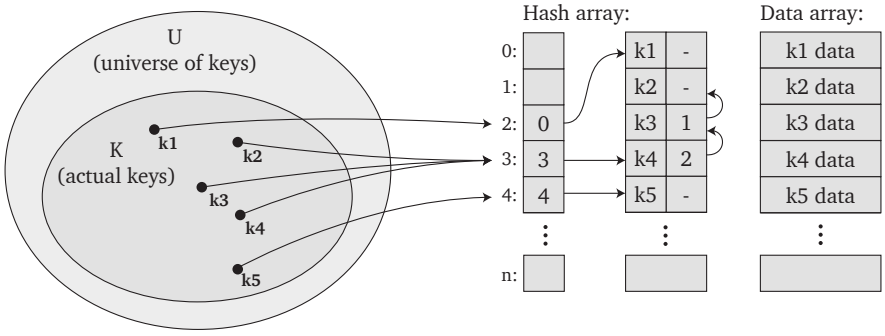
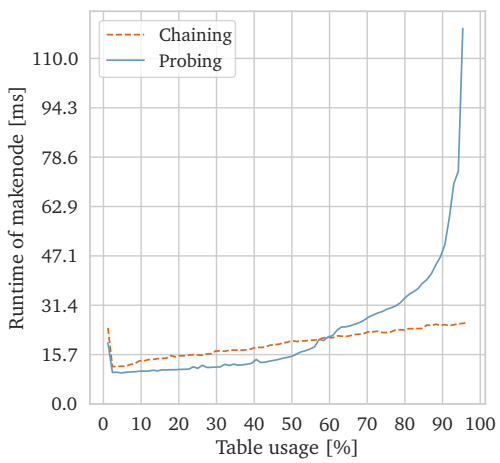


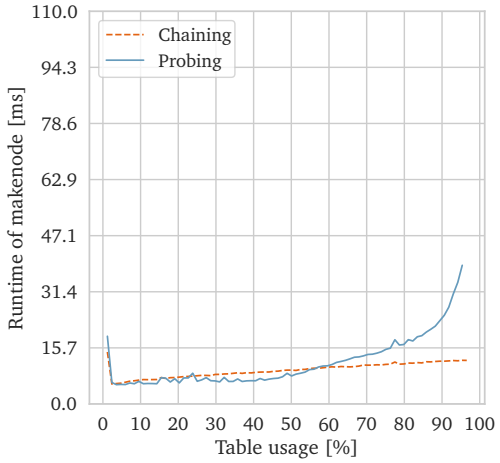
Figure 5.2: Example of the Sylvan hash map with chaining

5.3 Chaining vs probing

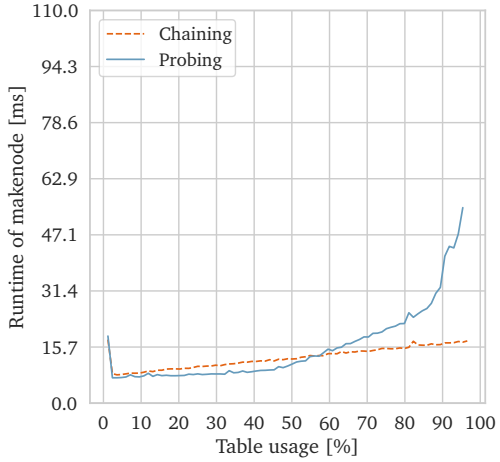
In this section, we will compare the performance impact of using the Sylvan hash map implementations with probing and chaining. The comparison was done by creating nodes until the table is full and measuring the time it takes to create a single node using the Sylvan function `mtbdd_makenode`. Therefore, testing the speed of the lookup function as a consequence. The test was run 10 times where the first run was warm up and its results were not considered. Furthermore, the test was done with 1, 2, 3 and 4 workers. The hash table with chaining introduces an overhead by maintaining the chain with all items for which the hash function produced the same hash. Therefore, it is expected to see runtime superiority of the hash map with probing over the hash map with chaining. In Figure 5.3, we see that indeed probing is superior until approximately 60% of the table capacity. The hash table with chaining linearly increases the runtime of the `make_node` function.



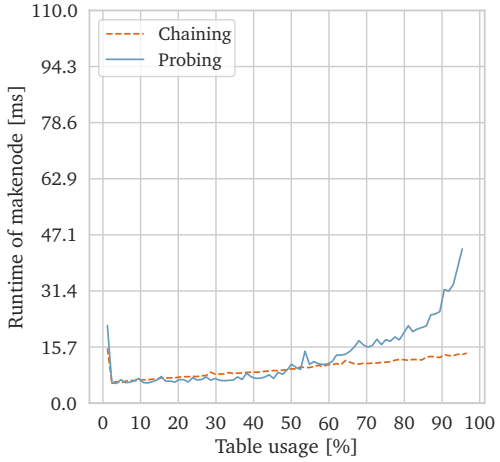
(a) Hash map comparison with 1 Lace worker



(b) Hash map comparison with 2 Lace workers



(c) Hash map comparison with 3 Lace workers



(d) Hash map comparison with 4 Lace workers

Figure 5.3: Hash maps with chaining and probing comparison

5.4 Mark-and-sweep vs reference counting

By using the bitmaps mentioned in Section 5.1, we can determine whether a value in the `data` table is used or not without the need to maintain a counter with references. However, the current Sylvan hash map implementation does not provide a function to delete the individual value from the table. Instead, it uses the mark-and-sweep technique to collect garbage. Mark-and-sweep contains two phases. First, all data that should stay is marked during the marking phase and then during the sweep phase, everything which is not marked is removed. The advantage is that less memory is necessary. However, deleting individual values is impossible since we do not know whether any other objects rely on the value. This only becomes known after the marking phase. An alternative to mark-and-sweep is a reference counter where the counter is used for each object, and every time another object references it, the counter is increased and when the reference is removed the counter is decreased. Using reference counters allows to implement selective garbage collection. For example, an object is considered dead when the reference count reaches 0, meaning it is not used anywhere else and can be deleted selectively.

When implementing dynamic variable reordering, often more than 1000 swaps can be performed in a single second. Each variable swap may produce some nodes which are no longer used and can be deleted. With mark-and-sweep garbage collection, this becomes an expensive run-time operation. After each swap, we need to go through all valid objects in the hash table, mark them and then rehash them again. Whereas, with reference counter, the counters can be updated on-fly during a single variable swap. Then, after each swap dead object can be removed from the hash map.

5.5 Conclusions

The goal of comparing Sylvan hash tables with different hash collision strategies was to understand the impact on the dynamic variable reordering. Both open addressing and separate chaining have their pros and cons. Open addressing, with techniques like tombstone deletion, offers efficient element removal and minimizes memory overhead. However, it can be prone to clustering, where consecutive collisions create long sequences of occupied slots. In [36], van Dijk argues that when using tombstones, over time, all empty buckets become tombstones, and the `find-or-insert` operation is forced to go over the entire probe sequence even if the table is close to being empty. On the other side, separate chaining handles collisions by utilizing linked lists, enabling better distribution of elements which requires additional memory for maintaining the linked list structure and introduces slightly higher overhead during element removal due to traversal operations. Based on the comparison of the `make_node` function using caching and probing, we see that no significant overhead is introduced. However, it is clear that chaining introduces a certain level of overhead when approximately less than 60% of the table is occupied. Furthermore, the chaining implementation allows a single element removal and together with reference garbage collection, they provide an efficient way to remove nodes selectively, which is important for making adjacent variable swap efficient operation as well. Hence, Sylvan hash map implementation with chaining is advised when using dynamic variable reordering.

Chapter 6

Adjacent variable swap

In order to implement the sifting algorithm, a core operation called adjacent variable swap is required. In this chapter, we will describe how adjacent variable swap works in Sylvan, what design choices were made and how bitmaps together with roaring bitmaps are important in making the adjacent variable swap efficient.

As described in Chapter 5, Sylvan has one unique table in which all nodes are stored. The ownership of the table is then split into regions and each region is owned by some Lace worker. The implication is that traversing through all nodes of a particular variable is asymptotically bounded to the number of nodes in the forest. In [33], Somenzi et al. argue that splitting the unique table into subtables with each subtable holding nodes specific to its variable, allows efficient implementation of the dynamic variable reordering. In Sylvan, the current design does not allow splitting the unique table into subtables due to the hash map regions shared between Lace workers. This means we need to make the traversal of the unique table as efficient as possible.

In this chapter, the implementation of efficient bitmap traversal will be detailed. Then, we will show how the traversal can be further optimized using the Roaring bitmaps. The remainder of the chapter will introduce garbage collection using reference counting and describe the design of the adjacent variable swap in Sylvan split into phase 0, phase 1, and phase 2. Lastly, the variable workflow is explained.

6.1 Efficient bitmap traversal

A bitmap, also known as a bit array, is a data structure that represents a sequence of bits, where each bit can be either 0 or 1. It is a simple and efficient way to represent a set of binary values. Sylvan uses the `bitmap2` for storing indexes to nodes in the unique table, meaning if, at a particular position in the bitmap, a value 1 is stored it means that the data table contains some valid data at that location, if the location holds 0, it means the position does not contain any useful data. A bitmap is split into words depending on the architecture of a particular system. The term "word" refers to a fixed-size unit of data that is handled as a single entity by the computer's hardware and software. The size of a word is determined by the architecture and design of the computer system. For example, a typical 64-bit architecture has a word size of 64 bits (8 bytes). This tightly correlates with CPU caches which are organized into a hierarchy of levels 1 (L1), 2 (L2)

and 3 (L3), where each level offers different capacities and speeds. The smallest unit of data that can be transferred between the cache and the main memory is often referred to as a cache line. A cache line typically corresponds to the word size of the computer architecture. By using a 64-bit architecture we can fit 8 words in a single cache line.

To illustrate a bitmap, consider Figure 6.1 with a simple example where word size is equal to 8 bits to make the example more concise. Suppose we would like to iterate only over the indices which contain 1. The naive approach is to go over each bit using bitwise operators and check its value. However, this solution is not scalable with a growing number of elements. The first improvement that can be made is to take advantage of compiler built-in bit counting functions such as GCC `ctz`[1] which returns the number of trailing 0-bits in x, starting at the least significant bit position. The time complexity of the function is $O(\log_2(n))$ which improves finding the successor bit in a word from a linear time to a logarithmic time operation. The second improvement that can be made is to take advantage of the fact that some words may be equal to zero in which case we can skip them during iterations altogether. This, however, strongly depends on how sparse the indices are in the bitmap.

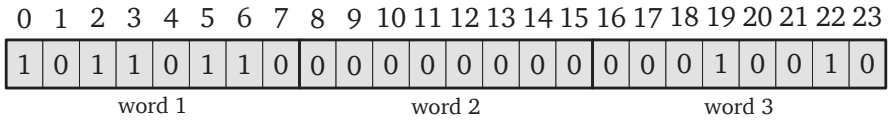


Figure 6.1: Bitmap example

We have implemented a set of bitmap operations compatible with Sylvan bitmaps with two variants namely, regular bitmaps and atomic bitmaps. The atomic bitmaps support the C11 atomic semantics. The implementation offers functions to set, clear or get a particular bit at a given position. To iterate over the bitmap, forward and backward iterators are implemented as well. Using these iterators, both compiler-built-in bits counting as well as skipping empty words are implemented. When profiling the application with the bitmap iterator, the unique table traversal was still the bottleneck. Hence, further optimization was necessary. This leads to the next section in which Roaring bitmaps are introduced and further improve the runtime of unique table traversal.

6.2 Roaring bitmaps

As described in Section 6.2, while compiler built-in bits counting and skipping over empty words provide a runtime improvement, the bottleneck of the application runtime was still unique table data traversal. The limitation of using plain bitmaps is that if the indices are sparse only a few words can be actually skipped and if many consecutive words are empty we still check each word separately instead of directly jumping over to the successor word holding some index. Besides the runtime performance, plain bitmaps have impractical memory usage when the size of the bitmap is large. One approach to avoid these issues is to use compressed bitmaps. Compressed bitmaps employ various techniques to reduce storage and allow efficient querying. For instance,

[1] https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

compressed bitmaps are used by Elasticsearch, Apache Spark, Netflix's Atlas, LinkedIn's Pinot, and others to accelerate queries [23].

One approach to design a compressed bitmap is using Roaring which partitions the space $[0, 2^{32})$ into 16-bit chunks which represent 32-bit unsigned integers of a set S. The 32-bit value stored in S is split into two parts. The most significant 16 bits are used as a shared key coupled with a reference to a value and the remaining least significant 16 bits are stored in a container as the value. Hence, Roaring bitmap is a key-value data structure where each key-value pair represent a set of all 32-bit integers of S that share the same most significant 16 bits [23, 22]. There are three types of containers in which the least significant 16 bits are stored:

- **bitset containers** of 2^{16} bits or 8kB
- **array containers** of 4096 sorted 16-bit integers
- **run containers** of a series of sorted $\langle l, r\rangle$ pairs indicating presence of $[l, l + r]$ values

Each container uses at most 2^{16} bits or 8kB of memory. This makes it possible to fit several containers into the level 1 CPU cache which is typically between 16kB to 64kB The type of a container is determined dynamically based on the values to minimize memory usage. For instance, when performing the intersection operation between two bitset containers, the Roaring bitmap decides whether the result should be an array or another bitset container. In [22], Lamire et al. further specify that any bitset container is required to hold a minimum of 4097 distinct values, whereas an array container can store a maximum of 4096 distinct values. Additionally, if a run container contains more than 4096 distinct values, it must be limited to having no more than 2047 runs; otherwise, the number of runs should be less than half the number of distinct values. For a more in-depth explanation of Roaring bitmaps, we refer to [22].

Roaring provides efficient random access with logarithmic time complexity $O(log(n))$, which is a significant improvement compared to an almost linear time complexity of the bitmap traversal presented in Section 6.2. To determine the presence of a 32-bit integer, a binary search is used on the container corresponding to the sixteen most significant bits of the integer. If this prefix is not found in the list, it infers that the integer is not present. In case a bitmap container is encountered during the search, it checks the corresponding bit. For array or run containers, a binary search is also used to locate the integer efficiently [22]. In this thesis, we use CRoaring[2] implementation of the Roaring bitmap written in C programming language.

6.3 Variable swap phase 0

The initial phase of the adjacent variable swap is called phase 0 during which hash table entries of nodes modified during the swap are removed. The terminal nodes are skipped since they are not swapped and will node be deleted regardless. Performing a removal of an entry from the hash table is not allowed while performing `lookup` operation. The `lookup` operation is performed conditionally based on the results of phase 1. Therefore, we first remove the entries separately in phase 0. To implement

[2]https://github.com/RoaringBitmap/CRoaring

an efficient adjacent variable swap in Sylvan, we need to avoid traversing the unique table as much as possible. Sylvan already contains a plain bitmap with node indices, making it the starting point. As discussed in the above sections, traversing a plain bitmap might be inefficient. Instead of using a plain bitmap during the swap phases, we use the efficient bitmap iterator presented in Section 6.2 to collect the indices into a Roaring bitmap denoted as `node_ids`. Then, in phase 0 we iterate over `node_ids` and remove the target hash entries from the hash table. Besides removing the hash entries, we collect nodes relevant for phase 1 into another Roaring bitmap denoted as `p1_ids`. For instance, suppose we want to swap variable i, then we collect indices of nodes with variable labels i and $i + 1$. As a consequence, we get a Roaring bitmap with random access time complexity $O(log(n_i + n_{i+1}))$ where n_i is the number of nodes for a variable i. This pattern is then repeated in phase 1, where node indices for phase 2 are collected.

The CRoaring bitmap implementation is not thread-safe, so we need to ensure that no two workers are trying to modify the same bitmap simultaneously. One approach is to use a parallel binary reduction pattern where the data is split into two smaller chunks which are then split again recursively. Using this pattern we can provide a unique Roaring bitmap to each worker and when the task is completed we merge the results of the sub-tasks recursively.

Algorithm 1 depicts the swap phase 0 procedure. It utilizes parallel reduction by splitting each task recursively and giving it a new roaring bitmap. Once the tasks are split by the `TASK_SIZE`, each task starts with its own first element and iterates until its own last element. Nodes with the variable labels i and $i + 1$ are collected and used during the variable swap phase 1. Among the tuning parameters of dynamic variable reordering in Sylvan, `TASK_SIZE` is also subject to tuning. The value is set to 1024, which resulted as the most optimal value based on tuning described in Section 8.3.

6.4 Variable swap phase 1

Once the hash entries of nodes with variable labels i and $i + 1$ are removed, we enter phase 1 of the swap. For all nodes with variable $i + 1$ we set a new variable label i and rehash them back into the hash table. For all nodes with variable i depending on a node with variable i, set the new variable label to $i+1$ and rehash them back into the hash table. The remaining nodes with variable label i which are dependent on $i+1$ nodes, their unique table indices are added into `p2_ids` roaring bitmap and are handled in phase 2. The parallel reduction pattern follows similarly as in phase 0. Algorithm 2 depicts the procedure invoked during variable swap phase 1 except the parallel reduction, which follows similarly as in phase 0. Phase 1 performs all the trivial swaps in which only the variable labels are swapped, and no nodes are created. Node deletion is handled separately after a variable swap is finished. We also maintain counters to keep track of the number of nodes per variable which are updated during phase 1. A node is said to be dependent on another node if one of its children's edges points to such a node. Since Sylvan stores all nodes in one unique table unlike CUDD, the dependent node swap is performed in a separate phase to let phase 1 first update all variable indices without interfering.

Algorithm 1 Variable swap phase 0

```
1:  procedure varswap_p0(var, first, count, node_ids, p1_ids)
2:      if count > TASK_SIZE then
3:          split ← count/2
4:          a ← init_roaring_bitmap
5:          SPAWN(varswap_p0, var, first, split, node_ids, a)
6:          b ← init_roaring_bitmap
7:          CALL(varswap_p0, var, first + split, count − split, node_ids, b)
8:          p1_ids.or_inplace(a)
9:          SYNC(varswap_p0)
10:         p1_ids.or_inplace(b)
11:         return
12:     end if
13:     iter ← p1_ids.init_iterator_at(first)
14:     while iter.has_val & iter.curr < first + count do
15:         index ← iter.curr
16:         advance_iterator(iter)
17:         node ← get_node(index)
18:         if node.is_leaf() then
19:             continue
20:         else if node.var = var or node.var = var + 1 then
21:             p1_ids.add(index)
22:             clear_one_hash(index)
23:         end if
24:     end while
25: end procedure
```

6.5 Variable swap phase 2

The last phase of the adjacent variable swap creates new nodes and also as a consequence creates dead nodes which the garbage collector later removes. For all nodes in `p2_ids` we determine the cofactors F_{00}, F_{01}, F_{10}, and F_{11}, and obtain nodes $(F_0, (F_{00}, F_{10}))$ and $(F_1, (F_{01}, F_{11}))$. Then we substitute the outgoing edges with new F_0 and F_1 and rehash both nodes back into the hash table. Besides modifying the nodes, we also update counters used by the garbage collection such as the number of nodes per variable, the number of internal references per node, and a total number of nodes. The Sylvan build function can query the total number of nodes. However, the function performs parallel counting over the nodes, which can be avoided by just maintaining one counter. The counters are later used by a heuristic called dynamic lower bounds implemented with the sifting algorithm. As described in Section 2.2, after a swap, F_0 and F_1 are no longer referenced by F and their reference counters are decreased. For new_F_0 and new_F_1 we increase the reference count since they will be in-place inserted to F. The new nodes might also be existing nodes required by some other functions. If the nodes are created we increase the following counters the number of all nodes, the number of nodes for the variable, and the number of internal references for its cofactors. Next, we

Algorithm 2 Variable swap phase 1

```
1:  procedure varswap_p1(var, first, count, p1_ids, p2_ids)
2:      if count > TASK_SIZE then
3:          // − −parallel reduction − −
4:      end if
5:      iter ← node_ids.init_iterator_at(first)
6:      while iter.has_val & iter.curr < first + count do
7:          index ← iter.curr
8:          advance_iterator(iter)
9:          if node.var = var + 1 then
10:             var_nnodes_add(var, 1)
11:             var_nnodes_add(var + 1, −1)
12:             node.set_variable(var)
13:             rehash(node)
14:             continue
15:         end if
16:         if node.depends_on(var) then
17:             p2_ids.add(index)
18:         else
19:             var_nnodes_add(var, −1)
20:             var_nnodes_add(var + 1, 1)
21:             node.set_variable(var + 1)
22:             rehash(node)
23:         end if
24:     end while
25: end procedure
```

insert the new node indices into the `node_ids`. Lastly, if exception `P2_CREATE_FAIL` is raised, it means the unique table is full and a recovery phase is required followed by table resizing and variable swap repeated from the beginning. In the actual implementation, we use a local reduction pattern using which we collect the counters locally and then update them all at once to avoid accessing shared resources.

Algorithm 3 Variable swap phase 2

```
1: procedure varswap_p2(var, first, count, p2_ids, node_ids)
2:     if count > TASK_SIZE then
3:         //--parallel reduction--
4:     end if
5:     iter ← node_ids.init_iterator_at(first)
6:     while iter.has_val & iter.curr < first + count do
7:         index ← iter.curr
8:         advance_iterator(iter)
9:         F0, F1 ← getlow(node), gethigh(node)
10:        F00, F01, F10, F11 ← getlow(F0), gethigh(F0), gethigh(F1), gethigh(F1)
11:        new_F0 ← nogc_make_node(var + 1, F00, F10)
12:        if new_F0 = invalid then
13:            raise P2_CREATE_FAIL
14:        end if
15:        F0.ref_nodes_add(−1)
16:        new_F0.ref_nodes_add(1)
17:        if new_F0.is_created() then
18:            nnodes_add(1)
19:            var_nnodes_add(var + 1, 1)
20:            F11.ref_nodes_add(1)
21:            F01.ref_nodes_add(1)
22:            node_ids.add(new_F0)
23:        end if
24:        new_F1 ← nogc_make_node(var + 1, F01, F11)
25:        if new_F1 = invalid then
26:            raise P2_CREATE_FAIL
27:        end if
28:        F1.ref_nodes_add(−1)
29:        new_F1.ref_nodes_add(1)
30:        if new_F0.is_created() then
31:            nnodes_add(1)
32:            var_nnodes_add(var + 1, 1)
33:            F10.ref_nodes_add(1)
34:            F00.ref_nodes_add(1)
35:            node_ids.add(new_F1)
36:        end if
37:        make_node(node, new_F0, new_F1)
38:        rehash(node)
39:    end while
40: end procedure
```

6.6 Garbage collection

Among the crucial aspects of making an efficient adjacent variable swap is the ability to selectively delete nodes. Current Sylvan version 1.8.0 relies on the mark-and-sweep algorithm as described in Section 5.1. As a consequence, it is not possible to delete individual nodes unless the table is traversed again which we aim to avoid. The approach introduced in Section 5.4 called reference counting allows deleting individual elements without traversing the entire unique table by maintaining reference counters. As a part of the thesis, we implemented a garbage collector based on reference counting. The individual counters rely on C11 Atomic using which the thread safety guarantees are provided. Counter maintaining number of all nodes in the unique table is of type `size_t` which is an unsigned integer data type defined by C/C++ standards, e.g. the C99 ISO/IEC 9899 standard. On a 64-bit architecture, the data type refers to a 64-bit unsigned integer. All other counters use 32-bit unsigned integers. To avoid overflows and underflows modifying the counters is guarded by flooring min counter value to 0 and ceiling maximum value to $2^{32} - 1$. We defined a counter structure holding an array of counters as follows:

```
1 typedef struct atomic_counters32 {
2     _Atomic(uint32_t) *container;
3     size_t size;
4 } atomic_counters32_t;
```

The downside of garbage collection implemented using reference counters compared to mark-and-sweep is the additional memory requirements to store the counters. In our implementation, the structure holding all data necessary for manual garbage collection (MRC) is defined as follows:

```
1 typedef struct mrc {
2     roaring_bitmap_t*    node_ids;      // unique table node indices
3     _Atomic(size_t)      nnodes;        // # of nodes all nodes in DD
4     atomic_counters32_t  ref_nodes;     // # of internal ref. per node
5     atomic_counters32_t  var_nnodes;    // # of nodes per variable
6     atomic_bitmap_t      ext_ref_nodes; // nodes with external references
7 } mrc_t;
```

Since Sylvan's primary garbage collector remains mark-and-sweep and reference counting is complementing it only during dynamic variable reordering, MRC initialisation is performed before every dynamic reordering procedure and then destroyed once reordering is completed. At every initialisation, `ref_nodes`, `var_nnodes`, and `ext_ref_nodes` are allocated dynamically. The overall number of bits allocated dynamically is respectively as follows $32n + 32v + n$ where n denotes the number of nodes and v denotes the number of variables in the forest and 32 refers to the 32-bit unsigned integer data type. The initialization procedure then traverses through all nodes and updates the counters respectively. Moreover, references created using Sylvan API `sylvan_protect` of `sylvan_ref` are collected into `ext_ref_nodes` bitmap introduced in Section 6.2. We mutate `ext_ref_nodes` concurrently, therefore we use our thread-safe implementation of bitmaps instead of using Roaring bitmaps.

The garbage collection routine is called at the end of every swap to delete all dead nodes. A node can become dead if it is not referenced internally as well as externally and it is still in the `bitmap2`. This means after the swap nodes with external references are never going to be deleted, however, nodes which are not externally referenced and whose internal reference counter reaches 0 will be deleted. To avoid iterating over all nodes in the unique table, we instead use the already created roaring bitmap `p1_ids` which contains all nodes of variables i and $i + 1$ where the reference counters could be decreased depending on the swap. For every node in `p1_ids`, we check whether the node became dead and if so then we perform a deletion. A node deletion involves removing the node index from the `node_ids` roaring bitmap and decreasing the `var_nnodes` counter. In case the given node is not leaf we decrease the respective reference counters and check whether the children are dead nodes as well. If so, then we repeat the procedure recursively until we do not encounter any more dead nodes. We follow this procedure for all nodes in the `p1_ids` roaring bitmap. After the garbage collection is completed, we reset the global thread local regions assigned to each worker. Moreover, we also clear the ownership `bitmap1`.

The garbage collection depends on the macro `USE_LINEAR_PROBING`. Based on the option flag given during compilation to CMake, the macro `USE_LINEAR_PROBING` can be either 1 or 0. If the linear probing table is not used, then during the garbage collection nodes are node deleted individually. Otherwise, we iterate over the entire unique table by invoking two functions namely, `clear_and_mark` and `rehash_all`. This has a negative performance impact compared to using the Sylvan hash table with chaining.

6.7 Variable swap workflow

By invoking the procedure depicted in Algorithm 4, all varswap phases including the garbage collection are executed. Suppose we want to sift down a variable at level i with $i + 1$, then providing level i is enough. When sifting up a variable, level $i - 1$ needs to be provided which will swap the variable at level $i - 1$ with level i.

The swap procedure also depends on whether the Sylvan hash map with linear probing or chaining is used. Besides the garbage collection which depends on the type of the table used, phase 0 is invoked only when chaining is used. This means `p1_ids` are not collected when using linear probing, and hence we traverse through the entire unique table in both phase 1 as well as during garbage collection. Lastly, after the garbage collection, we update `level_to_order`, and `order_to_level` by invoking `swap_level_mappings`.

In case phase 2 raises an exception, we perform a recovery phase in which we undo the current changes. The exception is raised if the table is full and no new node could be created. After the recovery, we raise an exception again to let the caller handle the swap error.

Algorithm 4 Variable swap

```
1:  procedure varswap(var)
2:      p1_ids ← init_roaring_bitmap
3:      p2_ids ← init_roaring_bitmap
4:      if USE_LINEAR_PROBING then
5:          clear_all_hashes()
6:          varswap_p1(var, 0, nodes.table_size, mrc.node_ids, p2_ids)
7:      else
8:          varswap_p0(var, 0, nodes.table_size, mrc.node_ids, p1_ids)
9:          varswap_p1(var, 0, nodes.table_size, p1_ids, p2_ids)
10:     end if
11:     if p2_ids.cardinality > 0 then
12:         try
13:             varswap_p2(var, 0, nodes.table_size, p2_ids, mrc.node_ids)
14:         catch P2_CREATE_FAIL exception
15:             varswap_recovery(var, mrc.node_ids)
16:             raise P2_CREATE_FAIL
17:         end try
18:     end if
19:     if USE_LINEAR_PROBING then
20:         mrc_gc(mrc.node_ids)
21:     else
22:         mrc_gc(p1_ids)
23:     end if
24:     swap_level_mappings(var, var + 1)
25: end procedure
```

Chapter 7

Sifting

In this chapter, we will introduce the sifting algorithm implementation in Sylvan. Once the adjacent variable swap is implemented, the algorithm is straightforward as described in Section 2.2. However, plain sifting does not need to provide the most optimal results. For instance, the algorithm does not stop early in situations in which no further improvements can be made which negatively impacts runtime. One approach to overcome redundant swaps is to use the dynamic lower-bounds heuristic. We will show how the dynamic lower bounds can be implemented together with the sifting algorithm in Sylvan.

Firstly, we will introduce the concept of variable interaction and we will show why it is needed to implement the dynamic lower bounds. Then, the dynamic lower bounds will be introduced. Finally, Rudell's sifting algorithm utilizing the dynamic lower bounds will be described after which the tunning parameters to adjust the reordering will be detailed.

7.1 Variable interaction

Consider Figure 7.1, if $F_{00} = F_{01}$ and $F_{10} = F_{11}$, then x does not depend on y. If this is the case for all the nodes of variable x, we say that variables x and y do not interact. If x and y do not interact, then it is not necessary to perform the adjacent variable swap since the resulting BDD would be logically the same function. This observation can be used to improve the runtime of the adjacent variable swap [33]. Sommenzi et al. argues in [33], that in practice over 90% of swaps can be performed inexpensively, e.g. if we are swapping non-interacting variables. However, this highly depends on a use case which we will discuss in Chapter 10.

Information about which variables interact together can be utilized during an adjacent swap operation and also it is necessary information to implement the dynamic lower bounds introduced in Section 7.2. By counting the number of nodes for each interacting variable during a sifting, we can decide to stop early if no further improvement can be obtained. It is enough to collect information about which variables interact together only once before each reordering since reordering will not affect it as we will show later in this section. To store variable interaction information we use the `atomic_bitmap_t` structure presented in Section 6.2. By using this data structure we

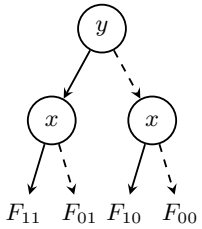


Figure 7.1: Variable interaction between x and y

reduce necessary memory requirements by only using 1-bit for each interaction and have thread-safety guarantees. Definition 7.1.1 provides a formal definition of this data structure holding the variable interaction.

Definition 7.1.1 (Variable interaction matrix)**.** *Let the interaction matrix I be the square matrix over $\{0, 1\}$, and let $I_{ij} = 0$ if and only if i-th and j-th variables satisfy [33]*:

$$f_{|x=0} = f_{|x=1} \vee f_{|y=0} = f_{|y=1}$$

To compute the variable interaction matrix, we use the external references from which we perform a depth-first search. This ensures any sub-function used by the externally referenced nodes will be reached and included in the matrix. As pointed out by Sommenzi et al. in [33], Definition 7.1.1 does not identify all non-interacting variables. Example 7.1.1 depicts a case in which two different situations yield the same results for the variable interaction matrix.

Example 7.1.1. *Consider Figure 7.2a, and let $D \equiv F$. The interaction matrix would conclude A and F interact. Now, in Figure 7.2b we swap variables A and B. The interaction matrix will still conservatively conclude A and F interact even though A and F do not interact anymore.*

Figure 7.2: Variable interaction matrix limitation

7.2 Dynamic lower bounds

When sifting a variable up or down, we might reach a point at which no further improvement can be made. In particular, if the number of nodes with target variables is 0.

By using the variable interaction matrix and maintaining the number of nodes per variable we can efficiently calculate this information before every swap and decide whether to proceed further or not. In fact, it is the idea behind the dynamic lower bounds introduced in Section 7.2. In this thesis, we implemented simple bounds calculations for sifting up and sifting down specified in Theorem 7.2.1 from Fabio et al. [33].

Theorem 7.2.1. *Let F be a BDD over X_n, and let N_i be the number of nodes at level i, with $0 \leq i < n$, for which we assume the natural ordering π with $\pi(i) = x_i (1 \leq i \leq n)$. When moving up a variable $j \in X_n$, the BDD size can not be reduced below:*

$$lb^{\uparrow}(j) = N_0 + \sum_{j \leq i < n} N_i$$

When moving down a variable $j \in X_n$, the BDD size can not be reduced below:

$$lb^{\downarrow}(j) = N_j + \sum_{0 \leq i < j} N_i$$

When sifting a variable up, $lb^{\uparrow}$ implies that the lower part of the BDD is unaffected, and $lb^{\downarrow}$ implies the upper part of the BDD is unaffected when sifting down. In fact, this implication follows implications of Theorem 2.2.1. The bounds can be further improved by using the variable interaction matrix especially when many individual BDDs are sifted. Thus, instead of counting every level, we only consider levels which interact with the target variable. The goal of the sifting algorithms is to identify the smallest encountered size and remember its position. Hence, we need to provide two arguments namely, `best_size` and `best_pos` which will be used to store this information. We assume that the arguments `best_size` and `best_pos` are pointer types and are mutated in place, hence mutating the value pointing to. Both arguments will be used later in the sifting algorithm in Section 7.3.

Sifting up Algorithm 5 depicts the sifting up procedure. The `l_bound` expresses the lower bound on the number of nodes. In other words, how many nodes can not be removed by further sifting up. Thus, we initialize the `l_bound` with the current number of nodes, and then we subtract interacting nodes which are in the lower part of the forest with respect to the variable at pos. These are the nodes that will not be removed and are part of the BDD. Next, we subtract nodes of y since they will not be removed regardless as shown in Section 2.2. Since we are changing the variable labels, we need to use the `level_to_order` mapping with the interaction matrix which was initialized at the very beginning of the reordering. Now the `l_bound` is initialised, and we proceed further with the algorithm by swapping the variable from its current position to the `low` which is defined by the caller. Since we are sifting up and the $varswap$ procedure swaps i with $i + 1$, we provide it $pos - 1$. Then, the best size and position are captured if an improvement was obtained. Now we update the `l_bound` with the new level y into which we swapped the variable in case it interacts with x. Finally, we update the `limit_size` if we reduced it by the `varswap` to keep up to date number of all nodes.

Algorithm 5 Sifting up

```
1:  procedure sift_up(pos, low, best_size, best_pos)
2:      limit_size, l_bound ← size
3:      x, y, y_index ← low, pos, level_to_order[pos]
4:      while x < pos do
5:          if test_interaction(level_to_order[x], y_index) then
6:              l_bound ← l_bound − var_nnodes_get(x)
7:          end if
8:          x ← x + 1
9:      end while
10:     l_bound ← l_bound − var_nnodes_get(y)
11:     while pos > low & l_bound ≤ limit_size do
12:         x, y, x_index ← pos − 1, pos, level_to_order[x]
13:         varswap(x)
14:         if size ≤ best_size then
15:             best_pos, best_size ← pos, size
16:         end if
17:         if test_interaction(x_index, y_index) then
18:             l_bound ← l_bound + var_nnodes_get(y)
19:         end if
20:         if size < limit_size then
21:             limit_size ← size
22:         end if
23:         pos ← pos − 1
24:     end while
25: end procedure
```

We repeat these steps until we reach the caller defined `low` or the `l_bound` reaches the current number of nodes which indicates there are no nodes to remove anymore.

Sifting down Algorithm 6 depicts the sifting down procedure. The `r_bound` expresses the upper bound on node decrease. In other words, the number of nodes that still can be deleted by further sifting down. Therefore, we start by initializing `r_bound` to zero, and then we add all interacting nodes in the upper part of the forest to `r_bound` with respect to the variable at pos. Then, we start swapping the variable until either we reach caller defined `high` or there are no nodes that could be deleted. Since we express `r_bound` as the upper bound on node decrease, we remove the interacting nodes for the swapped level before each swap. Lastly, we swap the variable and update `best_size`, `best_pos`, and `limit_size` as we did in Algorithm 5.

Algorithm 6 Sifting down

```
1: procedure sift_down(pos, high, best_size, best_pos)
2:     r_bound, y, x_index, limit_size ← 0, high, level_to_order[pos], size
3:     while y > pos do
4:         if test_interaction(x_index, level_to_order[y]) then
5:             r_bound ← r_bound + var_nnodes_get(y)
6:         end if
7:         y ← y − 1
8:     end while
9:     while pos < high & size − r_bound < limit_size do
10:        x, y, y_index ← pos, pos + 1, level_to_order[y]
11:        if test_interaction(x_index, y_index) then
12:            r_bound ← r_bound − var_nnodes_get(y)
13:        end if
14:        varswap(x)
15:        if size ≤ best_size then
16:            best_pos, best_size ← pos, size
17:        end if
18:        if size < limit_size then
19:            limit_size ← size
20:        end if
21:        pos ← pos + 1
22:    end while
23: end procedure
```

7.3 Rudell's sifting algorithm

In this section, we will present the main algorithm used in Sylvan to dynamically reorder variables, built on top of all previously presented algorithms. In particular, we will present the implementation of Rudell's sifting algorithm introduced in Section 2.2. As described earlier, the algorithm reduces the number of nodes by sifting each variable given all other variables are fixed in their position, up and down and then returning to the most optimal position and proceeding further until all variables are sifted. The algorithms for sifting up and down are described in Algorithm 5 and Algorithm 6 respectively. Now we introduce the sifting back algorithm which is the last algorithm necessary to introduce Rudell's sifting algorithm.

Sifting back Algorithm 7 depicts the sifting back procedure used for returning to the most optimal position. Sifting back is a straightforward procedure, given a variable at position `pos`, `best_size`, and `best_possition`, it sifts the variable either to the `best_possition` or to the closest position with the number of nodes equal to `best_size`. We use two while loops using which the variable is either shifted down or up. Similarly, as with sifting up and down, we assume the arguments `pos`, `best_size`, and `best_possition` are a pointer type and the values are mutated in-place.

Algorithm 7 Sifting back

```
1: procedure sift_back(pos, best_size, best_pos)
2:     while pos ≤ best_pos do
3:         if size = best_size then
4:             return
5:         end if
6:         varswap(pos)
7:         pos ← pos + 1
8:     end while
9:     while pos ≥ best_pos do
10:        if size = best_size then
11:            return
12:        end if
13:        varswap(pos − 1)
14:        pos ← pos − 1
15:    end while
16: end procedure
```

Rudell's sifting Algorithm 8 depicts the Rudell's sifting algorithm. The public API using which dynamic variable reordering is triggered is named `reduce_heap`. It internally first calls pre reordering procedure, then sift procedure and lastly, post reorder procedure. The pre and post-reordering procedures take care of memory allocation and deallocation as well as initializing and deinitializing necessary objects such as MRC or interaction matrix. The algorithm starts by sorting the existing levels in descending order based on the number of nodes per each level. Sorting the levels is an inexpensive operation thanks to the counters maintained by MRC holding a number of nodes per level. We only need to query counters per each level and sort them in descending order. Then, we mark with -1 each level which contains less number of nodes than the provided `nodes_threshold` value. Furthermore, we initialize all states such as `i`, `pos`,`best_pos`,`best_size`. Lastly, before sifting the variables based on the order given by `levels`, we capture the current `level_to_order` mapping into `old_level_to_order`. We need to capture the mapping since the variables will be sifted which would invalidate `sorted_levels` variable.

Now we proceed further with the while loop by getting the variable position. Firstly, we obtain the respective level from the `sorted_levels`. If the variable is marked (-1) already we can break the while loop. Since the levels are sorted there are no more levels to sift. Otherwise, we use the captured mapping the obtain the correct variable which we then translate back to the actual level. We use two mappings to maintain the information linking which variable is currently at the initial sorted variable `sorted_levels`. Next, we check the boundaries defined by the caller. Lastly, before invoking the sifting procedures we update the current `pos`,`best_pos` so it corresponds to the particular level we will sift as each level is sifted independently.

Invoking the sifting procedures is split into four cases, firstly the boundaries `low` and `high` are checked. In case we are at the boundaries, we only need to shift either up or down depending if we are at the low or high boundary respectively. Otherwise,

we need to determine which boundary is closer and shift to that boundary first. The boundary distance is determined by comparing $pos - low$ with $high - pos$. If $pos - low$ is larger than $high - pos$ it means we are further away from the upper boundary, hence we are in the lower part of the forest and we need to sift down first and then up and finally back to the optimum position. Finally, the else branch is executed which means we are in the upper part of the forest, hence we sift up, down and return back to the optimum position. We catch `P2_CREATE_FAIL` exception and call post-order procedures to perform the cleanup, then we invoke the mark-and-sweep Sylvan garbage collection which also resizes the table. Finally, we are able to start the reordering again with the resized table by invoking the pre-reordering and sift procedures.

7.4 Reordering configurations and callbacks

Using dynamic variable reordering might perform differently in different use cases, it might also be that the user has specific needs such as levels restriction or reordering time restrictions. For these needs, we introduce the sifting configurations using which Sylvan user is able to adjust the sifting. Moreover, we also introduce several callbacks using which different stages of sifting can be tracked.

Reordering configs The structure holding the configurations for the dynamic variable reordering is defined as follows:

```
typedef struct reorder_configs
    uint32_t          threshold;       // max number of nodes per level
    double            max_growth;      // max. allowed size growth
    uint32_t          max_swap;        // max. number of swaps per sifting
    uint32_t          max_var;         // max. number of vars swapped per sifting
    double            time_limit_ms;   // time limit in milliseconds
    reordering_type_t type;            // type of reordering algorithm
    bool              print_stat;      // flag to print the sifting results
} reorder_configs_t;
```

Each struct member can be configured using public Sylvan API. Sylvan compiled doc will contain all information about the reordering API. In terms of the `reordering_type_t`, we currently provide two options `SYLVAN_REORDER_SIFT` which is plain sifting without the dynamic lower bounds and `SYLVAN_REORDER_BOUNDED_SIFT` which is the variant shown in this thesis utilizing the dynamic lower bounds.

Sifting callbacks In case Sylvan user is interested in tracking different stages of reordering we provide three callbacks namely, pre-reordering, progress, and post-reordering callbacks. The progress reordering callbacks are invoked after each variable sifting if the number of nodes was reduced. See below for an example of how to attach the callbacks:

```
VOID_TASK_0(reordering_start) {
    // execute custom pre-reordering procedures here...
```

```
}
VOID_TASK_0(reordering_progress) {
    // execute custom progress-reordering procedures here...
}
VOID_TASK_0(reordering_end) {
   // execute custom post-reordering procedures here...
}
int main(int argc, char **argv) {
    // --snip--
    sylvan_re_hook_prere(TASK(reordering_start));
    sylvan_re_hook_progre(TASK(reordering_progress));
    sylvan_re_hook_postre(TASK(reordering_end));
    // --snip--
}
```

Reorder stats To analyse the performance of the reordering we collect three Sylvan statistical data points attributes, namely the number of reordering calls, the number of reordering swaps and the total time spent on the reordering. To use Sylvan stats, it is necessary to enable the CMake option `SYLVAN_STATS`. See below for an example output of reordering statistics:

```
Variable reordering
RE executions        1
RE swaps             1,183
Total time spent     0.328742 sec.
```

Algorithm 8 Rudell's sifting

```
1: procedure sift(low, high, nodes_threshold)
2:     sorted_levels ← get_descending_levels_sorted_by_nnodes()
3:     sorted_levels ← mark_skipped_levels(sorted_levels, nodes_threshold)
4:     i, pos, best_pos, best_size ← 0, 0, 0, size
5:     old_level_to_order ← level_to_order
6:     while i < number_of_levels do
7:         level ← sorted_levels[i]
8:         if level = −1 then
9:             break
10:        end if
11:        pos ← order_to_level[old_level_to_order[level]]
12:        if pos < low or pos > high then
13:            continue
14:        end if
15:        best_pos, best_size ← pos, size
16:        try
17:            if pos = low then
18:                sift_down(pos, low, best_size, best_pos)
19:                sift_back(pos, best_size, best_pos)
20:            else if pos = high then
21:                sift_up(pos, high, best_size, best_pos)
22:                sift_back(pos, best_size, best_pos)
23:            else if (pos − low) > (high − pos) then
24:                sift_down(pos, low, best_size, best_pos)
25:                sift_up(pos, high, best_size, best_pos)
26:                sift_back(pos, best_size, best_pos)
27:            else
28:                sift_up(pos, high, best_size, best_pos)
29:                sift_down(pos, low, best_size, best_pos)
30:                sift_back(pos, best_size, best_pos)
31:            end if
32:        catch P2_CREATE_FAIL exception
33:            post_reorder()
34:            sylvan_gc()
35:            pre_reorder()
36:            return sift(low, high, nodes_threshold)
37:        end try
38:        i ← i + 1
39:    end while
40: end procedure
```

Chapter 8

Evaluation of the dynamic variable reordering

To understand the strengths and weaknesses of the dynamic variable reordering implementation, we will evaluate it through a series of benchmarks. Firstly, we will introduce the evaluation framework[1] with which all the benchmarks are implemented. Then we will profile the reordering procedures to identify the bottlenecks and possible areas for future work. Furthermore, we will evaluate the performance of the dynamic variable reordering to achieve the most optimal results. After evaluating the dynamic variable reordering in Sylvan, we will compare it in a fair way with the state-of-the-art BDD package CUDD.

8.1 Experimental setup

Several benchmarks have been set up to evaluate the dynamic variable rendering implementation in Sylvan. To keep the benchmarks in one environment, we set up a Docker file using which a Docker container can be created. Docker is a tool which decouples the operating system from the requirements of an application. The evaluation framework contains all benchmarks presented in this chapter and also in Chapter 5. The benchmarks were compiled and ran on a cluster computer with the test machine Dell R750XA with the 2xSilver-4314 CPU, where each benchmark was assigned 16 cores and 16GB RAM. For measuring many consecutive reordering runtimes, we used the benchmarking tool Hyperfine[2].

Variable reordering in BDDs provides two primary benefits, reduced runtime and memory consumption. The reduced memory consumption is referred to as the quality of the reordering, where a smaller number of nodes (less memory) means higher reorder quality and vice versa. Therefore, three types of experiments are set up to evaluate the runtime and quality namely, reordering profiles, Sylvan regression tests, and comparison between Sylvan and CUDD.

The reordering profiles provide insight into the runtime effect of the procedures invoked during the reordering. The reordering procedure includes procedures which are

[1] https://github.com/apdofficial/sylvan-benchmarks
[2] https://github.com/sharkdp/hyperfine

ideally invoked only once, such as `pre_reorder` or `post_reorder` and other procedures which are invoked many times, such as the `sift_up`, `sift_down` or `sift_back`. This implies that the runtime effect of the procedures is different depending on the frequency of the reordering invocations. Therefore, the profiler was executed on two use cases namely, manual reordering and semi-automatic reordering. Manual reordering runs exactly once, whereas semi-automatic reordering is invoked by calling a test procedure that determines whether the reordering is necessary. We use manual reordering to sift 169619 nodes loaded at once from the publicly accessible model `add10y.aag`[3]. The semi-automatic reordering was triggered four times for the same model as the manual reordering in one execution with the following number of nodes 4640, 661, 1421, 3016. The model represents a safety game and is part of the Reactive Synthesis Competition, as Section 2.7 explains.

The regression tests provide insight into how tuning parameters affect runtime and quality of the reordering. We tested various values for `TASK_SIZE` used to set the granularity of the parallelization, *max growth*, which controls how much a BDD can grow during sifting up or down, *nodes threshold* using which BDDs with the less than desired size will be skipped during the reordering, and *number of workers* to understand how the reordering scales.

The Sylvan comparison with CUDD gives an insight into how the dynamic reordering performs in real use cases compared to the state of art CUDD package. To compare the BDD packages in a fair way, we use the same Safety Games models from which BDDs were constructed. As a benchmark, we use a BDD safety game solver developed by Walker[4] for the Reactive Synthesis Competition, which supports both Sylvan and CUDD. The solver won the 2014, 2015, 2016 and 2017 sequential realizability tracks [34]. Since the solver supports both Sylvan and CUDD, it is an ideal candidate for a fair comparison, as we can select desired reordering trigger points with the same BDDs to reorder and measure the resulting runtime and quality. The comparison will be split into two parts. Firstly, we will provide the same conditions to both Sylvan and CUDD, such as the same BDDs with the same number of nodes and trigger the reordering simultaneously. Secondly, we will compare Sylvan and CUDD from the user perspective, meaning we will compare the actual runtime of the entire application rather than just the runtime of the reordering. Moreover, we will use CUDD automatic and Sylvan semi-automatic reorderings and compare the runtimes and quality. Both packages were compiled using the same test machine and software configurations with the -O3 optimisation flag.

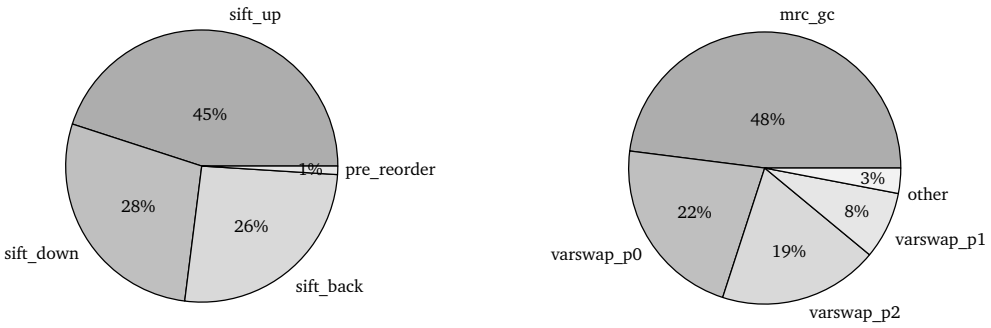
8.2 Reordering procedure profiles

This section presents the profiler results of manual and semi-automatic reordering. Each reordering starts by calling the main reordering procedure, which contains the following sub-procedures `pre_reorder`, `sift_up`, `sift_down`, `sift_back`, and, `post_reorder`. The sifting procedures are explained with their sub-procedures in detail in Chapter 6, and adjacent variable swap procedures are described in Chapter 7.

[3] https://github.com/SYNTCOMP/benchmarks

[4] https://github.com/apdofficial/sylvan-benchmarks/tree/main/syntcomp

Manual reordering Figures 8.1, 8.2, and 8.3 contain profiles for procedures executed during the manual ordering. Firstly, we observe that Figure 8.1a shows the `pre_reorder` procedure takes 1% of the relative runtime, which means no significant resources are used even when we provide 169619 nodes to initialize the reordering objects. During the `pre_reorder` procedure, the variable interaction matrix and the reference counters are initiated. The profile of `varswap` in Figure 8.1b shows that when executing a larger model, the garbage collection procedure takes almost half of the runtime. In particular, as seen in Figure 8.2a, the majority of time is spent on `mrc_gc_go`, which performs the actual deletion of the nodes. The `mrc_gc_go` procedure iterates over the modified nodes and recursively deletes dead nodes as described in Section 6.6.

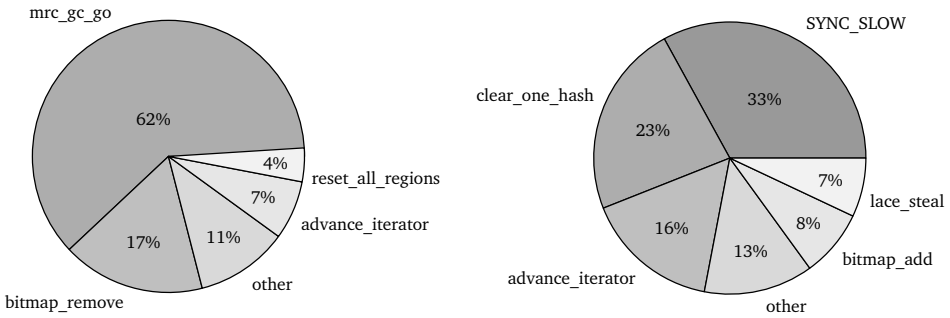


(a) reduce_heap (b) varswap

Figure 8.1: Manual reordering `reduce_heap` and `varswap` profiles

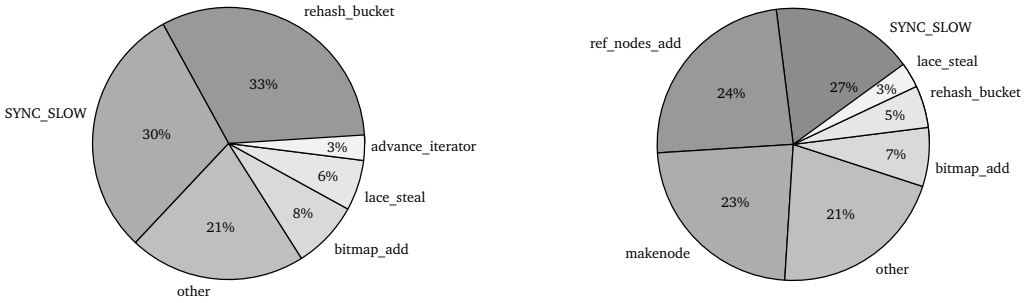
Now, consider Figures 8.2b, 8.3a, and 8.3b, each contains `SYNC_SLOW` which is function generated by Lace for each procedure respectively. The relatively high runtime spent on `SYNC_SLOW` could indicate that tasks need to be frequently moved from the shared part of the Lace stack back to the private part, suggesting a possible shortage of available tasks for stealing. This may result in a higher fraction of the queue being shared. However, as we will see later in this Section, we need to consider a trade-off between the number of tasks and task cost. The tuning parameter `TASK_SIZE` is shown in Algorithm 1, and its impact on the runtime is presented in Figure 8.9. The `clear_one_hash`, `rehash_bucket`, and `makenode` are expected to take a significant part of the runtime. However, in Figure 8.1b, we see a significant part of the runtime is taken by `ref_nodes_add`, which mutates the atomic reference counters `ref_nodes` described in Section 6.6 and is a possible hotspot. The `lace_steal` is a Lace internal function called during a task stealing.

Semi-automatic reordering Figures 8.4, 8.5, and 8.6 depict the profiles of procedures executed during the automatic reordering. The main procedure in Figure 8.4a shows results similar to manual reordering. Procedure *pre_reorder* is expected to take less runtime since the number of nodes given to reordering is smaller. However, at the same time, it is executed multiple times. In Figure 8.4b, `mrc_gc` procedure is more balanced than other procedures, which is expected since the garbage collection is done on fewer nodes. Nevertheless, the runtime of procedure `reset_all_regions` now increased due



(a) mrc_gc (b) varswap_p0

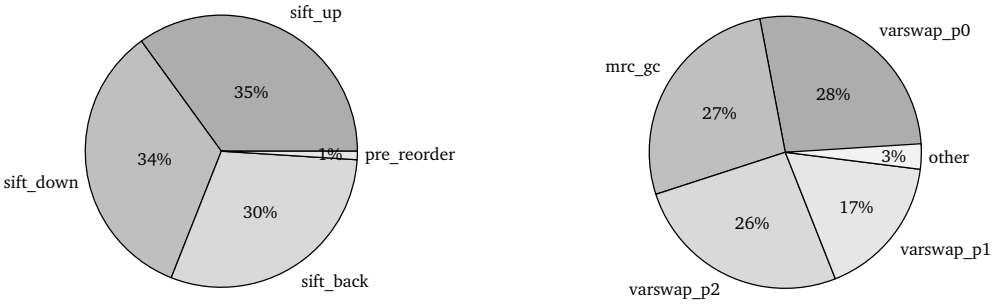
Figure 8.2: Manual reordering `mrc_gc` and `varswap_p0` profiles



(a) varswap_p1 (b) varswap_p2

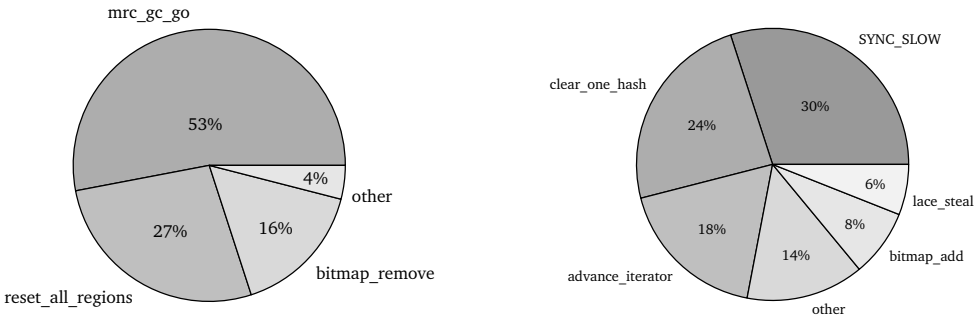
Figure 8.3: Manual reordering `varswap_p1` and `varswap_p2` profiles

to a higher number of invocations, and `advance_iterator` procedure dropped below 1% as expected due to a smaller number of nodes. Consider Figures 8.5b, 8.6a, and 8.6b, similarly as with manual reordering, `clear_one_hash`, `rehash_bucket`, and `makenode` take a significant part of the runtime which is expected.



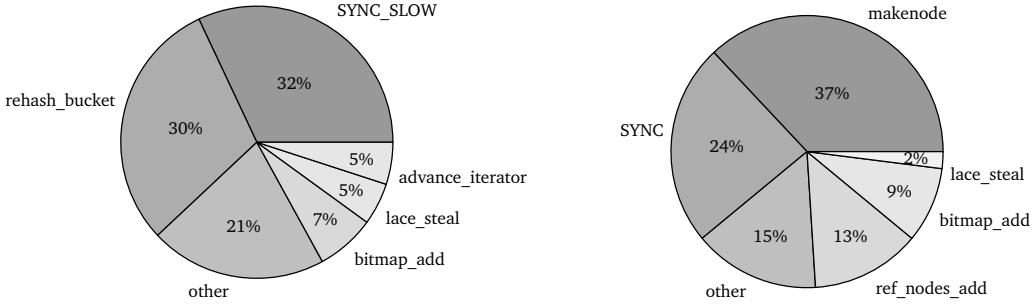
(a) reduce_heap (b) varswap

Figure 8.4: Semi-automatic reordering `reduce_heap` and `varswap` profiles



(a) mrc_gc (b) varswap_p0

Figure 8.5: Semi-automatic reordering `mrc_gc` and `varswap_p0` profiles



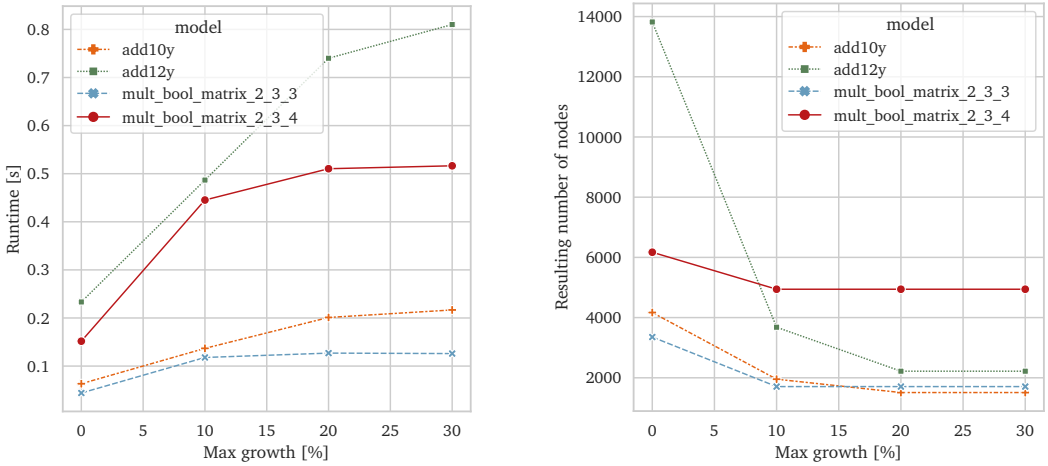
(a) varswap_p1 (b) varswap_p2

Figure 8.6: Semi-automatic reordering `varswap_p1` and `varswap_p2` profiles

8.3 Regression tests

The results of the dynamic reordering parameters tuning, such as the *max growth*, *nodes threshold*, and `TASK_SIZE` are presented in this section. Both *max growth* and *nodes threshold* impact runtime and quality of the reordering. The `TASK_SIZE` parameter impacts only the runtime. Finally, we test how the dynamic variable reordering scales with an increased number of Lace workers.

Max growth In Figure 8.7, the results of runtime and quality tests of the *max growth* tuning are depicted. We observe in Figure 8.7a, that with an increased maximum growth percentage, the runtime execution also increases. However, the rate of change slows down with increasing maximum growth. The biggest impact on runtime is between 0 to 10%. In Figure 8.7b, the reordering quality is improved the most in between 0 to 10%. Therefore, the runtime is most negatively impacted between 0 to 10% since reordering needs to perform more variable swaps to reduce the size. We also observe that increasing max growth beyond 20% does not provide further quality improvements and only increases the runtime. Setting the *max growth* below 10% reduces quality on average by more than 50% which will negatively impact the runtime of the consecutive reordering. Thus, the *max growth* parameter seems optimal between 10 to 20% depending on whether the priority is runtime or quality, respectively.

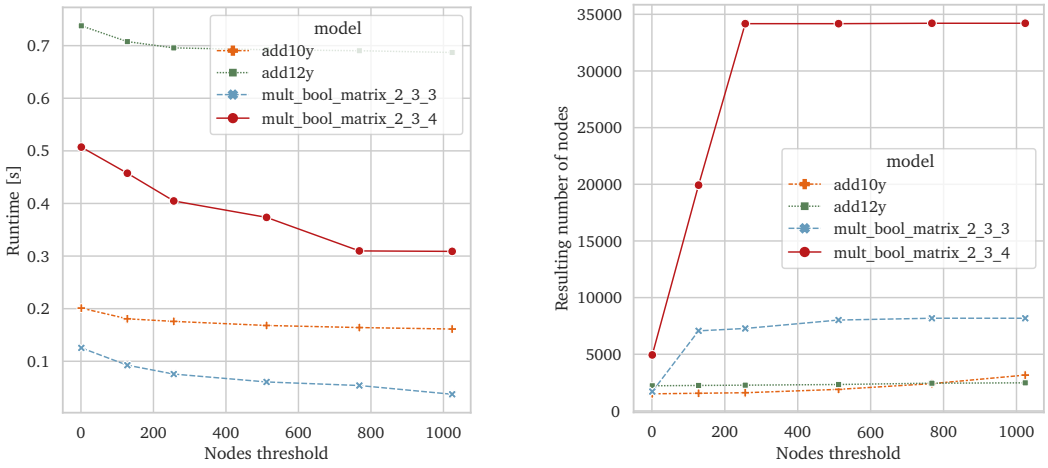


(a) Max growth runtime tests

(b) Max growth quality tests

Figure 8.7: Max growth tuning

Nodes threshold The *nodes threshold* tuning restuls are depicted in Figure 8.8. Figure 8.8a shows that increasing the threshold positively impacts the runtime. It is expected since more variables which do not satisfy the threshold are skipped. However, in Figure 8.8b, the quality effect differs per model characteristics. The difference between the models is that the add model contains fewer variables with more nodes, and the matrix multiplication model contains more variables with fewer nodes per variable. This implies that setting the exact values is use-case specific. Nevertheless, the threshold values between 0 and 128 provide an average runtime speedup of 0 to 10% with no quality trade-off in the specified use case.



(a) Nodes threshold runtime tests

(b) Nodes threshold quality tests

Figure 8.8: Nodes threshold tuning

Task size To understand the impact of tuning parameter `TASK_SIZE`, we present a regression test comparing several task sizes and identifying the most optimal value. In Figure 8.9, we present the results of an experiment in which three safety synthesis models are used, namely `add10y.aag`, `add12y.aag`, `add14y.aag` with the 38133, 169619, and 744993 nodes, respectively. Moreover, the manual reordering was used to load the nodes at once. We test the parallelization of Algothim 4 since the remaining reordering algorithms are executed sequentially. The task size determines the number of indices processed by each task. However, the workload distribution is not even since the nodes inserted in the unique table might be spread over the entire table. We observe that tasks with size 128 perform the worst, followed by 512. However, tasks with size 1024 performed better than tasks with size 4096. When the task size is too small, it has a negative impact on the runtime since many roaring bitmaps have to be created, which is the case with the sizes 128 and 512. On the other hand, tasks with sizes beyond 1024 do not seem to improve the runtime. Consequently, they increase the `SYNC_SLOW` runtime, which means there is a lack of tasks as discussed in Section 8.2. Hence, the most optimal value for the `TASK_SIZE` tuning parameter is 1024.

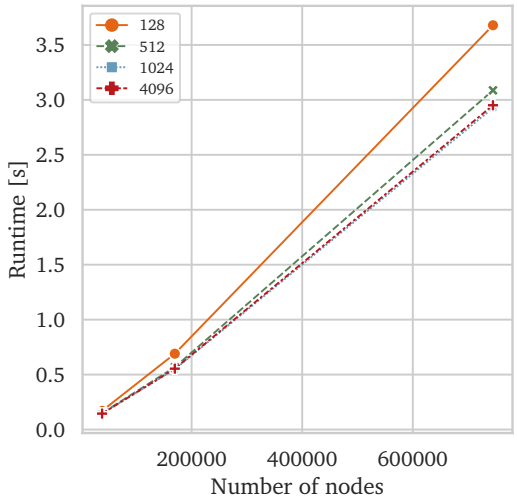


Figure 8.9: Task size runtime impact

Number of workers Figure 8.10 depicts the number of workers runtime impact with 1 to 16 Lace workers. Considering we run varswap in parallel and the other algorithms such as sift_up, sift_down or sift_back sequentially, together with Amdahl's law, we see expected results. Amdahl's Law states that there is a theoretical upper bound on the speed up of a parallel execution bounded by the part of the program which runs sequentially[2]. Hence, even with an infinite number of processors, there is an upper bound on speed up for algorithms which contain sequential executions. The portion of the sequential execution for the dynamic reordering depends on the use case and a test machine. With bigger BDDs single swap takes a bigger portion of the algorithm, consequently improving the speed up. Therefore, the bigger the model is, the bigger

the speed up. However, we observe that the peak is reached on average with 8 workers, and an increasing number of workers negatively impacts the runtime. We provide two hypotheses, mutually not exclusive, on why the speed up decreases after a certain number of Lace workers:

- The degree of parallelization depends on the task size; the smaller the task size, the higher the degree of parallelization since we can create more tasks for the same job. If there is not enough tasks for a Lace worker, it steals a task from a victim. Since the tasks do not have an evenly distributed workload, it might be that the victim has to steal a task back from the thief, which is called leapfrogging. If this happens often, the runtime speed up is decreased. Thus, decreasing the task size will provide more tasks for each worker and should improve the parallelization speed up. However, in the task size test, we saw that the task itself has a creation cost due to creating a roaring bitmap to iterate over the unique table nodes efficiently. Thus, reducing the task creation cost positively impacts the speed up only until the task size 1024.
- We observed in the reordering profiles that in the variable swap phase 2, significant runtime is spent on updating the node reference counters, which act as a hotspot. The increased number of workers increases the probability of accessing the same shared reference counter. This negatively affects the runtime. A solution could be to use the local reduction pattern, calculate the reference difference locally in each task, merge the results with its parent, and eventually update the shared resource in one go, or update the shared resource at the end of each task. However, this solution requires an efficient data structure that can hold sparse large indices, mapping them to their reference counter difference.

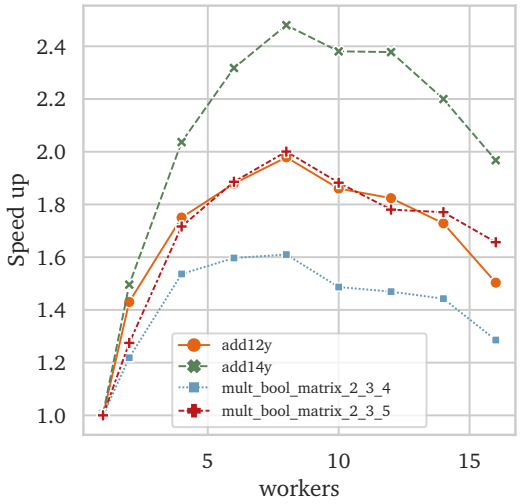


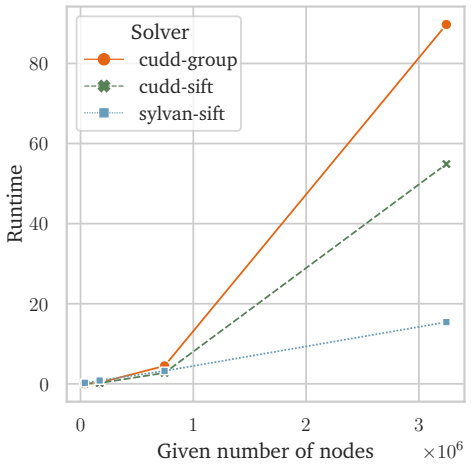
Figure 8.10: Number of workers speed up

8.4 Safety games

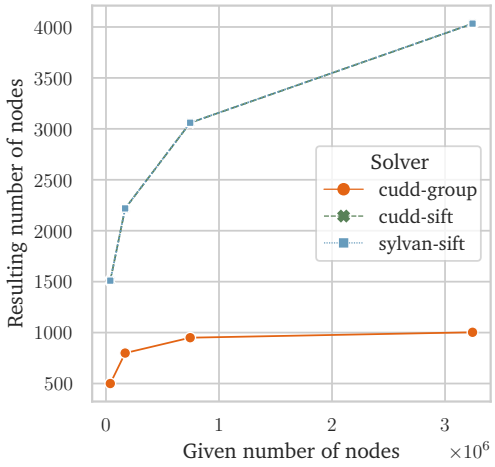
In this section, we compare the dynamic reordering in Sylvan with the state of the art BDD package CUDD. Firstly, we compare the Sylvan and CUDD BDD solvers by providing the same conditions. We trigger the reordering at the same locations of the code and disable CUDD automatic reordering to avoid interference. Then, we compare Sylvan and CUDD from the user perspective, meaning we compare the actual runtime of the entire application rather than just the runtime of the reordering.

Manual reordering Figures 8.11 and 8.12 show the results of reordering two safety game models namely, *add* the adder benchmarks and *mult_bool_matrix* the matrix multiplier benchmarks. For more information about *add*, and *mult_bool_matrix* models, we refer to [18] and [20], respectively. We tested the runtime of a single reordering execution on different BDD sizes after the AIG gates were loaded into the BDD packages. We let each BDD package load the complete model first and then reorder the variables. CUDD offers several reordering heuristics as described in Chapter 3. To make the comparison fair, we use Rudell's sifting with dynamic lower bounds as we provide in Sylvan, and we also added the Group sifting algorithm explained in Section 3.2, which was developed specifically for the CUDD package. We used 8 workers in Sylvan due to the results presented in the regression tests. We observe that CUDD is performant until approximately 10^6 nodes, after which Sylvan outperforms CUDD reordering runtime as seen in Table 8.1. The reordering quality of both Sylvan and CUDD Rudell's sifting is equal. The CUDD Group sifting yields better-reordering quality as seen in Figure 8.11, which means Rudell's sifting due to local minima trap resulted in more does. On the other hand, in Figure 8.12, we observe that the quality is slightly wors with the Group sifting heuristic compared to Rudell's sifting. In particular, Table 8.1 shows that the Group sifting reordering quality is worse by less than 1%. Lastly, we observe that the dynamic variable reordering reduces the number of nodes and consequently memory requirements by as high as 353 times with the *mult_bool_matrix_2_3_8* benchmark shown in Table 8.1.

Automatic reordering Besides manually triggering reordering, CUDD also provides automatic reordering. When the automatic reordering is enabled, it checks at every unique table lookup operation whether the table size has doubled compared to the previous reordering. If the reordering was not yet initiated, it requires at least 4000 nodes to start the reordering. The reordering status is checked before every operation, which allows to start reordering at any time. In the case of Sylvan, we require a user to invoke `test_reduce_heap` after a Sylvan operation completes. The reason is that the garbage collection might be started as the operation proceeds, during which reordering is not allowed. Therefore, starting reordering at arbitrary execution time is currently not supported. Figure 8.13 depicts the overall runtime benchmarks between semi-automatic Sylvan and automatic CUDD reorderings. We observe that CUDD outperforms Sylvan in every benchmark. Moreover, Group sifting yields even further runtime improvements with the *add* models compared to just CUDD sifting with dynamic lower bounds. The runtime superiority is likely due to two factors. Firstly, we observed with the manual reordering that Sylvan starts outperforming CUDD only with reordering above 10^6

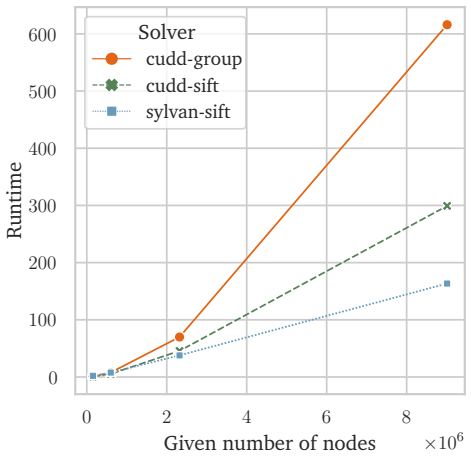


(a) Safety game add runtime benchmarks

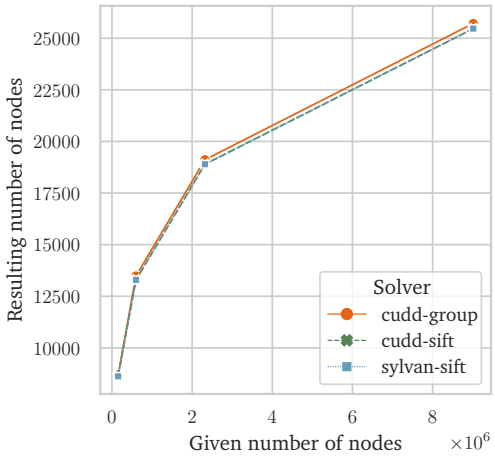


(b) Safety game add quality benchmarks

Figure 8.11: Sylvan and CUDD with the same reordering trigger points (add)



(a) Safety game matrix runtime benchmarks



(b) Safety game matrix quality benchmarks

Figure 8.12: Sylvan and CUDD with the same reordering trigger points (matrix)

nodes. With automatic reordering, the number of nodes is kept as low as possible, which is in advantage of CUDD. Secondly, thanks to the fully automatic reordering in CUDD, nodes can be reduced further compared to Sylvan, which only reorders after an operation completes. Consequently, CUDD runs reordering more times, yielding better runtime and quality than Sylvan. Lastly, we observe that the Sylvan semi-automatic reordering improves the execution runtime significantly. For instance, consider benchmark *add14y.aag* and Tables 8.2 and 8.1, with only one reordering per execution, the reordering alone took 3.2 seconds whereas, with the semi-automatic reordering, the entire execution including loading, reordering and solving took 0.5 seconds.

Model	Initial Size	Resulting Size	Reordering Time [s]	Solver
add10y.aag	38133	1510	0.247967	sylvan-sift
add10y.aag	38132	1509	0.030000	cudd-sift
add10y.aag	38132	500	0.040000	cudd-group
add12y.aag	169618	799	0.300000	cudd-group
add12y.aag	169618	2218	0.200000	cudd-sift
add12y.aag	169619	2219	0.822110	sylvan-sift
add14y.aag	744993	3060	3.222659	sylvan-sift
add14y.aag	744992	950	4.470000	cudd-group
add14y.aag	744992	3059	2.730000	cudd-sift
add16y.aag	3243423	4033	15.411469	sylvan-sift
add16y.aag	3243422	1003	89.720000	cudd-group
add16y.aag	3243422	4032	54.870000	cudd-sift
mult_bool_matrix_2_3_5.aag	154640	8622	0.770000	cudd-sift
mult_bool_matrix_2_3_5.aag	154640	8695	1.070000	cudd-group
mult_bool_matrix_2_3_5.aag	154641	8623	2.017540	sylvan-sift
mult_bool_matrix_2_3_6.aag	599360	13295	7.738878	sylvan-sift
mult_bool_matrix_2_3_6.aag	599359	13482	7.860000	cudd-group
mult_bool_matrix_2_3_6.aag	599359	13294	5.360000	cudd-sift
mult_bool_matrix_2_3_7.aag	2320047	18895	45.640000	cudd-sift
mult_bool_matrix_2_3_7.aag	2320048	18896	37.805228	sylvan-sift
mult_bool_matrix_2_3_7.aag	2320047	19115	69.900000	cudd-group
mult_bool_matrix_2_3_8.aag	9012706	25715	615.990000	cudd-group
mult_bool_matrix_2_3_8.aag	9012706	25463	299.180000	cudd-sift
mult_bool_matrix_2_3_8.aag	9012707	25464	163.407733	sylvan-sift

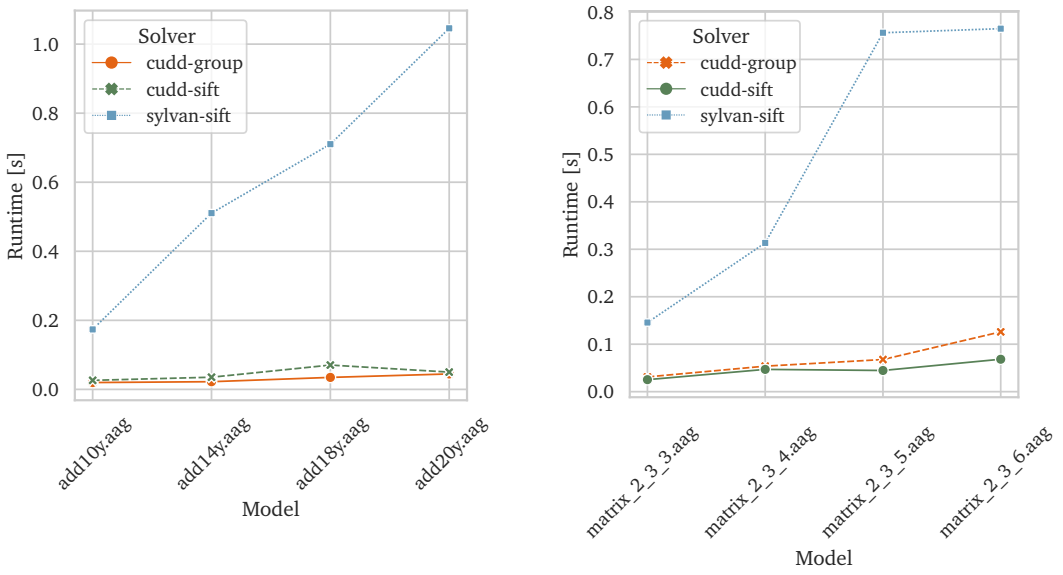
Table 8.1: Sylvan and CUDD manual benchmarks

8.5 Conclusions

The Sylvan dynamic variable reordering evaluation aimed to determine the reordering runtime and quality effects. The evaluation was split into three parts namely, profiling the reordering procedures, performing a regression test on the selected tuning parameters, and comparing the manual and semi-automatic reorderings with the state of art BDD package CUDD.

We observed that the Sylvan reordering procedure does not take full advantage of the parallelization due to a task cost introduced by creating separate roaring bitmaps for each task. Moreover, we observed that the unique table is traversed efficiently even with one table for the entire forest using the roaring bitmaps. This implies real-time performance could be obtained by decreasing task cost and consequently increasing the parallelization degree.

The tunning regression tests showed that the optimal *max growth* is between 10 to 20% depending on whether runtime or quality is a priority, respectively. The *nodes threshold* parameter seems optimal between 0 and 128, providing an average runtime



(a) Safety game add benchmarks

(b) Safety game matrix benchmarks

Figure 8.13: Sylvan semi-automatic and CUDD automatic reorderings

speedup of 0 to 10% with negligible negative quality impact in models containing minority variables with nodes below the threshold. The optimum parallel granularity for the `TASK_SIZE` seems to be with a task size of 1024. Further increase of the task sizes resulted in no further runtime improvements. The runtime speed-up peak was reached with 8 Lace workers, and further worker increase impacted the runtime only negatively. We also observed the parallelization speedup is increased with the model size, which is expected due to the increased variable swap parallel procedure portion.

In the manual reordering benchmarks, CUDD is performant until approximately 10^6 nodes, after which Sylvan outperforms CUDD. Moreover, the dynamic variable reordering reduces the number of nodes and consequently memory requirements by as high as 353 times in the presented benchmarks. Conversely, in the automatic reordering benchmarks, CUDD outperforms Sylvan. Moreover, Group sifting yields even further runtime improvements with certain models compared to just sifting with dynamic lower bounds. Lastly, observed that the Sylvan semi-automatic reordering improves the execution runtime significantly. In some cases, with only one reordering per execution, the reordering alone took 3.2 seconds, whereas with the semi-automatic reordering, the entire execution, including loading, reordering and solving, took 0.5 seconds.

In general, Sylvan reordering is effective in reducing runtime and memory consumption. The reordering takes advantage of the parallelization, using which Sylvan outperformed state of the art CUDD package in reorderings beyond a 10^6 number of nodes. Furthermore, the significance of the reordering quality implies some models larger than the available memory could be loaded using the reordering, which could not be otherwise.

Model	stddev	Runtime [s]	min	max	Solver
add10y.aag	0.001438	0.019986	0.018080	0.021609	cudd-group
add10y.aag	0.001397	0.026103	0.023695	0.027994	cudd-sift
add10y.aag	0.004455	0.172905	0.168463	0.182778	sylvan-sift
add14y.aag	0.001190	0.022621	0.020128	0.023856	cudd-group
add14y.aag	0.001315	0.035290	0.032725	0.037109	cudd-sift
add14y.aag	0.007131	0.510578	0.498666	0.522277	sylvan-sift
add18y.aag	0.001290	0.034658	0.032446	0.036656	cudd-group
add18y.aag	0.002971	0.071830	0.065476	0.073264	cudd-sift
add18y.aag	0.012516	0.710484	0.692769	0.729251	sylvan-sift
add20y.aag	0.001253	0.050076	0.048315	0.051733	cudd-sift
add20y.aag	0.015232	1.052314	1.016005	1.059145	sylvan-sift
add20y.aag	0.001430	0.045475	0.042260	0.046292	cudd-group
mult_bool_matrix_2_3_3.aag	0.002120	0.025479	0.021961	0.028465	cudd-sift
mult_bool_matrix_2_3_3.aag	0.001227	0.031433	0.028372	0.032427	cudd-group
mult_bool_matrix_2_3_3.aag	0.004420	0.144638	0.139832	0.153181	sylvan-sift
mult_bool_matrix_2_3_4.aag	0.001460	0.047192	0.044608	0.048792	cudd-sift
mult_bool_matrix_2_3_4.aag	0.001114	0.053860	0.052163	0.055332	cudd-group
mult_bool_matrix_2_3_4.aag	0.009467	0.314767	0.298625	0.324041	sylvan-sift
mult_bool_matrix_2_3_5.aag	0.024882	0.746512	0.731614	0.801891	sylvan-sift
mult_bool_matrix_2_3_5.aag	0.002318	0.044529	0.039302	0.047248	cudd-sift
mult_bool_matrix_2_3_5.aag	0.003337	0.068252	0.060287	0.072121	cudd-group
mult_bool_matrix_2_3_6.aag	0.002976	0.069254	0.062656	0.070729	cudd-sift
mult_bool_matrix_2_3_6.aag	0.002629	0.126161	0.122164	0.130229	cudd-group
mult_bool_matrix_2_3_6.aag	0.017171	0.759130	0.749500	0.798918	sylvan-sift

Table 8.2: Sylvan semi-automatic and CUDD automatic benchmarks

Chapter 9

Reordering user guide

In this chapter, we will provide a basic guide from the Sylvan user perspective on how to employ variable reordering. Using dynamic variable reordering in existing projects might require certain work to be done. For instance, if the user depends on a certain variable order it will not anymore hold after a variable reordering is performed. However, if the user requires only a logical function of a certain variable then the variable reordering will not cause any influence. For users that require to keep track of the variables, we provide an API using which these changes can be tracked. When using reordering it is convenient to use levels and map levels to a certain variable. This way we make a clear distinction between specific variable order and logical function at some level. Sylvan provides the following public API to map levels to variable order and vice versa; `sylvan_level_to_order` and `sylvan_order_to_level`. By default Sylvan has dynamic variable reordering disabled which means calling any API related to reordering will not have any effect. To enable the variable rendering us the following API after the Sylvan package is initialized `sylvan_init_reorder`. When dynamic variable reordering is enabled, the configurations described in Section 7.4 can be set together with attaching the callbacks.

To invoke the reordering Sylvan API offers three functions each at a different depth; `sylvan_reorder_perm` which is for users that want to reorder the variables based on a given permutation, `sylvan_reduce_heap` which is for users that would like to invoke reordering with either with our without the dynamic lower bounds, and lastly `sylvan_test_reduce_heap` which is for users that need CUDD like automatic reordering. The function `sylvan_test_reduce_heap` will check whether reordering is needed and if so the default reordering type unless specified will be used to reorder the variables. If the reordering was not yet run, a minimum of 400 nodes is required to start the reordering. Otherwise, reordering will be started every time the number of nodes doubles compared to the last reordering.

As seen in Chapter 8, if performance is of the highest importance then calling `sylvan_test_reduce_heap` after every Sylvan operation is recommended. Also, in cases a model to be loaded using Sylvan is too large, it might be more optimal to use dynamic variable reordering since generally less memory will be required and Sylvan will also keep the unique table size as low as possible.

In case of running into issues, Sylvan defines an enum `reorder_result_t` which

hols currently has thirteen error codes describing errors during individual reordering stages. Also, Sylvan API provides a function `sylvan_reorder_resdescription` using which a given error code can be mapped to an error message so an error message can be printed instead of an error code for the user's convenience.

Chapter 10

Discussion

The results presented in Chapter 8 are promising. We observed that when using a larger number of nodes, Sylvan dynamic variable reordering keeps pace and even outperforms the state of art BDD package CUDD. Despite the strategical subtable design used in CUDD, Sylvan can iterate over all nodes using the roaring bitmaps efficiently and in parallel. However, we also observed that when automatic reordering is enabled in CUDD, Sylvan no longer performs comparably. On the other hand, we also saw that loading a large model using reordering reduces runtime and requirements on memory consumption by requiring less nodes. This implies that using variable reordering allows loading models into Sylvan, which would not be possible otherwise due to memory limitations. In this chapter, we discuss the possible improvements that could be made in Sylvan to improve dynamic variable reordering further, and we reflect on the methodology and the execution process.

10.1 Sifting parallelization

In section 8.3, we observed that the most optimal `TASK_SIZE` is 1024. Decreasing the `TASK_SIZE` parameter increased runtime. Moreover, we observed in Section 8.2 that `SYNC_SLOW` took a significant part of varswap, which could mean there is not enough tasks available for stealing. Hence, it's clear that the current design does not benefit fully from the possible parallelization, but increasing the number of tasks has a negative impact on the runtime. In terms of Lace, creating more tasks should not have more overhead. However, we use the parallel reduction pattern by which we create a Roaring bitmap for each task and then merge the results once the children tasks are finished. Thread-safe roaring bitmaps could be a solution. We discuss the roaring bitmaps and thread safety in more detail in the following section. The reason for using roaring bitmaps is to efficiently traverse the Sylvan unique table where all nodes are stored. Traversing nodes of a particular variable requires traversing the entire table, unlike in CUDD, where a separate suitable is created for each variable. Thus, another option could be to use the suitable design in Sylvan as well, which complicates the implementation of parallelism due to the region division. The remainder of the sifting algorithm must also have parallel implementation to take advantage of parallelism. Lastly, we also observed in Section 8.2, that mutating reference counters in varswap phase 2 tasks a

noticeable part of the runtime. A solution could be to use the local reduction pattern, calculate the reference difference locally in each task, merge the results with its parent, and eventually update the shared resource in one go. However, this solution requires an efficient data structure that can hold large sparse indices, mapping them to their reference counter difference.

10.2 Roaring bitmaps and thread-safety

The CRoaring bitmap implementation used in this thesis is not thread-safe. Therefore, we faced the challenge of using the roaring bitmaps with multiple Lace workers in a thread-safe manner. A solution to overcome this challenge is to use the local reduction pattern to create a local roaring bitmap for each task so that each worker can mutate its bitmap safely. Then, the bitmap is recursively merged with its parent bitmap so that the root task can merge the results with the desired bitmap. However, this creates a cost for task creation. One option to avoid creating local roaring bitmaps would be using our thread-safe implementation of the regular bitmaps. We observed that the runtime of traversing regular bitmaps impacts the runtime more negatively than using a local copy of the roaring bitmap for each worker. Another solution could be to use thread-safe, perhaps lock-free roaring bitmaps, which could be shared with all workers. To avoid hotspots by providing the same shared resources to all workers, one could use the roaring bitmap containers where the data is stored and distribute them among the workers. Depending on the data characteristics, such as the number of elements or how sparse the data is, the respective number of containers is created. This approach would also improve the fact that the task workload is currently not evenly distributed since we only split the tasks into ranges of indices from zero to the maximum table size. In practice, the indices can be distributed ununiformly across the entire table. Lastly, Sylvan uses regular bitmaps to store the ownership data and the indices of nodes in the unique table. The memory and runtime could be improved by using the above-described thread-safe roaring bitmaps instead.

10.3 Swapping non-interacting variables

When two variables do not interact the variable swap can become an inexpensive operation if we can swap only the mappings and avoid modifying the unique table altogether. The inexpensive swaps in CUDD are handled by only swapping the subtables. Sylvan does not have subtables, and swapping only the mappings will not be enough since the rest of Sylvan relies directly on reading the variable label from the internal mtbddnode. To implement the inexpensive swap in Sylvan, we need to ensure that the variable label will be correctly mapped to the corresponding variable at every place. For instance, we would need to use mappings in every Sylvan operation. The performance gain from just swapping the mappings is that we can avoid all phases of the swap including the garbage collection, since no nodes will or will become dead. However, the practical performance gain depends on a use case, for instance, the safety game benchmarks use many interacting variables and hence we did not see a noticeable impact of CUDD inexpensive swaps. The difference will be much more significant for other use cases when

there are many non-interacting variables. Hence, implementing inexpensive swaps in Sylvan is a significant performance improvement for specific use cases.

10.4 Automatic reordering

The most significant difference when comparing dynamic variable reordering in Sylvan and CUDD was obtained when automatic reordering was enabled in CUDD. As a consequence, CUDD was able to outperform Sylvan in every benchmark. The reason is that when automatic reordering is enabled in CUDD, before every unique table lookup operation, a check is done on whether reordering is necessary. This means the variables are reordered more often than compared to Sylvan. This implies CUDD can avoid the local minima trap more effectively, which is a disadvantage of Rudell's sifting algorithm. To implement similar automatic reordering in Sylvan, we need to handle cooperation with garbage collection. During a Sylvan operation, garbage collection might be triggered. Sylvan distinguishes two phases namely, normal operation and garbage collection. However, we would require a third phase during which the reordering would be performed. Therefore, running dynamic variable reordering before every unique table lookup operation is not supported. Running dynamic reordering more often yields smaller BDDS and improves runtime. Hence, supporting variable reordering before every unique table lookup operation might provide the most significant improvements among all suggested improvements for automatic dynamic reordering benchmarks.

10.5 Reordering heuristics

Rudell's sifting algorithm sifts one variable at a time, given all other variables are fixed. As a consequence, the algorithm might be trapped in local minima and never reach the global minima. Moreover, the smaller the BDD size is, the faster the variable rendering can be performed. Due to these reasons, Somenzi et al. implemented a sifting algorithm for CUDD called group sifting. Group sifting helps to avoid local minima traps by sifting a group of variables identified by symmetry rather than just sifting a single variable. Indeed, the heuristic comparison presented in Section 8.4shows that CUDD group sifting is more performant than Rudell's sifting by reducing BDD to a smaller size and consecutively increasing the runtime. Hence, implementing group sifting in Sylvan would result in improved quality of reordering as well as improved runtime.

10.6 Methodology

To reach the research objectives in a structured manner, the methodology described in Chapter 4 was defined. It contains four standalone work packages with defined Tasks ordered chronologically, using which the planning was prepared. Using the planning, thesis progress could be tracked together with the expected outcomes of each work package. However, along the process, we found that several parts of the reordering need to be improved to make Sylvan comparable to CUDD with respect to runtime.

This implied additional tasks had to be added, such as implementing the reference-based garbage collector described in Chapter 5 or the roaring bitmap traversal described in Chapter 6. The planning had to be adjusted, and priorities were shifted. As a result, originally planned Weighted Model Counting (WMC) reordering benchmarks were not included in this thesis due to time limitations. DPMC solver[1] that solves WMC and supports both CUDD and Sylvan provides a different type of benchmark where many small BDDs are created during the process. The options were to either skip further reordering improvements and continue with the work plan, which would have a significant negative impact on the reordering runtime or to shift the priorities. Therefore the priorities were shifted. Consequently, the remaining allocated time for the evaluation appeared to be not sufficiently large for all desired experiments. Thus, including WMC benchmarks and additional synthetic benchmarks could provide deeper insight into regression tests and the comparison between Sylvan and CUDD.

[1] https://github.com/apdofficial/sylvan-benchmarks

Chapter 11

Conclusions

We have researched, implemented and evaluated the dynamic variable reordering in the Sylvan BDD package. The two primary benefits of dynamic variable reordering are reduced runtime and reduced memory consumption. Based on the evaluation, we observed that Sylvan reordering is effective in reducing runtime and memory consumption. The reordering takes advantage of the parallelization, using which Sylvan outperforms the state of the art CUDD package in reorderings with more than 1 000 000 nodes. Furthermore, the significance of the reordering quality implies some models larger than the available computer memory could be loaded using the reordering which could not be otherwise. The answers to the research questions are as follows:

RQ1: *How to implement dynamic variable reordering using the sifting algorithm in Sylvan?*

Sylvan uses one unique table for all nodes, making it asymptotically bounded to the size of the unique table when traversing nodes of a variable. To make the traversal efficient, roaring bitmap can be used as an efficient iterator to traverse the nodes with the random access time complexity $O(log(n))$. Moreover, we observed that mark-and-sweep is not optimal as a garbage collection mechanism since it requires traversing the entire unique table every time and does not allow selective deletion. Instead, we introduced reference-based garbage collection which can delete the elements as soon as the reference counter for any given element reaches zero. Lastly, we used a parallel reduction pattern and parallelized the adjacent variable swap which takes advantage of the task based parallelism in Sylvan. Roaring bitmaps, reference-based garbage collection, and the sifting algorithm with parallel variable swap allow efficient implementation of dynamic variable reordering in Sylvan.

RQ2: *How to tune different parameters to maximize the performance of the dynamic variable reordering in Sylvan?*

The tunning regression tests showed that the optimal *max growth* is between 10 to 20% depending on whether runtime or quality is a priority, respectively. The *nodes threshold* parameter seems optimal between 0 and 128, providing an average runtime speedup of 0 to 10% with negligible negative quality impact in models containing minority variables with nodes below the threshold. The optimum parallel granularity

for the `TASK_SIZE` seems to be with a task size of 1024. Further increase in the task sizes resulted in no further runtime improvements. The runtime speed-up peak was reached with 8 Lace workers, and further worker increase impacted the runtime only negatively. Lastly, we also observed that the parallelization speedup is increased with the model size, which is expected due to the increased variable swap parallel procedure portion.

RQ3: *How can a fair comparison of the dynamic variable reordering performance be made between Sylvan and CUDD?*

By using the same input models and benchmark solvers for both Sylvan and CUDD and triggering the same reordering heuristics at the same execution point with the same BDDs, the comparison can be made fair. In the manual reordering benchmarks, CUDD is performant until approximately 1 000 000 nodes, after which Sylvan outperforms CUDD. Moreover, the dynamic variable reordering reduces the number of nodes and, consequently, memory requirements by as high as 353 times in the presented benchmarks. Conversely, in the automatic reordering benchmarks, CUDD outperforms Sylvan. Moreover, the CUDD Group sifting yields even further runtime improvements with certain models compared to just sifting with dynamic lower bounds. Lastly, we observed that the Sylvan semi-automatic reordering improves the execution runtime significantly. In some cases, with only one reordering per execution, the reordering alone took 3.2 seconds, whereas with the semi-automatic reordering, the entire execution, including loading, reordering and solving, took 0.5 seconds.

RQ4: *How does the Sylvan hash map with chaining collision avoidance affect the performance compared to linear probing collision avoidance w.r.t dynamic variable ordering?*

Based on the comparison of the Sylvan hash map with chaining and probing, we conclude that no significant overhead is introduced when using the chaining. However, it is clear that chaining introduces a certain level of overhead when approximately less than 60% of the table is occupied. Beyond 60%, chaining outperforms linear probing. On the other hand, the current Sylvan hash map implementation does not allow single-item removal, and an additional approach using tombstones would need to be implemented. However, when using tombstones, over time, all empty buckets become tombstones which is undesired. On the other hand, the chaining implementation allows a single element removal and together with the reference garbage collection, they provide an efficient way to remove nodes selectively. This has been observed to be important for making the adjacent variable swap efficient. Hence, when using dynamic variable reordering, Sylvan hash map implementation with chaining is advised.

Finally, considering the above conclusions, the answer to the central research question is as follows:

How to maximize the performance of the dynamic variable reordering for binary decision diagrams in the Sylvan BDD package?

Using a hash map with chaining collision avoidance and reference garbage collection provides an efficient way to delete items in Sylvan selectively. Furthermore, roaring bitmaps provide an efficient way to traverse unique table in Sylvan. Efficient deletion of individual items together with efficient unique table traversal is necessary for maximizing the performance of the adjacent variable swap, which is a core operation used during Rudell's sifting algorithm. The runtime can be further improved by taking advantage of take-based parallelism in the adjacent variable swap. Using the dynamic lower bounds, the reordering stops early when no further improvements can be made without negatively impacting the quality. Lastly, by controlling parameters such as maximum growth and nodes threshold, the reordering can be steered towards improved reordering runtime or quality.

The direction for future work on the dynamic variable reordering in Sylvan was given in several directions namely, sifting parallelization, thread-safe roaring bitmaps, improving non-interacting swaps, implementing automatic reordering, implementing more elaborate reordering heuristics such as group sifting and proposing additional benchmarks such as the Weighted Model Counting.

Bibliography

[1] Akers. "Binary Decision Diagrams". In: *IEEE Transactions on Computers* C-27.6 (1978), pp. 509–516. DOI: 10.1109/TC.1978.1675141.

[2] Gene Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20)". In: *Solid-State Circuits Newsletter, IEEE* 12 (Feb. 2007), pp. 19 –20. DOI: 10.1109/N-SSC.2007.4785615.

[3] *Atomic operations library - cppreference.com*. URL: https://en.cppreference.com/w/c/atomic.

[4] Beate Bollig and Ingo Wegener. "Improving the variable ordering of OBDDs is NP-complete". In: *IEEE Transactions on Computers* 45 (9 1996), pp. 993–1002. ISSN: 00189340. DOI: 10.1109/12.537122.

[5] Romain Brenguier et al. "Compositional Algorithms for Succinct Safety Games". In: *Electronic Proceedings in Theoretical Computer Science, EPTCS* 202 (Feb. 2016), pp. 98–111. DOI: 10.4204/EPTCS.202.7. URL: http://arxiv.org/abs/1602.01174http://dx.doi.org/10.4204/EPTCS.202.7.

[6] Randal E. Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams". In: *ACM Computing Surveys (CSUR)* 24 (3 Sept. 1992), pp. 293–318. ISSN: 15577341. DOI: 10.1145/136035.136043. URL: https://dl-acm-org.ezproxy2.utwente.nl/doi/10.1145/136035.136043.

[7] Sagar Chaki and Arie Gurfinkel. "BDD-based symbolic model checking". In: *Handbook of Model Checking* (May 2018), pp. 219–245. DOI: 10.1007/978-3-319-10575-8_8/COVER. URL: https://link-springer-com.ezproxy2.utwente.nl/chapter/10.1007/978-3-319-10575-8_8.

[8] Tom van Dijk. "The parallelization of binary decision diagram operations for model checking". In: (2012). URL: http://essay.utwente.nl/61650/.

[9] Tom van Dijk and Jaco C. van de Pol. "Lace: non-blocking split deque for work-stealing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8806 (2014), pp. 206–217. ISSN: 16113349. DOI: 10.1007/978-3-319-14313-2_18/COVER. URL: https://link-springer-com.ezproxy2.utwente.nl/chapter/10.1007/978-3-319-14313-2_18.

[10] Tom van Dijk et al. “A comparative study of BDD packages for probabilistic symbolic model checking”. In: *Lecture Notes in Computer Science* 9409 (2015), pp. 35–51. ISSN: 16113349. DOI: 10.1007/978-3-319-25942-0_3/COVER. URL: https://link-springer-com.ezproxy2.utwente.nl/chapter/10.1007/978-3-319-25942-0_3.

[11] Tom Van Dijk, Alfons Laarman, and Jaco Van De Pol. “Multi-Core BDD Operations for Symbolic Reachability”. In: *Electronic Notes in Theoretical Computer Science* 296 (Aug. 2013), pp. 127–143. ISSN: 1571-0661. DOI: 10.1016/J.ENTCS.2013.07.009.

[12] Tom Van Dijk and Jaco Van De Pol. “Sylvan: Multi-core Decision Diagrams”. In: *Lecture Notes in Computer Science* (2015).

[13] Rolf Drechsler and Wolfgang Glint. “Using Lower Bounds during Dynamic BDD Minimization”. In: (2 2000).

[14] Jeffrey M. Dudek, Vu H.N. Phan, and Moshe Y. Vardi. “ADDMC: Weighted Model Counting with Algebraic Decision Diagrams”. In: *Proceedings of the AAAI Conference* 34 (02 Apr. 2020), pp. 1468–1476. ISSN: 2374-3468. DOI: 10.1609/AAAI.V34I02.5505. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5505.

[15] Rüdiger Ebendt and Rolf Drechsler. “Effect of improved lower bounds in dynamic BDD reordering”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25 (5 May 2006), pp. 902–908. ISSN: 02780070. DOI: 10.1109/TCAD.2005.854632.

[16] Institute for Formal Models and Verification. *AIGER*. URL: https://fmv.jku.at/aiger/.

[17] Justin E. Harlow and Franc Brglez. “Design of experiments and evaluation of BDD ordering heuristics”. In: *International Journal on Software Tools for Technology Transfer* 3 (2 2001), pp. 193–206. ISSN: 14332779. DOI: 10.1007/S100090100052/METRICS. URL: https://link-springer-com.ezproxy2.utwente.nl/article/10.1007/s100090100052.

[18] Swen Jacobs et al. “The First Reactive Synthesis Competition (SYNTCOMP 2014)”. In: (2014).

[19] Swen Jacobs et al. “The Reactive Synthesis Competition (SYNTCOMP): 2018-2021”. In: (June 2022). DOI: 10.48550/arxiv.2206.00251. URL: https://arxiv-org.ezproxy2.utwente.nl/abs/2206.00251v1.

[20] Swen Jacobs et al. “The Second Reactive Synthesis Competition (SYNTCOMP 2015)”. In: (2015), pp. 27–57. DOI: 10.4204/EPTCS.202.4.

[21] Chuan Jiang et al. “Variable Reordering in Binary Decision Diagrams”. In: *26th International Workshop on Logic Synthesis* (Jan. 2017).

[22] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. “Consistently faster and smaller compressed bitmaps with Roaring”. In: (2016). DOI: 10.1002/spe.2402. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.2402.

[23] Daniel Lemire et al. “Roaring Bitmaps: Implementation of an Optimized Software Library”. In: (2022).

[24] Yukio Miyasaka, Alan Mishchenko, and Masahiro Fujita. “A Simple BDD Package without Variable Reordering and Its Application to Logic Optimization with Permissible Functions”. In: (2016).

[25] Dirk Moller’, Paul Molitor, and Rolf Drechsler’. “Symmetry Based Variable Ordering for ROBDDs”. In: *IFIP Advances in Information and Communication Technology* (1995), pp. 70–81. DOI: 10.1007/978-0-387-34920-6_7. URL: https://link-springer-com.ezproxy2.utwente.nl/chapter/10.1007/978-0-387-34920-6_7.

[26] Shipra Panda and Fabio Somenzi. “Who are the variables in your neighborhood”. In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers* (1995), pp. 74–77. ISSN: 10923152. DOI: 10.1109/ICCAD.1995.479994.

[27] Shipra Panda, Fabio Somenzi, and Bernard F. Plessier. “Symmetry detection and dynamic variable ordering of decision diagrams”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (1994), pp. 628–631. ISSN: 02780070.

[28] R. Rudell. “Dynamic variable ordering for ordered binary decision diagrams”. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)* (1993), pp. 42–47. DOI: 10.1109/ICCAD.1993.580029. URL: http://ieeexplore.ieee.org/document/580029/.

[29] Ellen M. Sentovich. “A brief study of BDD package performance”. In: *Lecture Notes in Computer Science* 1166 (1996), pp. 389–403. ISSN: 16113349. DOI: 10.1007/BFB0031823/COVER. URL: https://link-springer-com.ezproxy2.utwente.nl/chapter/10.1007/BFb0031823.

[30] Claude E. Shannon. “A symbolic analysis of relay and switching circuits”. In: *Electrical Engineering* 57 (12 July 2013), pp. 713–723. ISSN: 0095-9197. DOI: 10.1109/EE.1938.6431064.

[31] Guoyong Shi. “A survey on binary decision diagram approaches to symbolic analysis of analog integrated circuits”. In: *Analog Integrated Circuits and Signal Processing* 74 (2 Feb. 2013), pp. 331–343. ISSN: 09251030. DOI: 10.1007/S10470-011-9773-8/TABLES/2. URL: https://link-springer-com.ezproxy2.utwente.nl/article/10.1007/s10470-011-9773-8.

[32] Fabio Somenzi. *CUDD: CU Decision Diagram Package Release 2.4.1*. 2015. URL: http://web.mit.edu.ezproxy2.utwente.nl/sage/export/tmp/y/usr/share/doc/polybori/cudd/cuddIntro.html.

[33] Fabio Somenzi. “Efficient manipulation of decision diagrams”. In: *International Journal on Software Tools for Technology Transfer* 3 (2 2001), pp. 171–181. ISSN: 14332779. DOI: 10.1007/S100090100042.

[34] *The Reactive Synthesis Competition | www.syntcomp.org*. 2023. URL: http://www.syntcomp.org/.

[35] *The Reactive Synthesis Competition – Adam Walker –*. URL: https://adamwalker.github.io/The-Reactive-Synthesis-Competition/.

[36] Tom van Dijk. "Sylvan: multi-core decision diagrams". English. PhD thesis. University of Twente, July 2016. ISBN: 978-90-365-4160-2. DOI: 10.3990/1.9789036541602.

[37] Pengcheng Zhang, Henry Muccini, and Bixin Li. "A classification and comparison of model checking software architecture techniques". In: *Journal of Systems and Software* 83 (5 May 2010), pp. 723–744. ISSN: 0164-1212. DOI: 10.1016/J.JSS.2009.11.709.