sylvan

MULTI-CORE DECISION DIAGRAMS

TOM VAN DIJK

Sylvan: Multi-core Decision Diagrams

Tom van Dijk

Graduation committee:

Chairman:	prof.dr. P.M.G. Apers
Promotor:	prof.dr. J.C. van de Pol
Members:	
prof.dr.ir. J.P. Katoen	University of Twente
prof.dr.ir. B.R.H.M. Haverkort	University of Twente
prof.dr. G. Ciardo	Iowa State University, USA
prof.dr.ir. J.F. Groote	Eindhoven University of Technology
Referee:	
dr. Y. Thierry-Mieg	Laboratoire d'Informatique de Paris 6, France

CTIT Ph.D. Thesis Series No. 16–398
Centre for Telematics and Information Technology
University of Twente, The Netherlands
P.O. Box 217 – 7500 AE Enschede

IPA Dissertation Series No. 2016–09
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN: 978–90–365–4160–2
ISSN: 1381–3617 (CTIT Ph.D. Thesis Series No. 16–398)
Available online at `http://dx.doi.org/10.3990/1.9789036541602`

Typeset with LaTeX
Printed by Ipskamp Printers Enschede
Cover design by Annelien Dam

SYLVAN: MULTI-CORE DECISION DIAGRAMS

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof.dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Wednesday, July 13th, 2016 at 16:45 hrs.

by

Tom van Dijk

born on September 10th, 1985
in Emmen, The Netherlands

This dissertation has been approved by:

Prof.dr. J.C. van de Pol (promotor)

Contents

Contents **v**

Acknowledgements **ix**

1 Introduction **1**
1.1 Symbolic model checking . . . 2
1.2 Binary decision diagrams . . . 4
1.3 Parallelism . . . 5
1.4 Earlier work in parallel BDDs . . . 9
1.5 Contributions . . . 10
1.5.1 Scalable hash tables with garbage collection . . . 11
1.5.2 Work-stealing framework Lace . . . 12
1.5.3 Multi-core decision diagram library Sylvan . . . 12
1.5.4 Multi-core on-the-fly reachability . . . 13
1.5.5 Multi-core symbolic bisimulation minimisation . . . 14
1.6 Publications . . . 14
1.7 Overview . . . 17

2 Decision diagrams **19**
2.1 Preliminaries . . . 19
2.1.1 Boolean logic and notation . . . 19
2.1.2 Binary decision diagrams . . . 20
2.1.3 Multi-terminal binary decision diagrams . . . 22
2.1.4 Multi-valued decision diagrams . . . 23
2.1.5 List decision diagrams . . . 24
2.2 Parallelizing decision diagrams . . . 26
2.2.1 Parallel operations . . . 26
2.2.2 Representation of nodes . . . 28
2.2.3 Unique table . . . 29
2.2.4 Computed table . . . 30

2.2.5 Garbage collection framework . . . 31
2.3 BDD algorithms . . . 34
2.3.1 Creating and reading BDD nodes . . . 34
2.3.2 Basic operations . . . 35
2.3.3 Relational products . . . 38
2.4 MTBDD algorithms . . . 40
2.5 LDD algorithms . . . 42

3 Load-balancing tasks with Lace 45
3.1 Task-based parallelism and work-stealing . . . 46
3.2 Existing work-stealing deques . . . 50
3.3 Design of the shared split deque . . . 52
3.4 Deque algorithms . . . 56
3.5 Correctness . . . 59
3.6 Implementation of the framework Lace . . . 62
3.6.1 Standard work-stealing functionality . . . 62
3.6.2 Interrupting tasks to run a new task tree . . . 63
3.7 Experimental evaluation . . . 64
3.7.1 Benchmarks . . . 65
3.7.2 Results . . . 66
3.7.3 Extending leapfrogging . . . 67
3.8 Conclusion and Discussion . . . 69

4 Concurrent nodes table and operation cache 73
4.1 Scalable data structures . . . 73
4.2 Unique table . . . 75
4.2.1 Original hash table . . . 77
4.2.2 Variant 1: Reference counter and tombstones . . . 81
4.2.3 Variant 2: Independent locations . . . 83
4.2.4 Variant 3: Using bit arrays to manage the data array . . . 87
4.2.5 Comparing the three variants . . . 90
4.3 Operation cache . . . 92
4.4 Conclusion and Discussion . . . 94

5 Application: State space exploration 97
5.1 On-the-fly state space exploration in LTSmin . . . 98
5.2 Parallel operations in a sequential algorithm . . . 101
5.3 Parallel learning . . . 101
5.4 Fully parallel on-the-fly symbolic reachability . . . 103
5.5 Experimental evaluation . . . 104
5.5.1 Experimental setup . . . 104
5.5.2 Experiment 1: Only parallel LDD operations . . . 105

5.5.3 Experiment 2: Parallel learning . . . 107
5.5.4 Experiment 3: Fully parallel reachability . . . 108
5.5.5 Experiment 4: Comparing nodes table variants 2 and 3 . . 110
5.5.6 Experiment 5: Comparing BDDs and LDDs . . . 112
5.6 Conclusion and Discussion . . . 113

6 Application: Bisimulation minimisation 117
6.1 Definitions . . . 119
6.2 Signature-based bisimulation minimisation . . . 121
6.2.1 Partition refinement . . . 122
6.3 Symbolic signature refinement . . . 123
6.3.1 Encoding of signature refinement . . . 123
6.3.2 The `refine` algorithm . . . 125
6.3.3 Computing inert transitions . . . 128
6.4 Implementation . . . 129
6.5 Experimental evaluation . . . 130
6.5.1 Experiments . . . 130
6.5.2 Results . . . 131
6.6 Conclusion and Discussion . . . 131

7 Conclusions 135
7.1 The multi-core decision diagram package Sylvan . . . 135
7.2 The work-stealing framework Lace . . . 136
7.3 The symbolic bisimulation minimisation tool SigrefMC . . . 136
7.4 Future directions . . . 137
7.4.1 Scalable data structures . . . 137
7.4.2 Other decision diagrams and operations . . . 137
7.4.3 Applications . . . 138
7.4.4 Formal verification of the algorithms . . . 139
7.5 Multi-core and beyond . . . 139

Bibliography 141

Summary 151

Samenvatting 153

Acknowledgements

More or less five years ago, during a lecture on model checking, the lecturer prof.dr. Jaco van de Pol casually mentioned that parallelizing the "apply" operation for binary decision diagrams was still an open problem. Since I entertained the opinion that in principle it should be possible to execute every sufficiently large problem in parallel, this seemed an interesting challenge to tackle. Several weeks later I had to admit that there was a tiny little detail to which I didn't see an obvious solution. Jaco pointed out some earlier research by Alfons Laarman, related to a scalable shared hash table, which could be adapted for the implementation of operations on binary decision diagrams. Combined with a work-stealing library that I found on the Internet, the result several months later was a prototype parallel library for binary decision diagrams called Sylvan, sufficient for a Master's thesis and a publication.

Four years have passed since. Jaco offered me a position at the University of Twente as a PhD student, to continue the research on parallelizing operations on decision diagrams, experimenting with other types of decision diagrams and exciting applications of the technology. Research has been done, programs have been written, papers have been published, some of the typical "rollercoaster" highs and lows of PhD research have been explored, such as the year in which no paper was published, and some unexpected yet very rewarding experimental results. It turns out that there are always new research directions to explore and ideas to investigate, even when you expect that obvious ideas have already been explored and investigated by others. For being my patient supervisor and supportive promotor, I most certainly owe Jaco my gratitude.

In general, I am quite happy to have spent so much of my time in the last few years with the Formal Methods & Tools research group, in particular with Mark, whose excessively loud presence inspired me greatly, and whose productivity and self discipline I can never hope to match; Enno, who was always ready to offer a bright smile regardless of whether or not my attempt at humor was actually successful; Tri, who was (and probably still is) deadly afraid that the photos of our time together would somehow go viral and embarrass

him in his home country; Marcus, who endured (and seemed to enjoy) many an offensive remark, hurled at him with, as I like to imagine, great speed and frightening accuracy; and finally Alfons, who is always ready to offer advice and criticism, political discussion and commentary and dozens of daily Facebook notifications.

I am also quite happy to have Gijs and Roald as my paranymphs. I have known Gijs for a very long time now. We have lived in the same house, we have both been a member of the VGST and of Phi, we worked in the same office, doing research with the same supervisor, and we shared several interests like philosophy and politics. Roald I met when checking out the local branch of the political youth organisation the Jonge Democraten. He welcomed me in the local branch and made sure that I was their chairman the next week with the motivation to engage in many discussions in local politics and to organize lectures and various visits. I fondly remember our friendship and regular exchange of opinions.

During my PhD research, I visited research groups in Aachen and Beijing. Both times, Sebastian Junges was present. I would like to thank him for interesting conversations in Aachen, and in particular for the fun we had traveling in China for just over two weeks, visiting a research group in Beijing, a conference in Nanjing, and various touristic locations in both places.

I thank the members of my committee for approving my thesis and providing helpful comments.

Finally, I would like to thank my family and friends for their support over the years, in particular my parents for giving me my first C compiler when I was about 11 years old, saying that it was time I used a real programming language instead of writing DOS batch files and hand-written assembly programs. They tolerated my addiction to programming and using the computer in general, and stimulated my finger dexterity via piano lessons, without which writing all that code would have taken much longer.

Chapter 1

Introduction

The research of this thesis is about parallelizing algorithms for decision diagrams. As fundamental data structures in computer science, decision diagrams find applications in many areas, in particular in symbolic model checking [Bur+92; Bur+94]. Symbolic model checking studies the formal verification of properties of complex systems, such as communication protocols, controller software and risk models. Most computation time for symbolic model checking is spent in operations on decision diagrams. Hence, improving the performance of decision diagram operations improves the performance of symbolic model checking.

As a fundamental data structure in computer science, decision diagrams are not only extensively used in symbolic model checking, but also in logic synthesis [Mal+88; MF89; Soe+16], fault tree analysis [RA02; BCT07], test generation [BAA95; AR86], and even to represent access control lists [Fis+05]. A recent survey paper by Minato [Min13] provides an accessible history of research activity into decision diagrams, listing applications to data mining [LB06], Bayesian network and probabilistic inference models [MSS07; ISM11], and game theory [Sak+11].

In the past, the processing power of computers increased mostly by improvements in the clock speeds and the efficiency of processors, which often do not require adaptations to algorithms. However, as physical constraints seem to limit such improvements, further increases in processing power of modern machines inevitably come from using multiple cores. To make optimal use of the processing power of multi-core machines, algorithms must be adapted.

One of the primary goals when adapting algorithms for multiple cores (parallelization), is to obtain good parallel scalability. The ideal linear scalability would obtain a speedup of N times when using N cores. This ideal is often not possible in practice, in particular we see three things: 1) the parallel version may be slower than the original non-parallel version, this is called the

sequential overhead; 2) adding cores often does not result in corresponding speedups: if we get a speedup of 3x with 5 cores, and a speedup of 6x with 10 cores, then the parallel efficiency is only 60%; 3) often the performance plateaus or even degrades after a certain number of cores due to insufficient parallel work, bottlenecks and communication costs. Good parallel scalability is obtained when the sequential overhead is low, the parallel efficiency is high, and bottlenecks and costs that limit the maximum number of cores can be avoided.

This thesis studies the following main questions:

1. Is a scalable implementation of decision diagram operations possible, and if so, how?
2. Does the scalability of decision diagram operations extend to sequential algorithms that use them, such as symbolic model checking?
3. What is the additional effect of further parallelizing algorithms that use parallel decision diagram operations?

We study these questions by implementing a prototype parallel decision diagram library called Sylvan, which is described in Chapter 2. Sylvan is based on two main ingredients. The first ingredient is a work-stealing framework called Lace, which is built around a novel concurrent task queue in Chapter 3. This framework enables us to implement decision diagram operations as tasks that are executed in parallel. The second ingredient consists of concurrent data structures: a single shared concurrent hash table that stores all nodes of the decision diagrams, and a single concurrent operation cache that stores the intermediate results of operations for reuse. These data structures are described in Chapter 4. We study the performance and parallel scalability of Sylvan for two applications: symbolic model checking in Chapter 5, and symbolic bisimulation minimisation in Chapter 6.

The current chapter provides an introduction to the subjects discussed in this thesis. We first introduce symbolic model checking (Section 1.1), binary decision diagrams (Section 1.2), and parallelism (Section 1.3). We look at some earlier work in parallel decision diagrams in Section 1.4. In Section 1.5 we discuss the contributions in this thesis and Section 1.6 lists the publications that they are based on. Finally, Section 1.7 gives an outline of the rest of the thesis.

1.1 Symbolic model checking

As the modern society increasingly depends on automated and complex systems, the safety demands on such systems increase as well. We depend on automated systems for basic infrastructure, to clean our water, to supply energy, to control our cars and trains, to monitor and process our financial transactions

and for the internet. We use systems for entertainment when watching TV or using the phone, or for cooking with modern stoves, microwaves and fridges. Failure or unexpected behavior in these ubiquitous systems can have many consequences, from mild annoyances to fatal accidents. This motivates research into the formal verification of such systems, as well as computing properties such as failure rates and time to recovery.

In model checking, systems are modeled as sets of possible states of the system and transitions between these states. System states are typically represented by Boolean vectors. Fixed point algorithms, which are procedures that repeatedly apply some operation until a fixed point is reached, play a central role in many model checking algorithms. An example of a fixed point algorithm is state space exploration ("reachability"), which computes all states reachable from the initial state of the system. Many model checking algorithms depend on state space exploration to determine the number of states, to check if an invariant is always true, to find cycles and deadlocks, and so forth.

A major challenge in model checking is that the space and time requirements of these algorithms increase exponentially with the size of the models. One technique to alleviate this problem is symbolic model checking [Bur+92; Bur+94]. In symbolic model checking, sets of states and transitions are represented by their characteristic (Boolean) functions, which are stored using binary decision diagrams (BDDs), whereas in traditional explicit-state model checking, states and transitions are typically stored and treated individually. One advantage of using BDDs for fixed point computations is that equivalence testing is a trivial check, since BDDs uniquely represent Boolean functions. As small Boolean formulas could describe very large state spaces, symbolic model checking has been very successful to push the limits of model checking in the past [Bur+92]. Symbolic representations are also quite natural for the composition of multiple transition systems, e.g., when composing systems from subsystems.

Bisimulation minimisation Another technique to reduce the state space explosion problem is bisimulation minimisation. Bisimulation minimisation computes a minimal system equivalent to the original system with respect to some notion of equivalence, for example when applying some abstraction to the state space, ignoring irrelevant variables or actions, or abstracting from internal transitions. Symbolic bisimulation minimisation combines the bisimulation minimisation technique with decision diagrams. This can speed up this process considerably, especially for suitable models [WHB06; Wim+06; Der07b; Wim+07]. Symbolic bisimulation minimisation also acts as a bridge between symbolically defined models and explicit-state analysis techniques, especially for models that have a very large state space and only few distinguishable

reachable states. This typically happens when abstracting from internal details.

1.2 Binary decision diagrams

One of the most fundamental concepts in computer science is Boolean logic, with Boolean variables, which are either `true` or `false`. Boolean logic and variables are particularly fundamental, as all digital data can be expressed in binary form. Boolean formulas are defined on Boolean variables and contain operations such as conjunction ($x \wedge y$), disjunction ($x \vee y$), negation ($\neg x$) and quantification ($\exists$ and $\forall$). Boolean functions are functions $\mathbb{B}^N \to \mathbb{B}$ (on N inputs), with a Boolean formula representing the relation between the inputs and the output of the Boolean function. Binary decision diagrams (BDDs) are a canonical and often concise representation of Boolean functions [Ake78; Bry86].

A (reduced, ordered) BDD is a rooted directed acyclic graph with leaves 0 and 1. Each internal node has a variable label x_i and two outgoing edges labeled 0 and 1, called the "low" and the "high" edge. Furthermore, variables are encountered along each directed path according to a fixed variable ordering. Duplicate nodes and nodes with identical outgoing edges (redundant nodes) are forbidden. It is well known that every Boolean function is represented by a unique BDD [Bry86]. See Figure 1.1 for examples of simple BDDs.

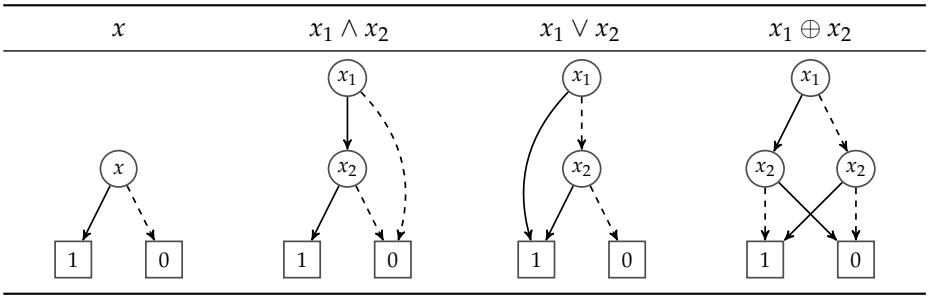


Figure 1.1 Binary decision diagrams for several Boolean functions. Internal nodes are drawn as circles with variables, and leaves as boxes. High edges are drawn solid, and low edges are drawn dashed. BDDs are evaluated by following the high edge when a variable x is `true`, or the low edge when it is `false`.

There are various equivalent ways to interpret a binary decision diagram, leading to the same Boolean function:

1. Consider every distinct path from the root of the BDD to the terminal 1. Every such path assigns `true` or `false` to the variables encountered along that path, by following either the high edge or the low edge. In this way, every path corresponds to a conjunction of literals, also called a cube. For

example, the cube $x_0\overline{x_1}x_3x_4\overline{x_5}\overline{x_7}$ corresponds to a path that follows the high edges of nodes labeled x_0, x_3 and x_4, and the low edges of nodes labeled x_1, x_5 and x_7. If the cubes $c_1, \ldots, c_k$ correspond to the k distinct paths in a BDD, then this BDD encodes the Boolean function $c_1 \vee \cdots \vee c_k$.

2. Alternatively, after computing $f_{x=1}$ and $f_{x=0}$ by interpreting the BDDs obtained by following the high and the low edges, a BDD node with variable label x represents the Boolean function $xf_{x=1} \vee \overline{x}f_{x=0}$.

In addition to BDDs with leaves 0 and 1, multi-terminal binary decision diagrams (MTBDDs) have been proposed [Bah+93; Cla+93] with arbitrary leaves, representing functions from the Boolean space $\mathbb{B}^N$ to other sets, for example integers ($\mathbb{B}^N \to \mathbb{N}$) and real numbers ($\mathbb{B}^N \to \mathbb{R}$). Complementary, multi-valued decision diagrams (MDDs) [Kam+98] generalize BDDs to the integer domain ($\mathbb{N}^N \to \mathbb{B}$).

1.3 Parallelism

A major goal in computing is to perform ever larger calculations and to improve their performance and efficiency. This can be accomplished using various techniques that are often orthogonal to each other, such as better algorithms, faster processors, and parallel computing using multiple processors. Faster hardware increases the performance of most computations, often regardless of the algorithm, although some algorithms benefit more from processor speed while others benefit more from faster memory access. For suitable algorithms, parallel processing can considerably improve the performance, on top of what is possible just by increased processor speeds. See e.g. the PhD thesis of Laarman [Laa14] for extensive work in multi-core explicit-state model checking.

A famous statement in computer science is Moore's Law [Moo65], which states that the number of transistors on chips doubles every 18 months. For a long time, one of the main consequences of a higher number of transistors, as well as their decreasing size, was that processors became faster and more efficient. However, physical constraints limit the opportunities for higher clock speeds, shifting attention from clock speeds to parallel processing. As a result, the processing power of modern chips continues to increase as Moore's Law predicts, but now efficient parallel algorithms are required to make use of multi-core computers.

For some algorithms, efficient parallelism is almost trivial. It is no coincidence that graphics cards contain thousands of small processors, resulting in massive speedups for very particular applications. Other algorithms are more difficult to parallelize. For example, some algorithms are inherently sequential, with few opportunities for the parallel execution of independent calculation paths. Other algorithms have enough independent paths for parallelization

in theory, but are difficult to parallelize in practice, for example because they are irregular and continuously require load balancing, moving work between processors. Some algorithms are memory intensive, i.e., they spend most of their time manipulating data in memory, which can result in bottlenecks due to the limited bandwidth between processors and the memory, as well as time spent waiting in locks.

The research in this thesis is about the parallelization of algorithms for decision diagrams, which are large directed acyclic graphs. They are typically irregular and mainly consist of unpredictable memory accesses with high demands on memory bandwidth. Decision diagrams are often used as the underlying operations of other algorithms. If the underlying decision diagram operations are parallelized, then sequential algorithms that use them may also benefit from the parallelization, although the effect may be small for algorithms that mostly consist of small decision diagram operations. We show this effect when applying parallel decision diagram operations to the (sequential) breadth-first-search symbolic state space exploration algorithm in Chapter 5. This already results in a good parallel speedup. We further show that even higher parallel speedups are obtained by also parallelizing the state space exploration itself (by exploiting a disjunctive partitioning of the transition relation) in addition to using parallel decision diagram operations.

Cache hierarchy and coherency In order to improve the performance of modern computer systems, most processors have several levels of caching. On contemporary multi-core systems, processors typically have a very fast but small L1 cache for each core, a slower but larger L2 cache that is sometimes shared between cores, and an even larger and slower L3 cache that is often shared with all cores on one processor.

The caches are connected with each other and with processor cores and with the memory via interconnect channels. On this interconnect network, data is transferred in blocks called cachelines, which are usually 64 bytes long. In addition, the "cache coherency protocol" ensures that all processors have a coherent view of the memory. When processors write or read cachelines, their caches at various levels communicate based on an internal state machine for each cacheline. Especially writing to memory often results in a lot of communication. For example, writing a cacheline results in an invalidation message to the other caches, which then have to refresh their view on the memory when they access that cacheline again. Because data is managed and transferred in blocks of 64 bytes, one issue in parallel computing is false sharing: if independent variables are stored on the same cacheline, then writing to one variable also causes the invalidation of the other variables, even if those other variables were not modified.

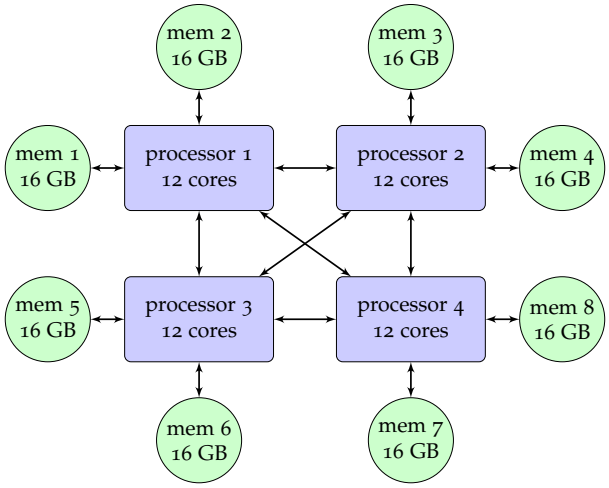


Figure 1.2 Example of a non-uniform memory access architecture, with in total 48 cores and 128 GB memory.

System architecture In this thesis, we work with non-uniform memory access (NUMA) architectures: systems that have multiple multi-core processors and multiple memories, connected via an interconnect network. For example, see Figure 1.2. The processors in NUMA machines view memory as a single uniform shared block of memory, but the underlying architecture is more like a message passing system. When reasoning about concurrency and bottlenecks in modern machines, one should consider the messages that are sent on the lower level due to the cache coherency protocol.

One consequence of having multiple memories is that typically memory access times depend on the distance between each processor and the accessed part of memory. The operating system offers some control over the location of certain memory blocks and to force threads to run on specific processors or processor cores. Although we do not discuss this in-depth in this thesis, trying to minimize the distance between accessed memory and the processors is something that we take into account when implementing the load balancing framework (Chapter 3) and the concurrent hash tables (Chapter 4).

Weak memory models In order to significantly improve the performance of many programs, processors typically have a so-called weak memory model. In a strong memory model, memory reads and memory writes are sequentially consistent, i.e., as if the operations of all the processors are executed in

some sequential order. If one processor performs a memory write, then other processors immediately see the updated value.

The x86 TSO memory model [Sew+10] that is used in many modern commodity processors, including the processors that we use for our experiments, is not sequentially consistent, but allows reordering reads before writes. Memory writes are buffered before reaching the caches and the memory, hence reads can occur before preceding memory writes are visible to other processors. The memory writes of each processor are not reordered, which is called "total store ordering". Special instructions called memory fences flush the write buffer before reading from memory. In reasoning about the behavior and correctness of algorithms on weak memory models, it is important to consider this reordering, as we see in Chapter 3.

Locking and lock-free programming Furthermore, for communication between processors, atomic operations are often used to avoid race conditions. A race condition exists when multiple threads access the same bytes in memory. Typically, such places in memory are then protected using "locks", but locks are notoriously bad for parallel performance, because other threads have to wait until the lock is released, and locks are often a bottleneck when many threads try to acquire the same lock.

A standard technique that avoids locks uses the `compare_and_swap` (`cas`) operation, which is an operation that is supported by many modern processors. This operation atomically compares the contents of a given address in shared memory to some given value and, if the contents match with the given value, changes the contents to a given new value. If multiple processors try to change the same bytes in memory using `cas` at the same time, then only one succeeds.

Datastructures that avoid locks are called non-blocking or lock-free. Such data structures often use the `cas` operation to make progress in an algorithm, rather than protecting a part that makes progress. For example, when modifying a shared variable, an approach using locks would first acquire the lock, then modify the variable, and finally release the lock. A lock-free approach would use `cas` to modify the variable directly. This requires only one memory operation rather than three, but lock-free approaches are typically more complicated to reason about, and prone to bugs that are more difficult to reproduce and debug.

In this thesis, we implement several non-blocking data structures, e.g., for the work-stealing framework in Chapter 3 and for the hash tables in Chapter 4.

1.4 Earlier work in parallel BDDs

This section is largely based on earlier literature reviews we presented in [DLP13; DP15].

Massively parallel computing (early '90s). In the early '90s, researchers tried to speed up BDD manipulation by parallel processing. The first paper [KC90] views BDDs as automata, and combines them by computing a product automaton followed by minimization. Parallelism arises by handling independent subformulae in parallel: the expansion and reduction algorithms themselves are not parallelized. They use locks to protect the global hash table, but this still results in a speedup that is almost linear with the number of processors. Most other work in this era implemented BFS algorithms for vector machines [OIY91] or massively parallel SIMD machines [CGS92; GRS95] with up to 64K processors. Experiments were run on supercomputers, like the Connection Machine. Given the large number of processors, the speedup (around 10 to 20) was disappointing.

Parallel operations and constructions An interesting contribution in this period is the paper by Kimura et al. [KIH92]. Although they focus on the construction of BDDs, their approach relies on the observation that suboperations from a logic operation can be executed in parallel and the results can be merged to obtain the result of the original operation. Our solution to parallelizing BDD operations follows the same line of thought, although the work-stealing method for efficient load balancing that we use was first published two years later [Blu94]. Similar to [KIH92], Parasuram et al. implement parallel BDD operations for distributed systems, using a "distributed stack" for load balancing, with speedups from 20–32 on a CM-5 machine [PSC94]. Chen and Banerjee implement the parallel construction of BDDs for logic circuits using lock-based distributed hash tables, parallelizing on the structure of the circuits [CB99]. Yang and O'Hallaron [YO97] parallelize breadth-first BDD construction on multi-processor systems, resulting in reasonable speedups of up to 4x with 8 processors, although there is a significant synchronization cost due to their lock-protected unique table.

Distributed memory solutions (late '90s). Attention shifted towards Networks of Workstations, based on message passing libraries. The motivation was to combine the collective memory of computers connected via a fast network. Both depth-first [ACM96; SB96; Bia+97] and breadth-first [San+96] traversal have been proposed. In the latter, BDDs are distributed according to variable levels. A worker can only proceed when its level has a turn, so these algorithms

are inherently sequential. The advantage of distributed memory is not that multiple machines can perform operations faster than a single machines, but that their memory can be combined in order to handle larger BDDs. For example, even though [SB96] reports a nice parallel speedup, the performance with 32 machines is still 2x slower than the non-parallel version. BDDNOW [MH98] is the first BDD package that reports some speedup compared to the non-parallel version, but it is still very limited.

Parallel symbolic reachability (after 2000). After 2000, research attention shifted from parallel implementations of BDD operations towards the use of BDDs for symbolic reachability in distributed [GHS06; CC04] or shared memory [ELC07; CZJ09]. Here, BDD partitioning strategies such as horizontal slicing [CC04] and vertical slicing [Hey+00] were used to distribute the BDDs over the different computers. Also the saturation algorithm [CLS01], an optimal iteration strategy in symbolic reachability, was parallelized using horizontal slicing [CC04] and using the work-stealer Cilk [ELC07], although it is still difficult to obtain good parallel speedup [CZJ09].

Multi-core BDD algorithms. There is some recent research on multi-core BDD algorithms. There are several implementations that are thread-safe, i.e., they allow multiple threads to use BDD operations in parallel, but they do not offer parallelized operations. In a thesis on the BDD library JINC [Oss10], Chapter 6 describes a multi-threaded extension. JINC's parallelism relies on concurrent tables and delayed evaluation. It does not parallelize the basic BDD operations, although this is mentioned as possible future research. Also, a recent BDD implementation in Java called BeeDeeDee [LMS14] allows execution of BDD operations from multiple threads, but does not parallelize single BDD operations. Similarly, the well-known sequential BDD implementation CUDD [Som15] supports multi-threaded applications, but only if each thread uses a different "manager", i.e., unique table to store the nodes in. Except for our contributions [DLP12; DLP13; DP15] related to Sylvan, there is no recent published research on modern multi-core shared-memory architectures that parallelizes the actual operations on BDDs. Recently, Oortwijn et al. [Oor15] continued our work by parallelizing BDD operations on shared memory abstractions of distributed systems using remote direct memory access.

1.5 Contributions

This thesis contains several contributions related to the multi-core implementation of decision diagrams. This section summarizes these contributions.

1.5.1 Scalable hash tables with garbage collection

The two main data structures used for decision diagrams are the hash table used to store the nodes of the decision diagrams, and the operation cache that stores results of intermediate operations. This operation cache is required for decision diagram operations, as we discuss in Chapter 2. The parallel scalability of algorithms for decision diagrams depends for a large part on these two main data structures. Chapter 4 presents these data structures based on the work that we did in [DLP13; DP15; DP16b].

An essential part of manipulating decision diagrams is garbage collection. Most operations on decision diagrams continuously create new nodes in the nodes table, and to free up space for these nodes, unused nodes must often be deleted. Our concurrent hash table is based on the efficient scalable hash table by Laarman et al. [LPW10]. This hash table uses a short-lived local lock that only blocks concurrent operations that are very likely to insert the same data, and uses a novel variation on linear probing based on cachelines ("walk the line"). This table only supports the `find-or-insert` operation. In [DLP13], we extend the hash table to support garbage collection. Our initial implementation reserves space in each bucket to count the number of internal and external references to each node. We modify this hash table [DP15] to remove the reference count and replace the implementation by a mark-and-sweep approach, with the external references stored outside the hash table. Finally, in [DP16b], we improve the hash table with bitmaps for bookkeeping (as described in Chapter 4), further simplifying the design of the hash table, as well as removing the short-lived lock and reducing the number of atomic operations per call.

The operation cache stores intermediate results of BDD operations. It is well known that an operation cache is required to reduce the worst-case time complexity of BDD operations from exponential time to polynomial time. In practice, we do not guarantee this property, but find that we obtain a better performance by allowing the cache to overwrite earlier results when there is a hash collision. We also implement a feature called "caching granularity" which controls how often computation results are cached. We see in practice that the cost of occasionally recomputing suboperations is less than the cost of always consulting the operation cache.

We implement the operation cache as a simplified hash table, which deals with hash collisions by overwriting existing cached results and prefers aborting operations when there are conflicts (such as race conditions). This avoids having to use locks and improves the performance in practice.

1.5.2 Work-stealing framework Lace

Since one of the important parts of a scalable multi-core implementation of decision diagram operations is load balancing, we investigate task-based parallelism using work-stealing in Chapter 3. Here, we present a novel data structure called a non-blocking split deque for work-stealing, which forms the basis of the work-stealing framework Lace. This framework is similar to the existing fine-grained work-stealing framework Wool [Fax08; Fax10]. In our initial implementation, we used Wool for load balancing as it is relatively easy to use and performs well compared to other libraries, especially compared to the well-known framework Cilk [PBF10]. We implemented our own framework Lace as a research vehicle and for features that are particularly useful for parallel decision diagrams, such as a feature where all workers cooperatively suspend their current tasks and start a new task tree. This is used to implement stop-the-world garbage collection in Sylvan.

1.5.3 Multi-core decision diagram library Sylvan

One of the main contributions of this thesis is the reusable multi-core decision diagram library Sylvan. Sylvan implements parallelized operations on decision diagrams for multi-core machines, and can replace existing non-parallel implementations to bring the processing power of multi-core machines to non-parallel applications. Sylvan implements binary decision diagrams (BDDs), list decision diagrams (LDDs), which are a kind of multi-valued decision diagrams used in the model checking toolset LTSmin [Kan+15], and multi-terminal binary decision diagrams (MTBDDs) [Bah+93; Cla+93]. We present these contributions in Chapter 2.

Parallel BDD and LDD operations For BDDs, Sylvan parallelizes many standard operations that are also implemented by sequential BDD libraries, such as the binary operators, existential and universal quantification, variable substitution and functional composition. Sylvan also implements parallelized versions of the minimization algorithms `restrict` and `constrain` (also called generalized cofactor), based on sibling-substitution [CM90]. In model checking, a specific variable ordering is popular for transition relations and `relnext` and `relprev` are specialized implementations to compute the successors and predecessors of sets of states for this particular variable ordering.

While these functions are not new, their parallel implementation in Sylvan is a novel contribution which provides good parallel scalability in real-world applications such as symbolic model checking and bisimulation minimisation.

In the application of model checking with the toolset LTSmin (Section 1.5.4 and Chapter 5), LDDs are an efficient representation of state spaces where state

variables are integers. Sylvan implements various set operations using LDDs, such as `union` ($f \vee g$), `intersect` ($f \wedge g$), `minus` ($f \wedge \neg g$), `project` (projection by abstracting from variables), and a number of specialized operations for LTSmin, also resulting in good parallel scalability.

Extensible parallel MTBDD framework Applications like bisimulation minimisation of probabilistic models require the representation of functions to other domains, such as real numbers or rational numbers, e.g., for representing the rates of Markovian transitions. MTBDDs can be used to store such functions. The well-known BDD package CUDD [Som15] implements MTBDDs (also called "algebraic decision diagrams" [Bah+93]) with floating-point (`double`) leaves. Several modified versions of CUDD exist that use different leaf types, such as in the `sigref` tool for bisimulation minimisation [WB10], which uses the GMP library for rational numbers.

Our approach offers a framework for various leaf node types. By default, Sylvan supports integers (`int64_t`), floating-point numbers (`double`), rational numbers with 32-bit numerators and 32-bit denominators, and rational numbers from the GMP library (`mpq_t`). The framework implements a number of multi-core operations on MTBDDs, such as `plus`, `minus`, `max`, `min` and `times`, as well as the algorithms for variable abstraction, `abstract_plus` (which is similar to existential quantification for Boolean functions), `abstract_times` (which is similar to universal quantification for Boolean functions), `abstract_max` and `abstract_min`. This framework is designed for adding custom operations and custom types, providing an example with the support for the GMP library.

1.5.4 Multi-core on-the-fly reachability

The main application for which we developed Sylvan is symbolic model checking. As discussed above, a fundamental algorithm in symbolic model checking is state space exploration. Chapter 5 discusses the application of parallel decision diagrams in the model checking toolset LTSmin, based on the work that we did in [DLP12; DLP13; DP15; Kan+15].

The model checking toolset LTSmin provides a language independent Partitioned Next-State Interface (Pins), which connects various input languages to model checking algorithms [BPW10; LPW11; DLP12; Kan+15]. In Pins, states are vectors of N integers. Transitions are distinguished in K disjunctive *transition groups*. The symbolic model checker in LTSmin is based around state space exploration to learn the model and check properties on-the-fly.

Initially, we simply use Sylvan for the BDD operations in LTSmin. We keep the algorithms in LTSmin sequential and only use the multi-core BDD operations. This already results in a good parallel speedup, of up to 30x on

48 cores, as discussed in Chapter 5. Subsequently, we exploit the disjunctive partitioning of the transitions and parallelize state space exploration in LTSmin using the Lace framework. We also parallelize the on-the-fly transition learning that LTSmin offers using specialized BDD operations. This results in a speedup of up to 40x on 48 cores. Besides BDDs, LTSmin also uses LDDs for model checking. We implement multi-core LDD operations in Sylvan and demonstrate that our parallel implementation of LDDs results in a faster implementation compared to BDDs, while the same high parallel scalability.

1.5.5 Multi-core symbolic bisimulation minimisation

Bisimulation minimisation alleviates the exponential growth of transition systems in model checking by computing the smallest system that has the same behavior as the original system according to some notion of equivalence. One popular strategy to compute a bisimulation minimisation is signature-based partition refinement [BO03]. This can be performed symbolically using binary decision diagrams to allow models with larger state spaces to be minimised [WHB07; Wim+06].

In [DP16a], on which Chapter 6 is based, we use the MTBDD framework in Sylvan for symbolic bisimulation minimisation. We study strong and branching symbolic bisimulation for labeled transition systems, continuous-time Markov chains, and interactive Markov chains. We introduce the notion of partition refinement with partial signatures. We extend Sylvan to parallelize the signature refinement algorithm, and develop a new parallel BDD algorithm to refine a partition, which conserves previous block numbers and uses a parallel data structure to store block number assignments. We also present a specialized BDD algorithm for the computation of inert transitions. The experimental evaluation, based on benchmarks from the literature, demonstrates a speedup of up to 95x sequentially. In addition, we find parallel speedups of up to 17x due to parallelisation with 48 cores. Finally, we present the implementation of these algorithms as a versatile framework that can be customized for state-based bisimulation minimisation in various contexts.

1.6 Publications

Parts of this thesis have been published in the following publications:

[DLP13] Tom van Dijk, Alfons Laarman, and Jaco van de Pol. "Multi-Core BDD Operations for Symbolic Reachability." In: *ENTCS* 296 (2013), pp. 127–143

This paper, presented at PDMC 2012, lays the foundations of multi-core BDD operations for symbolic model checking, using the work-stealing framework

Wool and our initial implementation of the concurrent hash table. This version of the hash table uses reference counting for the garbage collection. We demonstrate the viability of our approach to multi-core BDD operations by evaluating its performance on benchmarks of symbolic reachability.

[DP14] Tom van Dijk and Jaco van de Pol. "Lace: Non-blocking Split Deque for Work-Stealing." In: *MuCoCoS*. vol. 8806. LNCS. Springer, 2014, pp. 206–217

This paper, presented at the MuCoCoS workshop in 2014, presents our work-stealing framework Lace, based on a novel non-blocking queue for work-stealing, and demonstrates its performance using a standard set of benchmarks. Chapter 3 is mostly based on this paper.

[DP15] Tom van Dijk and Jaco van de Pol. "Sylvan: Multi-Core Decision Diagrams." In: *TACAS*. vol. 9035. LNCS. Springer, 2015, pp. 677–691

This paper, presented at TACAS 2015, presents an extension of Sylvan with operations on list decision diagrams (LDDs) for symbolic model checking. We also investigate additional parallelism on top of the parallel BDD/LDD operations, by exploiting the disjunctive partitioning of the transition relation in the model checking toolset LTSmin. Applying these transition relations in parallel results in improved parallel speedups. In addition, we extend Sylvan with support for parallel transition learning, which is required for scalable on-the-fly reachability. We replace the concurrent hash table with a modified version, that uses a mark-and-sweep approach for garbage collection, eliminating some bookkeeping and complexity in the hash table implementation.

[DP16a] Tom van Dijk and Jaco van de Pol. "Multi-Core Symbolic Bisimulation Minimisation." In: *TACAS*. vol. 9636. LNCS. Springer, 2016, pp. 332–348

This paper, presented at TACAS 2016, is about the application of Sylvan to symbolic bisimulation minimisation. This technique creates the smallest model that is equivalent to the original model according to some notion of bisimulation equivalence. We treated strong and branching bisimulation. We show how custom BDD operations result in a large speedup compared to the original, and that using multi-core BDD operations results in good parallel scalability. This paper has been selected to be extended for a journal paper in a special issue of STTT. Chapter 6 is mostly based on this paper.

[DP16b] Tom van Dijk and Jaco van de Pol. "Sylvan: Multi-core Framework for Decision Diagrams." In: *STTT* (2016). Accepted.

This journal paper is an extended version of [DP15], which was selected for a special issue of STTT, and presents the extension of Sylvan with a versatile

implementation of MTBDDs, allowing symbolic computations on integers, floating-points, rational numbers and other types. Furthermore, we modify the nodes table with a version that requires fewer `cas` operations per created node. This paper also elaborates in more detail on the operation cache, parallel garbage collection and details on memory management in Sylvan.

The author of this thesis has also contributed to the following publications:

[DLP12] Tom van Dijk, Alfons W. Laarman, and Jaco van de Pol. "Multi-core and/or Symbolic Model Checking." In: *ECEASST* 53 (2012)

This invited paper at AVOCS reviews the progress in high-performance model checking using the model checking toolset LTSmin and mentions Sylvan as a basis for scalable parallel symbolic model checking.

[Kan+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. "LTSmin: High-Performance Language-Independent Model Checking." In: *TACAS 2015*. Vol. 9035. LNCS. Springer, 2015, pp. 692–707

This paper presents a number of extensions to the LTSmin model checking toolset, with support for new modelling languages, additional analysis algorithms, and multi-core symbolic model checking using Sylvan. The paper presents an overview of the toolset and its recent changes, and we demonstrate its performance and versatility in two case studies.

[Dij+15] Tom van Dijk, Ernst Moritz Hahn, David N. Jansen, Yong Li, Thomas Neele, Mariëlle Stoelinga, Andrea Turrini, and Lijun Zhang. "A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking." In: *SETTA*. vol. 9409. LNCS. Springer, 2015, pp. 35–51

This paper compares the performance of various BDD/MTBDD packages for the analysis of large systems using symbolic (probabilistic) methods. We provide experimental results for several well-known probabilistic benchmarks and study the effect of several optimisations. Our experiments show that no BDD package dominates on a single core, but that parallelisation with Sylvan yields significant speedups.

[ODP15] Wytse Oortwijn, Tom van Dijk, and Jaco van de Pol. "A Distributed Hash Table for Shared Memory." In: *Parallel Processing and Applied Mathematics*. Vol. 9574. LNCS. Springer, 2015, pp. 15–24

This paper, presented at PPAM 2015, studies the performance of a distributed hash table design, which uses a shared memory abstraction with Infiniband and

remote direct memory access. This paper is part of the research by Oortwijn into parallelizing BDD operations on distributed systems, similar to our approach on multi-core systems.

1.7 Overview

The remainder of this thesis is organized in the following way:

Chapter 2 gives a high-level overview of decision diagrams and decision diagram operations. We discuss the design of the parallel decision diagram package Sylvan and the various parallelized algorithms, as well as the MTBDD framework.

Chapter 3 presents the work-stealing framework Lace and our non-blocking work-stealing deque.

Chapter 4 discusses the main concurrent data structures: the hash table that contains the nodes of the decision diagrams, and the operation cache that stores the intermediate results of the operations.

Chapter 5 demonstrates the application of multi-core decision diagram operations in LTSmin. We show that just using the multi-core operations in a sequential state space exploration results in a speedup of up to 30x with 48 cores, which is further improved to up to 40x by also parallelizing the state space exploration algorithm.

Chapter 6 applies the multi-core decision diagram operations of Sylvan to symbolic bisimulation minimisation. We also implement custom (MT)BDD operations for bisimulation minimisation, resulting in speedups of 95x sequentially, and additionally in parallel up to 17x with up to 48 cores.

Chapter 7 concludes the thesis with a reflection of what has been achieved, and some promising directions for future work.

Chapter 2

Decision diagrams

The current chapter provides a more detailed overview of decision diagrams and the parallel decision diagram operations in Sylvan. We first review Boolean logic (Section 2.1.1), binary decision diagrams (Section 2.1.2), multi-terminal decision diagrams (Section 2.1.3), multi-valued decision diagrams (Section 2.1.4), and list decision diagrams (Section 2.1.5). Section 2.2 discusses the main challenges for parallelism and our strategy to parallelize the decision diagram operations in Sylvan. Here, we also discuss garbage collection and memory management in Sylvan. Section 2.3 gives an overview of the BDD algorithms that we parallelized in Sylvan. Section 2.4 presents the MTBDD framework in Sylvan and describes the main operations that we implemented. Finally, section 2.5 briefly describes the LDD algorithms that we parallelized for set operations in symbolic model checking.

2.1 Preliminaries

2.1.1 Boolean logic and notation

One of the most fundamental concepts in computer science is Boolean logic, with Boolean variables, which are either `true` or `false`. Boolean logic and variables are particularly fundamental, as all digital data can be expressed in binary form. Boolean formulas are defined on Boolean variables and contain operations such as conjunction ($x \wedge y$), disjunction ($x \vee y$), negation ($\neg x$) and quantification ($\exists$ and $\forall$). Boolean functions are functions $\mathbb{B}^N \to \mathbb{B}$ (on N inputs), with a Boolean formula representing the relation between the inputs and the output of the Boolean function.

In this thesis, we often use 0 to denote `false` and 1 to denote `true`. In addition, we use the notation $f_{x=v}$ to denote a Boolean function f where the variable x is given value v. For example, given a function f defined on N

variables:

$$f(x_1, \ldots, x_i, \ldots, x_N)_{x_i=0} \equiv f(x_1, \ldots, 0, \ldots, x_N)$$
$$f(x_1, \ldots, x_i, \ldots, x_N)_{x_i=1} \equiv f(x_1, \ldots, 1, \ldots, x_N)$$

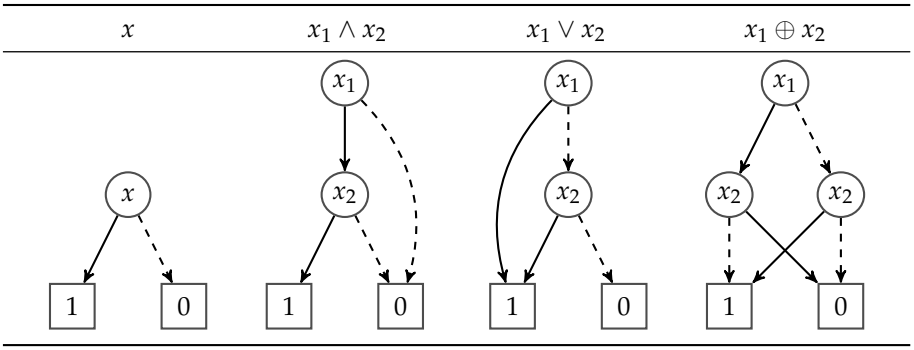
This notation is especially relevant for decision diagrams, as they are recursively defined on the value of a variable.

2.1.2 Binary decision diagrams

Binary decision diagrams (BDDs) are a concise and canonical representation of Boolean functions $\mathbb{B}^N \to \mathbb{B}$ [Ake78; Bry86], and are one of the most basic structures in discrete mathematics and computer science.

A (reduced, ordered) BDD is a rooted directed acyclic graph with leaves 0 and 1. Each internal node has a variable label x_i and two outgoing edges labeled 0 and 1, called the "low" and the "high" edge. Furthermore, variables are encountered along each directed path according to a fixed variable ordering. Duplicate nodes (two nodes with the same variable label and outgoing edges) and nodes with two identical outgoing edges (redundant nodes) are forbidden. It is well known that, given a fixed order, every Boolean function is represented by a unique BDD [Bry86].

The following figure shows the BDDs for several Boolean functions. Internal nodes are drawn as circles with variables, and leaves as boxes. High edges are drawn solid, and low edges are drawn dashed. Given a valuation of the variables, BDDs are evaluated by following the high edge when the variable x is `true`, or the low edge when it is `false`.



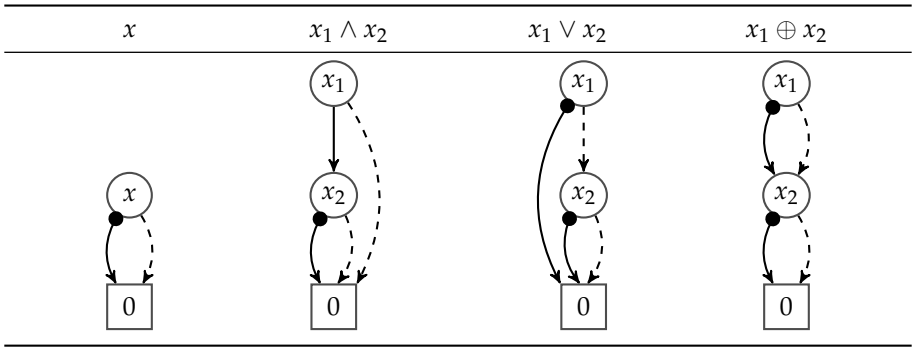
There are various equivalent ways to interpret a binary decision diagram, leading to the same Boolean function:

1. Consider every distinct path from the root of the BDD to the terminal 1. Every such path assigns `true` or `false` to the variables encountered along

that path, by following either the high edge or the low edge. In this way, every path corresponds to a conjunction of literals, sometimes called a cube. For example, the cube $x_0\overline{x_1}x_3x_4\overline{x_5}$ corresponds to a path that follows the high edges of nodes labeled x_0, x_3 and x_4, and the low edges of nodes labeled x_1 and x_5. If the cubes $c_1, \ldots, c_k$ correspond to the k distinct paths in a BDD, then this BDD encodes the function $c_1 \vee \cdots \vee c_k$.

2. Alternatively, after computing $f_{x=1}$ and $f_{x=0}$ by interpreting the BDDs obtained by following the high and the low edges, a BDD node with variable label x represents the Boolean function $xf_{x=1} \vee \overline{x}f_{x=0}$.

In addition, we use complement edges [BRB90] as a property of an edge to denote the negation of a BDD, i.e., the leaf 1 in the BDD will be interpreted as 0 and vice versa, or in general, each terminal node will be interpreted as its negation. This is a well-known technique. We write $\neg$ to denote toggling this property on an edge. The following figure shows the BDDs for the same simple examples as above, but with complement edges:

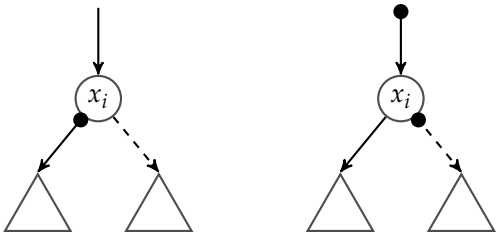


As this example demonstrates, always strictly fewer nodes are required, and there is only one ("false") terminal node. The terminal "true" is simply a complement edge to "false". We only allow complement marks on the high edges to maintain the property that BDDs uniquely represent Boolean functions (see also below).

The interpretation of a BDD with complement edges is as follows:

1. Count the complement edges on each path to the terminal 0. Since negation is an involution ($\neg\neg x = x$), each path with an odd number of complement edges is a path to "true", and with cubes $c_1, \ldots, c_k$ corresponding to all such paths, the BDD encodes the Boolean function $c_1 \vee \cdots \vee c_k$.
2. If the high edge has a complement mark, then the BDD node represents the Boolean function $x\neg f_{x=1} \vee \overline{x}f_{x=0}$, otherwise $xf_{x=1} \vee \overline{x}f_{x=0}$.

With complement edges, the following BDDs are identical:



Complement edges thus introduce a second representation of a Boolean function: if we toggle the complement mark on the two outgoing edges and on all incoming edges, we find that it encodes the same Boolean function. By forbidding a complement on one of the outgoing edges, for example the low edge, BDDs remain canonical representations of Boolean functions, since then the representation without a complement mark on the low edge is always used [BRB90].

2.1.3 Multi-terminal binary decision diagrams

In addition to BDDs with leaves 0 and 1, multi-terminal binary decision diagrams (MTBDDs) have been proposed [Bah+93; Cla+93] with arbitrary leaves, representing functions from the Boolean space $\mathbb{B}^N$ onto any set. For example, MTBDDs can have leaves representing integers (encoding $\mathbb{B}^N \to \mathbb{N}$), floating-point numbers (encoding $\mathbb{B}^N \to \mathbb{R}$) and rational numbers (encoding $\mathbb{B}^N \to \mathbb{Q}$). In our implementation of MTBDDs, we also allow for partially defined functions, using a leaf $\perp$. See Figure 2.1 for a simple example of such an MTBDD.

Similar to the interpretation of BDDs, MTBDDs are interpreted as follows:

1. An MTBDD encodes functions from a domain $D \subseteq \mathbb{B}^N$ onto some codomain C, such that for each path to a leaf $V \in C$, all inputs matching

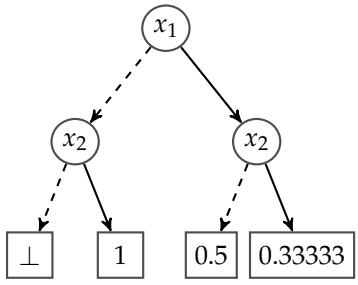


Figure 2.1 A simple MTBDD for a function which maps $\overline{x_1}x_2$ to 1, $x_1\overline{x_2}$ to 0.5, and x_1x_2 to 0.33333. The function is undefined for the input $\overline{x_1}\overline{x_2}$.

the corresponding cube c map to V. Also, given all such cubes $c_1, \ldots, c_k$, the domain D equals $c_1 \vee \cdots \vee c_k$. All paths corresponding to cubes not in D, i.e., for which the function is not defined, lead to the leaf $\bot$.

2. If an MTBDD is a leaf with the label V, then it represents the function $f(x_1, \ldots, x_N) \equiv V$. Otherwise, it is an internal node with label x. After recursively computing $f_{x=1}$ and $f_{x=0}$ by interpreting the MTBDDs obtained by following the high and the low edges, the node represents a function $f(x_1, \ldots, x_N) \equiv$ if x then $f_{x=1}$ else $f_{x=0}$.

Similar to BDDs, MTBDDs can have complement edges. This works only for leaf types for which negation is properly defined, i.e., each leaf x has a unique negated counterpart $\neg x$, such that $\neg\neg x = x$ and $\neg x \neq x$. In general, this does not work for numbers as $0 = -0$ in ordinary arithemetic. In addition, this also does not work for partially defined functions, as the negation of $\bot$ is not properly defined. In practice this means that we do not use complement edges on MTBDDs, except for total functions that are Boolean (Boolean MTBDDs are identical to BDDs, see also Section 2.2.2).

2.1.4 Multi-valued decision diagrams

Multi-valued decision diagrams (MDDs, sometimes also called multi-way decision diagrams) are a generalization of BDDs to the other domains, such as integers [Kam+98]. Whereas BDDs represent functions $\mathbb{B}^N \to \mathbb{B}$, MDDs represent functions $\mathbb{D}_1 \times \cdots \times \mathbb{D}_N \to \mathbb{B}$, for finite domains $\mathbb{D}_1, \ldots, \mathbb{D}_N$. They are typically used to represent functions on integer domains like $(\mathbb{N}_{<v})^N$.

Instead of 2 outgoing edges, each internal MDD node with variable x_i has n_i labeled outgoing edges. For example for integers, these edges could be labeled 0 to $n_i - 1$. Like BDDs, MDDs can be used to represent sets by their

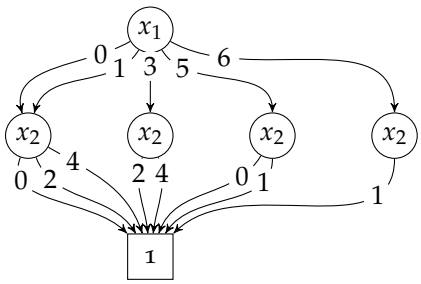


Figure 2.2 Edge-labeled MDD (hiding the paths to 0) representing the set $\{\langle 0,0\rangle, \langle 0,2\rangle, \langle 0,4\rangle, \langle 1,0\rangle, \langle 1,2\rangle, \langle 1,4\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 5,0\rangle, \langle 5,1\rangle, \langle 6,1\rangle\}$.

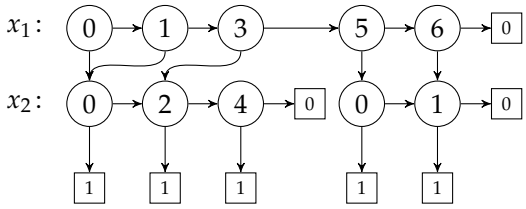


Figure 2.3 LDD representing the set $\{\langle 0,0\rangle, \langle 0,2\rangle, \langle 0,4\rangle, \langle 1,0\rangle, \langle 1,2\rangle, \langle 1,4\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 5,0\rangle, \langle 5,1\rangle, \langle 6,1\rangle\}$. We draw the same leaf multiple times for aesthetic reasons.

characteristic function. See Figure 2.2 for an example of an MDD representing a set of integer pairs, where we hide the edges to terminal 0 to improve the readability. In this thesis, we study list decision diagrams (see below) as an alternative to multi-valued decision diagrams.

2.1.5 List decision diagrams

List decision diagrams (LDDs) are an alternative to multi-valued decision diagrams. They represent sets of integer vectors, such as sets of states in model checking. List decision diagrams encode functions $(\mathbb{N}_{<v})^N \to \mathbb{B}$. LDDs were initially described in [BP08, Sect. 5].

A list decision diagram is a rooted directed acyclic graph with leaves 0 and 1. Each internal node has a value v and two outgoing edges labeled $>$ and $=$, also called the "right" and the "down" edge. Along the "right" edges, values v are encountered in ascending order. The "down" edge never points to leaf 0 and the "right" edge never points to leaf 1. Duplicate nodes are forbidden.

LDD nodes have a property called a level (and its dual, depth), which is defined as follows: the root node is at the first level, nodes along "right" edges stay in the same level, while "down" edges lead to the next level. The depth of an LDD node is the number of "down" edges to leaf 1. All maximal paths from an LDD node have the same depth.

See Figure 2.3 for an example of an LDD that represents the same set of integer pairs as the MDD in Figure 2.2.

There are various equivalent ways to interpret a list decision diagram, leading to the same set of integer vectors:

1. Consider the paths from the root of an LDD of depth k to the terminal 1. Every such path follows a "down" edge exactly k times, and assigns the value v_i of the node at the level i (with $1 \leq i \leq k$), where the "down" edge is followed. In this way, every path corresponds to a k-tuple $(v_1, \ldots, v_k)$.

Then the LDD represents the set of all the k-tuples that correspond to these paths.

2. An LDD with value v represents the set $\{v\,w \mid w \in S_{\text{down}}\} \cup S_{\text{right}}$, where S_{down} and S_{right} are the interpretations of the LDDs obtained by following the "down" and the "right" edge, and the leaves 0 and 1 are represented by $\emptyset$ and $\{\epsilon\}$, respectively.

LDDs compared to MDDs. A typical method to store MDDs in memory is to store the variable label x_i plus an array holding all n_i edges (pointers to nodes), e.g., in [MD02]: `struct node { int lvl; node* edges[]; }`. New nodes are dynamically allocated using `malloc` and a hash table ensures that no duplicate MDD nodes are created. Alternatively, one could use a large `int[]` array to store all MDDs (each MDD is represented by $n_i + 1$ consecutive integers) and represent edges to an MDD as the index of the first integer. In [CMS03], the edges are stored in a separate `int[]` array to allow the number of edges n_i to vary. Implementations of MDDs that use arrays to implement MDD nodes have two disadvantages. (1) For *sparse* sets (where only a fraction of the possible values are used, and outgoing edges to 0 are not stored) using arrays is a waste of memory. (2) MDD nodes typically have a variable size, complicating memory management. List decision diagrams can be understood as a linked-list representation of "quasi-reduced" MDDs. LDDs were initially described in [BP08, Sect. 5]. Like MDDs for integer domains, they encode functions $(\mathbb{N}_{<v})^N \to \mathbb{B}$.

Quasi-reduced (MT)BDDs and MDDs are a variation of normal (fully-reduced) (MT)BDDs and MDDs. Instead of forbidding redundant nodes (with two identical outgoing edges), quasi-reduced (MT)BDDs and MDDs forbid skipping levels. Quasi-reduced (MT)BDDs and MDDs are also canonical representations of (MT)BDDs and MDDs. In [CMS03], Ciardo et al. mention advantages of quasi-reduced MDDs: edges that skip levels are more difficult to manage and quasi-reduced MDDs are cheaper than alternatives to keep saturation operations correct. Also, the variables labels do not need to be stored as they follow implicitly from the depth of the MDD.

LDDs have several advantages compared to MDDs [BP08]. LDD nodes are binary, so they have a fixed node size which is easier for memory allocation. They are better for sparse sets: valuations that lead to 0 simply do not appear in the LDD. LDDs also have more opportunities for the sharing of nodes, as demonstrated in the example of Figure 2.2, where the LDD encoding the set $\{2,4\}$ is used for the set $\{0,2,4\}$ and reused for the set $\{\langle 3,2\rangle, \langle 3,4\rangle\}$, and similarly, the LDD encoding $\{1\}$ is used for $\{0,1\}$ and for $\{\langle 6,1\rangle\}$. A disadvantage of LDDs is that their linked-list style introduces edges "inside"

the MDD nodes, requiring more memory pointers, similar to linked lists compared with arrays.

2.2 Parallelizing decision diagrams

The requirements for the efficient parallel implementation of decision diagrams are not the same as for a sequential decision diagram library. We refer to the paper by Somenzi [Somo1] for a detailed discussion on the implementation of decision diagrams. Somenzi already established several aspects of a BDD package. The two central data structures of a BDD package are the *unique table* (or *nodes table*) and the *computed table* (or *operation cache*). Furthermore, garbage collection is essential for a BDD package, as most BDD operations continuously create and discard BDD nodes. This section discusses these topics in the context of a multi-core implementation.

Section 2.2.1 describes the core ingredients of parallel decision diagram operations using a generic example of a BDD operation. Section 2.2.2 describes how we represent decision diagram nodes in memory. In Section 2.2.3 and Section 2.2.4 we discuss the unique table and the computed table in the parallel context. Finally, in Section 2.2.5 we discuss garbage collection.

2.2.1 Parallel operations

In this subsection we look at Algorithm 2.1, a generic example of a BDD operation. This algorithm takes two inputs, the BDDs x and y, to which a binary operation F is applied.

```
1  def apply(x, y, F):
2      if x and y are leaves or trivial : return F(x, y)
3      Normalize/simplify parameters
4      if result ← cache[(x, y, F)] : return result
5      v = topvar(x,y)
6      do in parallel:
7          low ← apply(x_{v=0}, y_{v=0}, F)
8          high ← apply(x_{v=1}, y_{v=1}, F)
9      result ← lookupBDDnode(v, low, high)
10     cache[(x, y, F)] ← result
11     return result
```

Algorithm 2.1 Example of a parallelized BDD algorithm: apply a binary operator F to BDDs x and y.

Most decision diagram operations first check if the operation can be applied immediately to x and y (line 2). This is typically the case when x and y are leaves. Often there are also other trivial cases that can be checked first.

We assume that F is a function that, given the same parameters, always returns the same result. Therefore we can use a cache to store these results. In fact, the use of such a cache is required to reduce the complexity class of many BDD operations from exponential time to polynomial time. In the example, this cache is consulted at line 4 and the result is written at line 10. In cases where computing the result for leaves or other cases takes a significant amount of time, the cache should be consulted first. Often, the parameters can be normalized in some way to increase the cache efficiency. For example, $a \wedge b$ and $b \wedge a$ are the same operation. In that case, normalization rules can rewrite the parameters to some standard form in order to increase cache utilization, at line 3. A well-known example is the if-then-else algorithm, which rewrites using rewrite rules called "standard triples" as described in [BRB90].

If x and y are not leaves and the operation is not trivial or in the cache, we use a function `topvar` (line 5) to determine the first variable of the root nodes of x and y. If x and y have a different variable in their root node, `topvar` returns the first one in the variable ordering of x and y. We then compute the recursive application of F to the cofactors of x and y with respect to variable v in lines 7–8. We write $x_{v=i}$ to denote the cofactor of x where variable v takes value i. Since x and y are ordered according to the same fixed variable ordering, we can easily obtain $x_{v=i}$. If the root node of x is on the variable v, then $x_{v=i}$ is obtained by following the low ($i = 0$) or high ($i = 1$) edge of x. Otherwise, $x_{v=i}$ equals x. After computing the suboperations, we compute the result by either reusing an existing or creating a new BDD node (line 9). This is done by a function `lookupBDDnode` which, given a variable v and the BDDs of result$_{v=0}$ and result$_{v=1}$, returns the BDD for result. See also Section 2.3.1.

Operations on decision diagrams are typically recursively defined on the structure of the inputs. To parallelize the operation in Algorithm 2.1, the two independent suboperations at lines 7–8 are executed in parallel. This type of parallelism is called *task-based parallelism*. A popular method to efficiently execute a task-based program in parallel and distribute the tasks among the available processors is called work-stealing, which we discuss in Chapter 3. In most BDD algorithms, a significant amount of time is spent in memory operations: accessing the cache and performing lookups in the unique table. To obtain a good speedup, it is vital that these two data structures are scalable. We discuss them in Section 2.2.3, Section 2.2.4 and Chapter 4.

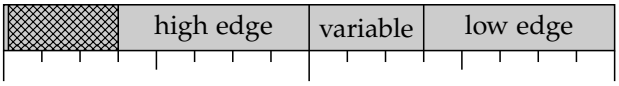
2.2.2 Representation of nodes

This subsection discusses how BDD nodes, LDD nodes and MTBDD nodes are represented in memory. We use 16 bytes for all types of nodes, so we can use the same unique table for all nodes and have a fixed node size. As we see below, not all bits are needed; unused bits are set to 0. Also, with 16 bytes per node, this means that 4 nodes fit exactly in a cacheline of 64 bytes (the size of the cacheline for many current computer architectures, in particular the x86 family that we use), which is very important for performance. If the unique table is properly aligned in memory, then only one cacheline needs to be accessed when accessing a node.

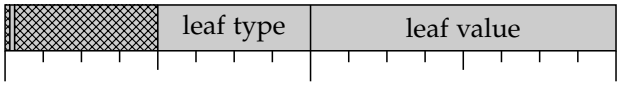
We use 40 bits to store the index of a node in the unique table. This is sufficient to store up to 2^{40} nodes, i.e. 16 terabytes of nodes, excluding overhead in the hash table (to store all the hashes) and other data structures. As we see below, there is sufficient space in the nodes to increase this to 48 bits per node (up to 4096 terabytes), although that would have implications for the performance (more difficult bit operations) and for the design of the operation cache.

Edges to nodes Sylvan defines the type BDD as a 64-bit integer, representing an edge to a BDD node. The lowest 40 bits represent the location of the BDD node in the nodes table, and the highest-significant bit stores the complement mark [BRB90]. The BDD 0 is reserved for the leaf `false`, with the complemented edge to 0 (i.e. `0x8000000000000000`) meaning `true`. We use the same method for MTBDDs and LDDs, although most MTBDDs do not have complemented edges. LDDs do not have complemented edges at all. The LDD leaf `false` is represented as 0, and the LDD leaf `true` is represented as 1. For the MTBDD leaf $\perp$ we use the leaf 0 that represents Boolean `false` as well. This has the advantage that Boolean MTBDDs can act as filters for MTBDDs with the MTBDD operation `times`. The disadvantage is that partial Boolean MTBDDs are not supported by default, but can easily be implemented using a custom MTBDD leaf.

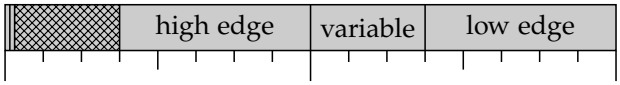
Internal BDD nodes Internal BDD nodes store the variable label (24 bits), the low edge (40 bits), the high edge (40 bits), and the complement bit of the high edge (1 bit, the first bit below).



MTBDD leaves For MTBDDs we use a bit to indicate whether a node is a leaf or not. MTBDD leaves store the leaf type (32 bits), the leaf contents (64 bits) and the fact that they are a leaf (1 bit, set to 1):

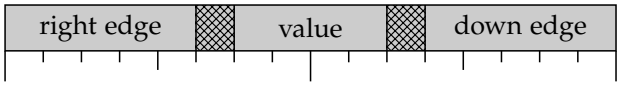


Internal MTBDD nodes Internal MTBDD nodes store the variable label (24 bits), the low edge (40 bits), the high edge (40 bits), the complement bit of the high edge (1 bit, the first bit below) and the fact they are not a leaf (1 bit, the second bit below, set to 0).



Internal BDD nodes are identical to internal MTBDD nodes, as unused bits are set to 0. Hence, the BDD 0 can be used as a terminal for Boolean MTBDDs, and the resulting Boolean MTBDD is identical to a BDD of the same function.

Internal LDD nodes Internal LDD nodes store the value (32 bits), the down edge (40 bits) and the right edge (40 bits):



2.2.3 Unique table

The unique table stores all decision diagram nodes and is essential to avoid duplicate nodes. This table is typically implemented as a hash table, in particular because the `find-or-insert` operations is performed in time O(1) on average (amortized) by a hash table. In general, the unique table can be either one shared table, or be split in multiple parts somehow. For example, Somenzi [Som01] argues for a subtable for each variable level, as this makes the implementation of variable reordering easier. The disadvantage of subtables is that their sizes must be adjusted dynamically, thus requiring the different parallel processes to cooperate on performing garbage collection and resizing when subtables are full. In addition, there is some overhead to compute the correct sizes for each table, which can be avoided by using a single table. Finally, subtables require the additional complexity of decreasing subtable sizes and compressing decision diagrams, which we avoid using a single table that only increases in size when this is needed.

In the past, there have been various proposals to split the unique table in several parts for parallel applications, for example to assign parts of the decision diagrams to certain processors or workstations. This is a consideration that can be orthogonal to parallelism. As we use work-stealing to perform the load balancing of the decision diagram operations, we have no control over which processor performs specific operations. Therefore, we use a single continuous block of memory, and we let the operating system take care of allocating memory blocks on all available memories in the system.

Manipulating BDDs with a hash table typically results in random access patterns in the memory. We somewhat improve on random access in Chapter 4 by letting each worker add new nodes in the "data" part of the hash table in a consecutive order, so that nodes that are related can often be found close to each other.

The unique table essentially requires two operations:

- a `find-or-insert` method that, given a 16-byte node, either finds the existing node in the table, or creates a new node.
- a method to delete nodes for garbage collection. Our implementation has a separate "data array" containing the nodes and a "hash array" containing the metadata. We require three operations:
  - `clear` removes all entries from the hash array;
  - `mark` marks a given node for reinsertion in the hash array; and
  - `rehash` reinserts a given node in the hash array.

These operations need to be highly scalable. The unique table and the implementation of `find-or-insert` are further described in Chapter 4, while parallel garbage collection is discussed in Section 2.2.5 below.

2.2.4 Computed table

Similar to the unique table, we use only one shared operation cache for all operations, because we want to minimize interaction between workers, such as synchronization when shared parts of memory are resized.

In [Som01], Somenzi writes that a lossless computed table guarantees polynomial cost for the basic synthesis operations, but that lossless tables (that do not throw away results) are not feasible when manipulating many large BDDs and in practice lossy computed tables (that may throw away results) are implemented. If the cost of recomputing subresults is sufficiently small, it can pay off to regularly delete results or even prefer to sometimes skip the cache to avoid data races. As we discuss in Chapter 4, we design the operation cache to abort operations as fast as possible when there may be a data race or the data may already be in the cache.

On top of this, our BDD implementation implements *caching granularity*, which controls when results are cached. Most BDD operations compute a result on a variable x_i, which is the top variable of the inputs. For granularity G, a variable x_i is in the cache block $i \bmod G$. Then each BDD suboperation only uses the cache once for each cache block, by comparing the cache block of the parent operation and of the current operation.

This is a deterministic method to use the operation cache only sometimes rather than always. In practice, we see that this technique improves the performance of BDD operations. If the granularity G is too large, the cost of recomputing results becomes too high, though, so care must be taken to keep G at a reasonable value.

2.2.5 Garbage collection framework

Operations on decision diagrams typically create many new nodes and discard old nodes. Nodes that are no longer referenced are typically called "dead nodes". Garbage collection, which removes dead nodes from the unique table, is essential for the implementation of decision diagrams. Since dead nodes are often reused in later operations, garbage collection should be delayed as long as possible [Som01].

There are various approaches to garbage collection. For example, a *reference count* could be added to each node, which records how often a node is referenced. Nodes with a reference count of zero are either immediately removed when the count decreases to zero, or during a separate garbage collection phase. Another approach is *mark-and-sweep*, which marks all nodes that should be kept and removes all unmarked nodes. We refer to [Som01] for a more in-depth discussion of garbage collection.

For a parallel implementation, reference counts can incur a significant cost, as accessing nodes implies continuously updating the reference count, increasing the amount of communication between processors, as writing to a location in memory requires all other processors to refresh their view on that location. This is not a severe issue when there is only one processor, but with many processors this results in excessive communication, especially for nodes that are commonly used.

When parallelizing decision diagram operations, we can choose to perform garbage collection "on-the-fly", allowing other workers to continue inserting nodes, or we can "stop-the-world" and have all workers cooperate on garbage collection. We use a separate garbage collection phase, during which no new nodes are inserted. This greatly simplifies the design of the hash table, and we see no major advantage to allow some workers to continue inserting nodes during garbage collection.

Some decision diagram implementations maintain a counter that counts how many buckets in the nodes table are in use and triggers garbage collection when a certain percentage of the table is in use. We want to avoid global counters like this and instead use a bounded "probe sequence" (see Chapter 4) for the nodes table: when the algorithm cannot find an empty bucket in the first K buckets, garbage collection is triggered. In simulations and experiments, we find that this occurs when the hash table is between 80% and 95% full.

As described in Chapter 4, decision diagram nodes are stored in a "data array", separated from the metadata of the unique table, which is stored in the "hash array". Nodes can be removed from the hash table without deleting them from the data array, simply by clearing the hash array. The nodes can then be reinserted during garbage collection, without changing their location in the data array, thus preserving the identity of the nodes.

We use a mark-and-sweep approach, where we keep track of all nodes that must be kept during garbage collection. Our implementation of parallel garbage collection consists of the following steps:

1. Initiate the operation using the Lace framework (Chapter 3) to arrange the "stop-the-world" interruption of all ongoing tasks.
2. Clear the hash array of the unique table, and clear the operation cache. The operation cache is cleared instead of checking each entry individually after garbage collection, although that is also possible.
3. Mark all nodes that we want to keep, using various mechanisms that keep track of the decision diagram nodes that we want to keep (see below).
4. Count the number of kept nodes and optionally increase the size of the unique table. Also optionally change the size of the operation cache.
5. Rehash all marked nodes in the hash array of the unique table.

To mark all used nodes, Sylvan has a framework that allows custom mechanisms for keeping track of used nodes. During the "marking" step of garbage collection, the marking callback of each mechanism is called and all used decision diagram nodes are recursively marked. Sylvan itself implements four such mechanisms (also for MTBDDs and LDDs):

- The `sylvan_protect` and `sylvan_unprotect` methods maintain a set of BDD pointers in a separate hash table. During garbage collection, each pointer is inspected and the BDD nodes are recursively marked.
- Each thread has a thread-local BDD stack, where BDD operations store intermediate results, with macros `bdd_refs_push` and `bdd_refs_pop`.
- Each thread has a thread-local stack of created suboperations (tasks of the Lace framework, see Chapter 3). Suboperations that return BDDs are stored in this stack (method `bdd_refs_spawn`), and during garbage

collection the results of finished suboperations are marked. The method `bdd_refs_sync` removes the suboperation from the stack.

- The `sylvan_ref` and `sylvan_deref` methods add and remove specific BDD nodes from a collection stored in a separate hash table. These methods exist because other BDD packages also implement them, but in practice, `sylvan_protect` and `sylvan_unprotect` are more useful.

To initiate garbage collection, we use a feature in the Lace framework (introduced in Chapter 3) that suspends all current work and starts a new task tree. This task suspension is a cooperative mechanism. Workers regularly check whether the current task tree is being suspended, either explicitly by calling a method from the parallel framework, or implicitly when creating or synchronizing on tasks. Implementations of BDD operations make sure that all used BDDs are accounted for, typically with `bdd_refs_push` and `bdd_refs_spawn`, before such checks.

The garbage collection process itself is also executed in parallel. Removing all nodes from the hash table and clearing the operation cache is an instant operation that is amortized over time by the operating system by reallocating the memory (see below). Marking nodes that must be kept occurs in parallel, mainly by implementing the marking operation as a recursive task using the Lace framework for parallelism descibed in Chapter 3. Counting the number of used nodes and rehashing all nodes (steps 4–5) is also parallelized using a standard binary divide-and-conquer approach, which distributes the memory pages over all workers.

Memory management Memory in modern computers is divided into regions called pages that are typically (but not always) 4096 bytes in size. Furthermore, computers have a distinction between "virtual" memory and "real" memory. It is possible to allocate much more virtual memory than we really use. The operating system is responsible for assigning real pages to virtual pages and clearing memory pages (to zero) when they are first used.

We use this feature to implement resizing of our unique table and operation cache. We preallocate memory according to a maximum number of buckets. Via global variables `table_size` and `max_size` we control which part of the allocated memory is actually used. When the table is resized, we simply change the value of `table_size`. To free pages, the kernel can be advised to free real pages using a `madvise` call (in Linux), but Sylvan only implements increasing the size of the tables, not decreasing their size.

Furthermore, when performing garbage collection, we clear the operation cache and the hash array of the unique table by reallocating the memory. Then, the actual clearing of the used pages only occurs *on demand* by the operating system, when new information is written to the tables.

```
1 def lookupBDDnode(v, low, high):
2     if low = high : return low
3     if comp(low) : return ¬lookupBDDnode(v, ¬low, ¬high)
4     try:
5         return find-or-insert ({v, idx(low), comp(high), idx(high)})
6     catch TableFull :
7         garbage-collect()
8         return find-or-insert ({v, idx(low), comp(high), idx(high)})
```

Algorithm 2.2 The `lookupBDDnode` method is given a variable v and two BDDs low and high and assumes that all variables in the BDDs low and high are ordered after the given variable v. `lookupBDDnode` returns the unique BDD representing the Boolean function "if v then high else low".

2.3 BDD algorithms

This section describes the BDD algorithms that we implemented in Sylvan.

2.3.1 Creating and reading BDD nodes

To create a BDD node, we use the `lookupBDDnode` method given in Algorithm 2.2. This method returns the BDD (edge to a BDD node) for the Boolean function "if v then `high` else `low`". The method ensures that the results of BDD operations are canonical reduced BDDs. An important assumption for this method is that the given BDDs `low` and `high` only contain variables that are after the given variable v in the variable ordering; the `ite` operation (Section 2.3.2) does not have this restriction.

The `lookup` algorithm first handles redundant nodes (line 2) and checks the complement edge rule (line 3), ensuring that there is never a complement on the low edge. We use $\neg$ to denote toggling the complement edge bit of a BDD. The nodes table method `find-or-insert` is called, which ensures that there are no duplicate nodes (lines 5 and 8). We use `comp(bdd)` and `idx(bdd)` to denote the complement bit and the index of the given BDD, as described in Section 2.2.2. If the nodes table is full (line 6), then garbage collection is performed (line 7) and the method `find-or-insert` is called again. If this second invocation of `find-or-insert` also raises the TableFull exception, then the exception is passed to the caller.

To directly read BDD nodes, methods `var`, `low` and `high` are available. The `low` and `high` methods also check if the complement bit of the input BDD is set, and if so, toggle the complement bit of the result. A method `topvar`, given a list of BDDs, returns the first variable in the BDDs according to the variable ordering, by inspecting the root node of the BDDs. For sets of variables, which

Operation	Implementation
$x \wedge y$	$\texttt{and}(x, y)$
$x \vee y$	$\texttt{not}(\texttt{and}(\texttt{not}(x), \texttt{not}(y)))$
$\neg(x \wedge y)$	$\texttt{not}(\texttt{and}(x, y))$
$\neg(x \vee y)$	$\texttt{and}(\texttt{not}(x), \texttt{not}(y))$
$x \oplus y$	$\texttt{xor}(x, y)$
$x \leftrightarrow y$	$\texttt{not}(\texttt{xor}(x, y))$
$x \rightarrow y$	$\texttt{not}(\texttt{and}(x, \texttt{not}(y)))$
$x \leftarrow y$	$\texttt{not}(\texttt{and}(\texttt{not}(x), y))$
if x then y else z	$\texttt{ite}(x, y, z)$
$\exists v\colon x$	$\texttt{exists}(x, v)$
$\forall v\colon x$	$\texttt{not}(\texttt{exists}(\texttt{not}(x), v))$

Table 2.1 Basic BDD operations on the input BDDs x, y, z.

```
1  def and(x, y):
2      if x = 1 : return y
3      if y = 1 ∨ x = y : return x
4      if x = 0 ∨ y = 0 ∨ x = ¬y : return 0
5      if result ← cache[(x, y)] : return result
6      v = topvar(x,y)
7      do in parallel:
8          low ← and(x_{v=0}, y_{v=0})
9          high ← and(x_{v=1}, y_{v=1})
10     result ← lookupBDDnode(v, low, high)
11     cache[(x, y)] ← result
12     return result
```

Algorithm 2.3 Parallelized BDD algorithm `and`, with as parameters the BDDs x and y. The result is a BDD representing $x \wedge y$.

are represented as conjunctions like $x \wedge y \wedge z$, we use `next` for `high`, which obtains $y \wedge z$ in this example if x is the first in the variable ordering.

2.3.2 Basic operations

For binary decision diagrams, Sylvan implements the basic BDD operations (Table 2.1) `and`, `not` and `xor`, the if-then-else (`ite`) operation, and `exists`. Implementing the basic operations in this way is common for BDD packages. Negation $\neg$ is performed using complement edges, see Section 2.1.2, and is basically free.

The parallelization of these functions is straightforward. See Algorithm 2.3

```
1  def exists(x, V):
2      if x = 0 ∨ x = 1 ∨ V = ∅ : return x
3      v = var(x)
4      while V ≠ ∅ ∧ var(V) < v :  V ← next(V)
5      if V = ∅ : return x
6      if result ← cache[(x, V)] : return result
7      if v = var(V) :
8          if x_{v=0} = 1 ∨ x_{v=1} = 1 ∨ x_{v=0} = ¬x_{v=1} :  result ← 1
9          else:
10             low ← exists(x_{v=0}, next(V))
11             if low = 1 :  result ← 1
12             else:
13                 high ← exists(x_{v=1}, next(V))
14                 result ← or(low, high)
15     else:
16         do in parallel:
17             low ← exists(x_{v=0}, V)
18             high ← exists(x_{v=1}, V)
19         result ← lookupBDDnode(v, low, high)
20     cache[(x, V)] ← result
21     return result
```

Algorithm 2.4 Parallelized BDD algorithm `exists`, with the BDD x and V the cube of variables that are abstracted via existential quantification.

for the parallel implementation of `and`. This algorithm checks the trivial cases (lines 2–4) before the operation cache (line 5), and then runs the two independent suboperations (lines 8–9) in parallel.

A more involved example is the parallelized algorithm `exists` (Algorithm 2.4) which computes existential quantification. This algorithm receives the input parameters x and V, where x is the BDD representing the function to which quantification is applied, and V is the BDD representing the conjunction of the variables that are abstracted away from x. After the trivial cases (line 2), we check if V actually contains variables that are in the BDD (lines 3–5), exploiting the fact that V is also an ordered BDD. This is also a normalization step for the cache, which is checked at line 6. Now, there are two cases: either the current root variable v is in V (lines 7–14) or it is not in V (lines 15–19). In the second case, we simply perform the two suboperations in parallel and compute the result. In the first case, after checking some trivial cases, we can either 1) perform the two suboperations in parallel; 2) perform the "low" suboperation first; or 3) perform the "high" suboperation first. If either of these suboperations returns 1, then the other does not need to be computed. The

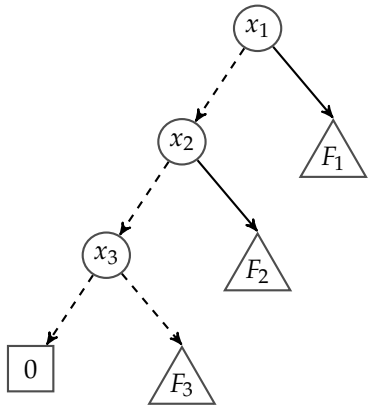


Figure 2.4 An example of a BDDMap encoding $[x_1 := F_1, x_2 := F_2, x_3 := F_3]$. The triangles F_1, F_2 and F_3 represent BDDs.

```
1  def compose(x, M):
2      if x = 0 ∨ x = 1 ∨ M = 0 : return x
3      v = var(x)
4      while M ≠ 0 ∧ var(M) < v :  M ← low(M)
5      if M = 0 : return x
6      if result ← cache[(x, M)] : return result
7      do in parallel:
8          low ← compose(low(x), M)
9          high ← compose(high(x), M)
10     if v = var(M) :  result ← ite(high(M), high, low)
11     else:  result ← lookupBDDnode(v, low, high)
12     cache[(x, M)] ← result
13     return result
```

Algorithm 2.5 Apply functional composition $x[M]$, where M is a mapping from variables to Boolean functions.

advantage of option 1 is that there is more opportunity for parallelization, at the cost of possible extra work. However, this extra independent work might not be necessary, since there is already a lot of independent work from the parallelization at lines 17–18 and inside the `or` operation. In Algorithm 2.4, we compute the "low" suboperation first.

Another operation that is parallelized similarly is the `compose` operation, which performs functional composition, i.e., substitute occurrences of variables in a Boolean formula by Boolean functions. For example, the substitution

```
1  def and_exists(x, y, V):
2      if x = 0 ∨ y = 0 ∨ x = ¬y : return 0
3      if x = 1 ∧ y = 1 : return 1
4      if x = 1 : return exists (y,V)
5      if y = 1 ∨ x = y : return exists (x,V)
6      v = topvar(x,y)
7      while V ≠ ∅ ∧ var(V) < v :  V ← next(V)
8      if V = ∅ : return and (x,y)
9      if result ← cache[(x, y, V)] : return result
10     do in parallel:
11         low ← and_exists(x_{v=0}, y_{v=0}, V)
12         high ← and_exists(x_{v=1}, y_{v=1}, V)
13     if v = var(V) : result ← or(low, high)
14     else: result ← lookupBDDnode(v, low, high)
15     cache[(x, y, V)] ← result
16     return result
```

Algorithm 2.6 The parallel BDD algorithm `and_exists`, which given BDDs x and y, and V a cube of variables, computes $\exists V : (x \wedge y)$.

$[x_1 := x_2 \vee x_3, x_2 := x_4 \vee x_5]$ applied to the function $x_1 \wedge x_2$ results in the function $(x_2 \vee x_3) \wedge (x_4 \vee x_5)$. Sylvan offers a functional composition algorithm based on a "BDDMap". This structure is not a BDD itself, but uses BDD nodes to encode a mapping from variables to BDDs. A BDDMap is based on a disjunction of variables, but with the "high" edges going to BDDs instead of the terminal 1. See Figure 2.4 for an example. This method also implements substution of variables, e.g. $[x_1 := x_2, x_2 := x_3]$. See Algorithm 2.5 for the algorithm `compose`. This parallel algorithm is similar to the algorithms described above, with the composition functionality at lines 10–11. If the variable is in the mapping M, then we use the `if-then-else` method to compute the substitution. If the variable is not in the mapping M, then we simply compute the result using `lookupBDDnode`.

Sylvan also implements parallelized versions of the BDD minimization algorithms `restrict` and `constrain` (also called generalized cofactor), based on sibling-substitution, which are described in [CM90] and parallelized similarly as the `and` algorithm above.

2.3.3 Relational products

In model checking using decision diagrams, relational products play a central role. Relational products compute the successors or the predecessors of (sets of) states. Typically, states are encoded using Boolean variables $\vec{x} = x_1, x_2, \ldots, x_N$.

```
1  def relnext(S, R, V):
2      if S = 0 ∨ R = 0 : return 0
3      if S = 1 ∧ R = 1 : return 1
4      v = topvar(S,R)
5      while var(V) < v : V ← next(V)
       // if V = ∅, we assume R is irrelevant
6      if V = ∅ : return S
7      if result ← cache[(S, R, V)] : return result
8      if v = var(V) :
9          x, x' ← unprimed v, primed v
10         V' ← V without x and x'
11         do in parallel:
12             a ← relnext(S_{x=0}, R_{x=0,x'=0}, V')
13             b ← relnext(S_{x=1}, R_{x=1,x'=0}, V')
14             c ← relnext(S_{x=0}, R_{x=0,x'=1}, V')
15             d ← relnext(S_{x=1}, R_{x=1,x'=1}, V')
16         do in parallel:
17             low ← or(a, b)
18             high ← or(c, d)
19         result ← lookupBDDnode(x, low, high)
20     else:
           // v is not in R, by assumption
21         do in parallel:
22             low ← relnext(S_{v=0}, R, V)
23             high ← relnext(S_{v=1}, R, V)
24         result ← lookupBDDnode(v, low, high)
25     cache[(S, R, V)] ← result
26     return result
```

Algorithm 2.7 The parallel algorithm `relnext`, which given the BDDs S (representing a set of states), R (representing a transition relation) and V (the cube of interleaved variables $\vec{x} \cup \vec{x}'$) computes the set of successor states defined on $\vec{x}$, i.e., $\big(\exists \vec{x}\colon (S \wedge R)\big)[\vec{x}' := \vec{x}]$. We assume that all variables in R are also in V.

Transitions between these states are represented using Boolean variables $\vec{x}$ for the source states and variables $\vec{x}' = x'_1, x'_2, \ldots, x'_N$ for the target states. Given a set of states S_i encoded as a BDD on variables $\vec{x}$, and a transition relation R encoded as a BDD on variables $\vec{x} \cup \vec{x}'$, the set of states S'_{i+1} encoded on variables $\vec{x}'$ is obtained by computing $S'_{i+1} = \exists \vec{x}\colon (S_i \wedge R)$. BDD packages typically implement an operation `and_exists` that combines $\exists$ and $\wedge$ to compute S'_{i+1}. See Algorithm 2.6 for the parallel version of `and_exists` in Sylvan.

Typically we want the BDD of the successors states defined on the unprimed

variables $\vec{x}$ instead of the primed variables $\vec{x}'$, so the `and_exists` call is then followed by a variable substitution that replaces all occurrences of variables from $\vec{x}'$ by the corresponding variables from $\vec{x}$. Furthermore, the variables are typically interleaved in the variable ordering, like $x_1, x'_1, x_2, x'_2, \ldots, x_N, x'_N$, as this often results in smaller BDDs. Sylvan implements specialized operations `relnext` and `relprev` that compute the successors and the predecessors of sets of states, where the transition relation is encoded with the interleaved variable ordering. See Algorithm 2.7 for the implementation of `relnext`. This function takes as input a set S, a transition relation R, and the set of variables V, which is the union of the interleaved sets $\vec{x}$ and $\vec{x}'$ (the variables on which the transition relation is defined). We first check for terminal cases (lines 2–3). These are the same cases as for the $\wedge$ operation. Then we process the set of variables V to skip variables that are not in S and R (lines 5–6). After consulting the cache (line 7), either the current variable is in the transition relation, or it is not. If it is not, we perform the usual recursive calls and compute the result (lines 21–24). If the current variable is in the transition relation, then we let x and x' be the two relevant variables (either of these equals v) and compute four subresults, namely for the transition (a) from 0 to 0, (b) from 1 to 0, (c) from 0 to 1, and (d) from 1 to 1 in parallel (lines 11–15). We then abstract from x' by computing the existential quantifications in parallel (lines 16–18), and finally compute the result (line 19). This result is stored in the cache (line 25) and returned (line 26). We implement `relprev` similarly.

2.4 MTBDD algorithms

Although multi-terminal binary decision diagrams are often used to represent functions to integers or real numbers, they could be used to represent functions to any domain. In practice, the well-known BDD package CUDD [Som15] implements MTBDDs with `double` (floating-point) leaves. For some applications, other types of leaves are required, for example to represent rational numbers or integers. To allow different types of MTBDDs, we designed a generic customizable framework. The idea is that anyone can use the given functionality or extend it with other leaf types or other operations.

By default, Sylvan implements five types of leaves:

Leaf type	**Function type**
BDDs `false` and `true` (¬`false`)	total functions $\mathbb{B}^N \to \mathbb{B}$ (BDDs)
64-bit integer (`uint64`)	partial functions $\mathbb{B}^N \to \mathbb{N}$
floating-point (`double`)	partial functions $\mathbb{B}^N \to \mathbb{R}$
rational leaves (`int32` + `uint32`)	partial functions $\mathbb{B}^N \to \mathbb{Q}$
GMP library leaves (`mpq`)	partial functions $\mathbb{B}^N \to \mathbb{Q}$

```
1  def abstract(x, V, F):
2      if x = 0 ∨ x = 1 ∨ V = ∅ : return x
3      if result ← cache[(x, V, F)] : return result
4      if x is a leaf or var(V) < topvar(x) :
5          sub ← abstract(x, next(V), F)
6          result ← F(sub, sub)
7      elif var(V) = topvar(x) :
8          do in parallel:
9              low ← abstract(x_{v=0}, next(V), F)
10             high ← abstract(x_{v=1}, next(V), F)
11         result ← F(low, high)
12     else:
13         do in parallel:
14             low ← abstract(x_{v=0}, V, F)
15             high ← abstract(x_{v=1}, V, F)
16         result ← lookupMTBDDnode(v, low, high)
17     cache[(x, V, F)] ← result
18     return result
```

Algorithm 2.8 Parallel MTBDD algorithm that applies the abstraction F for the variables in V.

The BDDs `false` and `true` (complemented `false`) are not encoded as MTBDD leaves as in Section 2.2.2, but we reuse the BDD 0 that is reserved for the leaf `false`. For the rational leaves we use 32 bits for the numerator and 32 bits for the denominator. Sylvan also implements the leaf type `mpq` which uses the GMP library for arbitrary precision arithmetic, i.e., an arbitrary number of bits for the numerator and the denominator. The framework supports partially defined functions, reusing the BDD `false` to mean $\perp$ for non-Boolean functions.

Sylvan implements a generic binary apply function, a generic monadic apply function, and a generic abstraction algorithm. The implementation of binary apply is similar to Algorithm 2.1. See Algorithm 2.8 for the implementation of abstraction.

On top of these generic algorithms, we implemented basic operators `plus`, `times`, `min` and `max` for the default leaf types. For all valuations of MTBDDs x and y that end in leaves a and b, they compute $a + b$, $a \times b$, $\min(a, b)$ and $\max(a, b)$. For Boolean MTBDDs, the `plus` and `times` operators are similar to $\vee$ and $\wedge$. In fact, when using `times` with a Boolean MTBDD (or a BDD) and an MTBDD of some other type, it acts as a filter, removing the subgraphs where the BDD is `false` and keeping the subgraphs where the BDD is `true`.

Sylvan supports custom leaves with 64-bit values. These 64-bit values can also be pointers. In that case, for the canonical representation of leaves it is

not sufficient to compare the 64-bit values, which is the default behavior. Also, the pointers typically point to dynamically allocated memory that must be freed when the leaf is deleted. To support custom leaves, Sylvan implements a framework where custom callbacks are registered for each leaf type. These custom callbacks are:

- `hash(value, seed)` computes a 64-bit hash for the leaf value and the given 64-bit seed.
- `equals(value1, value2)` returns 1 if the two values encode the same leaf, and 0 otherwise. The default implementation simply compares the two values.
- `create(pointer)` is called when a new leaf is created with the 64-bit value references by the pointer; this allows implementations to allocate memory and replace the referenced value with the final value.
- `destroy(value)` is called when the leaf is garbage collected so the implementation can free memory allocated by `create`.

We use this functionality to implement the GMP leaf type. The GMP leaf type is essentially a pointer to a different data structure to support arbitrary precision arithmetic. The above functions are implemented as follows:

- `hash` follows the pointer and hashes the contents of the `mpq` data structure.
- `equals` follows the two pointers and compares their contents.
- `create` clones the `mpq` data structure and writes the address of the clone to the given new leaf.
- `destroy` frees the memory of the cloned data structure.

2.5 LDD algorithms

We implemented various LDD operations that are required for model checking in LTSmin (see Chapter 5). As LDDs are primarily used to represent sets of state vectors (consisting of integers) and transitions, we do not provide the pseudocode of these operations, but specify their functionality on sets of state vectors and transitions. In LTSmin, the notion of a projection of a set onto certain variables means abstracting away from variables that we are not interested in, to obtain states on only the given variables. This is similar to existential quantification. Many operations work with these projections, also called "short vectors". We often use "helper LDDs" typically named `meta`, which describe (encoded as a tuple in an LDD) how a set of states or transitions are related to the "long vector". There are several operations that combine other operations. This is good for performance, as potentially expensive intermediate results do not need to be computed.

Operation	Informal semantics
`union(A,B)`	compute $A \cup B$
`minus(A,B)`	compute $A \setminus B$
`zip(A,B)`	compute X, Y such that $X = A \cup B, Y = B \setminus A$
`project(A,p)`	compute the projection of the set A according to the projection vector `p`
`project_minus(A,p,B)`	compute `minus(project(A,p),B)`
`intersect(A,B)`	compute $A \cap B$
`match(A,B,meta)`	compute $A \cap B$, but B is defined on subset of the variables of A according to `meta`
`join(A,B,pA,pB)`	compute $A \cap B$, but A and B are projections of the state vector, described by `pA` and `pB`
`cube(v,k)`	compute the LDD representing the singleton set containing the k-tuple with the values in `v`
`member_cube(A,v,k)`	check if `cube(v,k)` is in the set A
`union_cube(A,v,k)`	compute `union(A,cube(v,k))`
`relprod(A,B,meta)`	compute the successors of the states in A according to the transition relation B which is described by `meta`
`relprod_union(A,B,m)`	compute `union(A,relprod(A,B,m))`
`relprev(A,B,meta,U)`	compute the predecessors of the states in A according to the transition relation B which is described by `meta`, restricted to states in U.
`satcount(A)`	compute the size of the set A
`sat_one(A)`	return one vector from the set A
`sat_all(A,cb)`	call the callback `cb` for each vector in A
`sat_all_par(A,cb)`	`sat_all`, but parallelized
`collect(A,cb)`	`sat_all_par`, but the callback returns some set encoded as an LDD, and all returned LDDs are combined using `union`
`match_sat(A,B,m,cb)`	compute `sat_all_par(match(A,B,m),cb)`

An example of a parallelized LDD operation is the implementation `union` in Algorithm 2.9. After the trivial cases, there are three cases left. Only the case where A and B both point to an LDD node with the same value gives opportunity for parallelization. Computing the new `down` and `right` LDDs are independent tasks and can be performed in parallel.

```
1  def union(A, B):
2      if (A = 1 ∧ B = 1) ∨ A = B : return A
3      if A = 0 : return B
4      if B = 0 : return A
5      if result ← cache[(A, B)] :  return result
6      if value(A) < value(B) :
7          right ← union(right(A), B)
8          result ← lookupLDDnode(value(A),down(A),right)
9      elif value(A) > value(B) :
10         right ← union(A, right(B))
11         result ← lookupLDDnode(value(B),down(B),right)
12     else:
13         do in parallel:
14             right ← union(right(A), right(B))
15             down ← union(down(A), down(B))
16         result ← lookupLDDnode(value(A),down,right)
17     cache[(A, B)] ← result
18     return result
```

Algorithm 2.9 Parallel LDD algorithm `union` that computes $A \cup B$.

Chapter 3

Load-balancing tasks with Lace

As outlined in Chapter 2, to efficiently parallelize decision diagram operations, we need to execute recursive subtasks in parallel. The tasks in decision diagram operations are extremely small, so the load balancing of these tasks is not only important for parallelism, but may result in a relatively heavy overhead if not efficient. Additionally, depending on the structure of the decision diagrams, the computations can be quite irregular, making load balancing challenging. The operations are also memory-intensive, as they perform few computations and rely mainly on accessing the operation cache and the nodes table. Thus communication for load balancing is even more costly with memory-intensive tasks than with computation-heavy tasks.

Fortunately, a well-known algorithm to effectively execute task-based algorithms like decision diagram operations in parallel is work-stealing [Blu94]. In work-stealing, there are as many workers as there are available processor cores, and each worker owns a local queue in which it stores new tasks. Workers without tasks (idle workers) steal and execute tasks from other workers. In particular, the implementation of work-stealing in the framework Wool [Fax10] has been shown to have good performance for fine-grained (small) tasks [PBF10].

The performance of work-stealing strongly depends on the implementation of the task queues. In the first published version of Sylvan [DLP13] we relied on the work-stealing framework Wool to parallelize BDD operations. We substituted this framework by Lace [DP14], in which we developed a novel work-stealing queue, aimed at minimizing interactions between different workers and the shared memory. In addition, we also developed features in Lace that are useful to implement stop-the-world garbage collection in Sylvan, where we halt the current set of tasks, and initiate a new task tree, suspending the old tasks until the new task tree is finished.

The novel work-stealing deque described in this chapter is a non-blocking work-stealing deque, based on a split task deque. The deque is split in two

parts, one of which is shared and one of which is private. Our design uses a dynamic split point between the shared and the private portions of the deque, and only requires memory fences when shrinking the shared portion. This circumvents the disadvantage of many concurrent deques, that they require expensive memory fences for local deque operations.

We evaluate the performance of Lace using several benchmarks, including standard Cilk benchmarks and the UTS benchmark [Oli+06]. We also compare our algorithm with Wool and with an implementation of the receiver-initiated private deque algorithm [ACR13] in the Lace framework.

This chapter is outlined as follows. We first discuss task-based parallelism and the work-stealing algorithm in Section 3.1. Section 3.2 gives an overview of existing implementations of work-stealing deques in the literature. We present a novel work-stealing deque in Section 3.3. Section 3.4 describes the algorithms of this deque, and in Section 3.5, we give an informal proof to show its correctness. We discuss details of the implementation of our work-stealing framework in Section 3.6, including a special feature to suspend current tasks and start a new task tree, which is very useful in Sylvan to perform garbage collection. Section 3.7 presents some empirical results on typical benchmarks from the literature. Finally, Section 3.8 concludes this chapter.

This chapter is based on the following publication:
[DP14] Tom van Dijk and Jaco van de Pol. "Lace: Non-blocking Split Deque for Work-Stealing." In: *MuCoCoS*. vol. 8806. LNCS. Springer, 2014, pp. 206–217

3.1 Task-based parallelism and work-stealing

The importance of parallel processing to improve the performance of software has become self-evident in recent years, especially given the availability of multi-core shared-memory systems and the physical limits that constrain processor speeds. Efficient parallel algorithms are now required to make effective use of the processing power of modern chips with multiple processors.

Task-based parallelism or nested control parallelism is a concept in parallel processing, where a computation is seen as a tree of tasks and dependencies that need to be scheduled on multiple processors. Tasks that are independent of each other can be executed in parallel.

For task parallelism that fits a "strict" fork-join model, i.e., each task creates the subtasks that it depends on, work-stealing is a well known framework for parallelism [Blu94], with implementations such as Cilk [Blu+96; FLR98] and Wool [Fax08; Fax10] that allow writing parallel programs in a style similar to sequential programs [ACR13]. It is well known that work-stealing is an effective task-based load balancing method. Work-stealing has been proven to be

```
1 def TASK_1(int, fib, int, k):
2     if k < 2 : return k
3     SPAWN(fib, k − 2)
4     SPAWN(fib, k − 1)
5     n ← SYNC
6     m ← SYNC
7     return n + m
8 fib10 ← CALL(fib, 10)
```

Algorithm 3.1 The Fibonacci algorithm as a simple example of task-based parallelism using SPAWN and SYNC statements.


```
1 do in parallel:
2     K ← F1(x, y, z)
3     L ← F2(a, b, c)
4     M ← F3(g, h)
```

```
1 SPAWN(F1, x, y, z)
2 SPAWN(F2, a, b, c)
3 M ← CALL(F3, g, h)
4 L ← SYNC
5 K ← SYNC
```

Figure 3.1 Running three tasks in parallel using SPAWN+SYNC and CALL.

optimal for a large class of problems and has tight memory and communication bounds [Blu94].

See Algorithm 3.1 for an example of a parallel program in the Cilk/Wool style. These and similar task-based parallel frameworks, such as the framework Lace that is the topic of this chapter, use keywords SPAWN and SYNC to expose parallelism. The SPAWN keyword creates a new task. Every SPAWN during the execution of the program must have a matching SYNC. The SYNC keyword matches with the last unmatched SPAWN, i.e., operating as if spawned tasks are stored on a stack. It waits until that task is completed and retrieves the result. In this chapter, we follow the semantics of Wool. In the original work-stealing papers and in Cilk, SYNC waits for all locally spawned subtasks, rather than the last unmatched subtask. Tasks are defined using a TASK_1 macro for tasks with 1 parameter, TASK_2 for tasks with 2 parameters, etc. Furthermore, there is also a CALL keyword that skips the task stack and immediately executes a task.

See also Figure 3.1 for an example of how tasks are typically executed in parallel. In this thesis, we use **do in parallel** to denote that tasks are executed in parallel, like in Figure 3.1.

In work-stealing, tasks are executed by a fixed number of workers, typically equal to the number of processor cores. Each worker owns a task pool into which it inserts new subtasks created by the task it currently executes. Idle workers steal tasks from the task pools of other workers. Worker are idle

Work-stealing operations	Task pool operations
`spawn`(task)	`push`(task)
`sync`	`peek`, `pop`
`steal-and-run`(victim)	`steal`

Table 3.1 Operations of the work-stealing algorithm and matching operations of the task pool of each worker.

either because they do not have any tasks to perform (e.g., at the start of a computation), or because all their tasks have been stolen and they have to wait for the result of the stolen tasks to continue the current task. Typically, one worker starts executing a root task and the other workers perform work-stealing to acquire subtasks.

One important aspect of the work-stealing algorithm is victim selection. For example in systems with hierarchy, e.g., a network of workstations, it might be useful to steal from local workers first before trying to steal from a remote worker. Another strategy would be to remember how much work other workers have after a steal attempt, and use this to select smart targets. In our implementation, workers with an empty task pool steal from random victims.

When synchronizing with a stolen task, a possible strategy for the victim is to steal from the thief until the stolen task is completed. By stealing back from the thief, a worker executes subtasks of the stolen task. This technique is called leapfrogging [WC93]. When stealing from random workers instead, the size of the task pool of each worker could grow beyond the size needed for complete sequential execution [Fax10], since stealing will build a new stack on top of the blocked join. Using leapfrogging rather than stealing from random workers thus limits the space requirement of the task pools to that of sequential execution, although in practice it is expensive to guarantee that the tasks that are stolen from the thief are really subtasks of the original task. It might be possible that the thief finished the original task and stole a different branch of the task tree after the victim checked the status of the stolen task. Our implementation also uses the leapfrogging strategy.

Another concern is which task(s) to steal. A simple algorithm is to steal the first unstolen task from the bottom of the stack. A variation could be to steal multiple tasks, or to steal a random task from anywhere in the stack. In our implementation, thieves steal the first unstolen task from the bottom of the stack.

See Table 3.1 for an overview of the work-stealing operations and how they match with operations on the task pool. The methods `spawn` and `sync` implement the keywords `SPAWN` and `SYNC`. The method `steal-and-run` tries

to steal a task from the given victim and, if successful, executes the task and communicates the result back to the owner of the task. The methods `push`, `peek`, `pop` and `steal` are implemented by the task pool:

- The `push`, `peek` and `pop` operations are only used by the owner of the stack, and the `steal` operation only by the thieves.
- The `push` operation puts a task on the stack.
- The `peek` operation fixes the status of the task at the top of the stack: either stolen or available as work. After `peek`, the top task, if not stolen, cannot be stolen until the next `push` (or if `peek` is called again).
- The `pop` operation removes the top-most task from the stack. Furthermore we assume that the task data remains in the task pool until overwritten by a `push` operation.
- The `steal` operation steals a task from the bottom of the stack, changing its status from available work to stolen work. Stolen tasks are kept on the stack so the results of tasks can be communicated back to the original owner of the task.

See Algorithm 3.2 for the implementation of the work-stealing operations. This implementation is fairly straight-forward in terms of task pool operations, as the difficult problems are solved by the implementation of the task pool. The `spawn` operation simply calls the `push` method of the task pool. The `sync` operation first determines whether the task at the top of the stack is stolen or not by calling `peek` (line 4). If it is not stolen, then we pop the task from the stack (which does not modify the task contents) and execute the task (lines 5–6). If the task is stolen, then we first wait until the thief field of the task is set (line 9). This field is set by the thief at line 15. Then, we steal tasks with the leapfrogging strategy (steal from the thief) until the original task is done (line 10), remove the task from the stack (line 11), which leaves the task contents and the result intact, and return the result (line 12). We use a special `pop-stolen` method at line 11 for optimization reasons discussed in Section 3.3. The `steal-and-run` method tries to steal a task from the chosen victim (line 14). If successful, the thief communicates its identity (line 15), executes the task (line 16), stores the result (line 17) and communicates that it is done (line 18). The worker thread is defined at lines 19–24: one worker executes the first task and the other workers try to steal and execute tasks from random victims. Once the root task is done, the flag `done` is set and all other workers can quit their loop.

In this implementation of work-stealing, we keep tasks on the stack and communicate the result via the task object on the stack. An alternative would be to store pointers to tasks instead of tasks on the stack, using only the operations `push` and `pop` to add and remove the pointers. In this case, task pools implement three operations `push`, `pop` and `steal`. This introduces an

```
1  def spawn(task):
2      push(task)
3  def sync():
4      res ← peek()
       // res is Work(task) or Stolen(task)
5      if res = Work(task) :
6          pop()
7          return task.execute()
8      else:
9          while task.thief = None : (loop)
10         while ¬ task.done :  steal-and-run (task.thief)
11         pop-stolen()
12         return task.result
13 def steal-and-run(victim):
14     if victim.steal() = Task(stolentask) :
15         stolentask.thief ← me
16         result ← stolentask.execute()
17         stolentask.result ← result
18         stolentask.done ← True
19 thread worker(id, roottask):
20     done ← False
21     if id = 0 :
22         roottask.execute()
23         done ← True
24     else: while done is False: steal-and-run(random victim)
```

Algorithm 3.2 The implementation of the work-stealing using leapfrogging when waiting for a stolen task to finish, i.e., steal from the thief.

extra indirection (and memory access) that can be avoided by directly storing tasks in the stack. The "direct task stack", that stores tasks in the stack, was proposed by Faxén and implemented in Wool [Fax08; Fax10]. See also below in Section 3.2 and Section 3.3.

3.2 Existing work-stealing deques

Task pools are commonly implemented using double-ended queues (deques) specialized for work-stealing. The first demonstrably efficient work-stealing scheduler for fully strict computations was presented in 1994 [Blu94] and its implementation in Cilk in 1996 [Blu+96]. One improvement of the original Cilk algorithm is the THE protocol in Cilk-5 [FLR98], which eliminates acquiring

the lock in `push` and in most executions of `pop`, but every `steal` still requires locking.

The first non-blocking work-stealing deque is the ABP algorithm, which uses a fixed-size array that might overflow [ABP01]. Two unbounded non-blocking deques were proposed, the deque by Hendler et al. based on linked lists of small arrays [Hen+06], and the Chase-Lev deque that uses dynamic circular arrays [CL05].

In weak memory models that allow reordering loads before stores, most deques that allow any spawned task to be stolen require a memory fence in every `pop` operation. Memory fences are expensive. For example, the THE protocol spends half of its execution time in the memory fence [FLR98].

Several approaches alleviate this problem. The split task queue by Dinan et al. [Din+09], designed for clusters of multiprocessor computers, allows lock-free local access to a private portion of the queue and can transfer work between the public and private portions of the queue without copying tasks. Thieves synchronize using a lock and the local process only needs to take the lock when transferring work from the public portion to the private portion of the queue. Michael et al. propose relaxed semantics for work-stealing: inserted tasks are executed *at least* once instead of *exactly* once, to avoid requiring memory fences and atomic instructions [MVS09]. In the work scheduler Wool [Fax08], originally only the first N tasks in the deque can be stolen, where N is determined by a parameter at startup. The `pop` operation only requires a memory fence when trying to pop a stealable task. In a later version, the number of stealable tasks is dynamically updated [Fax10].

In some work-stealing algorithms, shared deques are replaced by private deques, and work is explicitly communicated using a message-passing approach. Recently, Acar et al. proposed two algorithms for work-stealing using private deques [ACR13]. See further [ACR13] for an overview of other work with private deques.

Tasks are often stored as pointers that are removed from the deque when the task is stolen [FLR98; ABP01; Hen+06; CL05]. To virtually eliminate the overhead of task creation for tasks that are never stolen, Faxén proposed a direct task stack, storing tasks instead of pointers in the work queue, implemented in Wool [Fax08; Fax10]. Rather than synchronizing with thieves on the metadata of the queue (e.g. variables `top` and `bot` in the ABP algorithm), Wool synchronizes on the individual task descriptors, using locks when synchronizing with potential thieves, similar to the THE protocol. Sundell and Tsigas presented a lock-free version of Wool [ST09; Fax10], which still synchronizes on the individual task descriptors.

The work presented in this chapter combines some of the ideas from earlier work with a unique non-blocking synchronization mechanism between

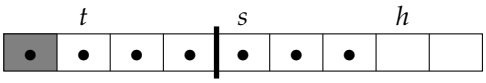


Figure 3.2 The split deque, with tail t, split point s and head h. A task at position x is **stolen** if $x < t$. It is **shared** if $x < s$, and **private** otherwise. Of the 7 tasks here, 4 are **shared** and 1 is **stolen**. It is sometimes possible that tasks are both **private** and **stolen**, i.e. $s < t$.

the worker and the thieves. Our non-blocking queue is a split task queue, like [Din+09] and [Fax08; Fax10], and features the direct task stack proposed in [Fax08]. Contrary to the design in [Fax08; Fax10], we synchronize on the metadata of the queue instead of on the individual task descriptors.

3.3 Design of the shared split deque

Acar et al. write that concurrent deques suffer from two limitations: 1) local deque operations (mainly pop) require expensive memory fences in modern weak-memory architectures; 2) they can be very difficult to extend to support various optimizations, especially steal-multiple extensions [ACR13]. They lift both limitations using private deques. Wool reduces the first limitation for concurrent deques by using a dynamic number of stealable tasks, but is difficult to extend for steal-multiple strategies, since tasks must be stolen individually.

This section presents a work-stealing deque that eliminates these limitations, by combining a non-blocking variant of the split task queue [Din+09] with direct task stealing from Wool [Fax08; Fax10]. A split point splits the deque into a shared portion and a private portion. This split point is modified in a non-blocking manner. See also Figure 3.2.

Similar to the direct task stack in Wool, the deque contains fixed-size task descriptors, rather than pointers to task descriptors stored elsewhere. Stolen tasks remain in the deque. The result of a stolen task is written to the task descriptor. This reduces the task-creation overhead of making work available for stealing, which is important since most tasks are never stolen. In pointer-based designs, there is the extra overhead of writing the pointer to enable work-stealing. This also means that with a pointer-based design, there is at least one additional accessed cacheline (the cacheline which stores the pointer), which is avoided with direct task stacks.

A major aspect of our design is which variables can be accessed by whom, and the resulting communication. To minimize unnecessary interactions between the thieves and the workers, we store all metadata of the queue in the following three cachelines. Remember that a cacheline is the "unit of transfer"

in a computer, and that most communication occurs when a worker writes to a memory location that is also in the cache of other workers.

Cacheline	Contents	Thief access	Owner access
Shared 1	`tail`, `split`, flag `allstolen`	Often	Sometimes
Shared 2	flag `movesplit`	Sometimes	Often
Private	`head`, `osplit`, flag `oallstolen`	–	Often

The deque is described using variables `tail`, `split` and `head`, which are indices in the task array (as in Figure 3.2), and the flags `allstolen` and `movesplit`.

- The `tail` variable is increased by thieves when stealing a task.
- The `tail` variable is decreased by the owner after removing stolen tasks, but this is delayed until the next time a fresh unstolen task is added (see the discussion below).
- The `split` variable is read by the thieves when stealing a task, but is only modified by the owner. The split point is changed either to increase the shared portion when all shared tasks are stolen, or to decrease the shared portion during `peek`, to ensure that the top-most task is in the private portion so it cannot be stolen.
- The `head` variable is only accessed by the owner; it is increased when tasks are added and decreased when tasks are removed.
- The flag `allstolen` is only modified by the owner. It is set when the `peek` operation detects that all tasks are stolen, and it is reset when the `push` operation adds a new task. The flag is used by thieves to avoid accessing the second shared cacheline.
- The flag `movesplit` is set by thieves to indicate that all shared tasks are stolen, so the split point should be moved. The flag is often checked by the owner and it is reset by the owner after moving the split point.

One of the design goals is to minimize communication between the owner and the thieves. Several choices reflect this goal:

- Thieves are not allowed to change the split point directly. Allowing this would force the owner to use expensive memory fences or atomic operations in executions of `peek` and `pop`. The disadvantage is that sometimes there may be tasks available that cannot be stolen until the owner moves the split point.
- The owner needs to check `movesplit` often, therefore we store it on a separate cacheline, to avoid "false sharing" with other variables.
- The owner normally does not need to know the value of `tail` and has its own copies of `split` (`osplit`) and `allstolen` (`oallstolen`).

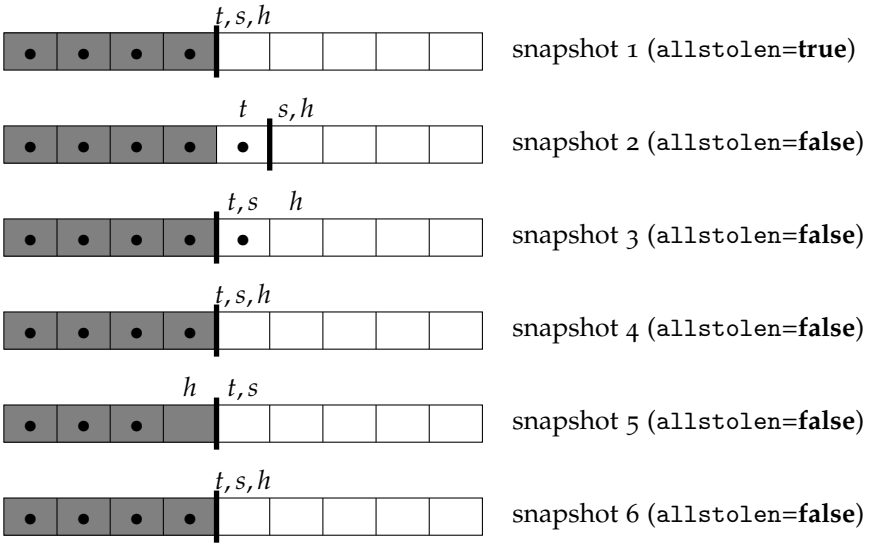


Figure 3.3 All tasks are stolen, so the owner sets the flag `allstolen` (snapshot 1). Task 4 is not yet done, so the owner steals a task from the thief. This task spawns a new task, and since `allstolen` is set, the `push` method sets t and s to appropriate values and resets the flag `allstolen` (snapshot 2). The new task is not stolen when `peek` is called and the split point is moved to prevent stealing after `peek` (snapshot 3). The task is then popped from the stack (snapshot 4) and executed. Now task 4 is done and is also popped from the stack (snapshot 5). Now another task is spawned, and since `allstolen` is no longer set, t and s are not appropriately updated (snapshot 6). It seems as if the new task 4 has been stolen, which is not the case.

- During normal operation (outside of moving the split point), thieves write to the first shared cacheline without affecting the owner and the owner only writes to the private cacheline.
- After stolen tasks have been removed from the deque, the owner must decrease `tail` so new tasks start as unstolen. This update to `tail` is delayed and performed by `push`, so removing multiple stolen tasks does not trigger excessive communication. We use the `oallstolen` in `push` to check if `tail` and `split` should be updated.

There is a special case where `tail` is not properly updated by `push`. When performing leapfrogging (stealing from the thief when the stolen task is not yet done), creating a new subtask causes `push` to update `tail` and `split` as expected and unset `allstolen`. However, when this subtask is not stolen and the originally stolen task is done, the `allstolen` flag should be set again to

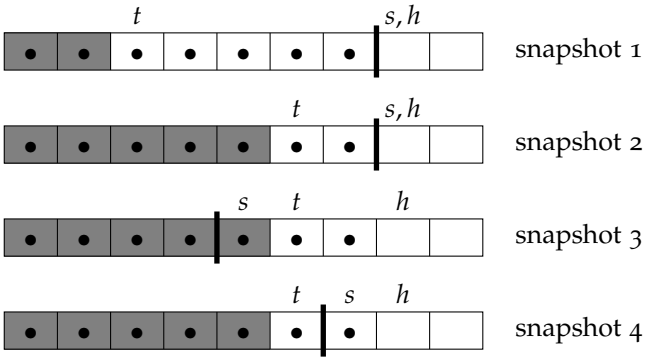


Figure 3.4 The owner reads variables t and s (snapshot 1), then thieves steal three tasks (snapshot 2) and the owner writes the new split point s (snapshot 3). After the memory fence, the owner reads the new value of t and updates the split point to its final position (snapshot 4). Without this memory fence, the owner would not know that more tasks had been stolen after the first new split point.

avoid the scenario in Figure 3.3. The easiest solution is to use a modified version of `pop` called `pop-stolen` when the popped task is stolen, which ensures that `allstolen` is set. An alternative solution without this special `pop-stolen` operation would check if `head` $<$ `split`, and if so, ensure that `allstolen` is set. This works, since always if `allstolen` is reset during leapfrogging, `head` $<$ `split` after popping the task that was stolen, as in snapshot 5.

In some cases, we need to use memory fences to ensure that the algorithms function correctly in weak memory models, and atomic operations such as `compare_and_swap` (`cas`) to avoid race conditions, as described in Section 1.3. The only variable that is written by multiple workers and that could cause problems is the `tail` variable on the first shared cacheline. Race conditions on the `movesplit` variable are not a problem for the correctness of the algorithm. We use one atomic `cas` operation and one memory fence to handle race conditions and reorderings from the weak memory model:

- Thieves use `cas` on `tail` and `split` simultaneously (they are adjacent in memory) to increase `tail` and detect modifications to `split` by the owner. The `cas` operation fails if the owner moved the split point, but the `cas` operation never modifies the `split` variable. The `cas` operation also ensures that only one thief succeeds in stealing a task. Thieves that simultaneously try to steal the same task fail their atomic `cas` operation.
- The owner does not use `cas` to move the split point, but instead uses a normal write followed by a memory fence. If the owner uses atomic `cas`, then it will have to perform this `cas` in a loop until it it successful,

```
1 def steal():
2     if allstolen : return None
3     t, s ← (tail, split)
4     if t < s :
5         if cas ((tail,split), (t,s), (t+1,s)) : return Task(t)
6         else: return None
7     elif ¬ movesplit : movesplit ← true
8     return None
```

Algorithm 3.3 The `steal` algorithm.

racing against the thieves. By using a normal write on the `split` variable, followed by a memory fence and a reread of the `tail` variable, it is guaranteed that no thief can write beyond the new split point after the change is globally visible, which the memory fence enforces.

See Figure 3.4 for an example with this memory fence. Note that moving the split point forward (increasing the shared portion) does not result in a problematic race condition.

3.4 Deque algorithms

We first explain the `steal` algorithm in detail. See Algorithm 3.3. It first checks `allstolen`, which also loads the values of `tail` and `split` as they are on the same cacheline. The method aborts if `allstolen` is set. We compare `tail` and `split` to see if there are still unstolen shared tasks. If this is the case, then we use atomic `cas` to increment the `tail` variable by 1 while keeping `split` unmodified. If this is succesful, then we have stolen task t and return this status. The atomic `cas` is unsuccessful if either some other thief has modified `tail`, or if the owner has modified `split`. If there are no unstolen shared tasks, then we check if `movesplit` is already set, and if not, then we set `movesplit`. This way, we avoid unnecessary communication if `movesplit` is already set.

Method `push` (Algorithm 3.4) adds a new task to the deque and increases `head` (lines 10–12). If `oallstolen` is set (line 13), then this is the first new task after removing one or more stolen tasks. We set `tail` and `split` such that the new task is shared and that it is the next task to be stolen (lines 14–15), and we reset `allstolen` and `movesplit` (lines 16–18). Otherwise, we check if `movesplit` is set and if it is, then we move the split point to halfway between the current split point and `head`, rounding up (lines 19–23). This is a safe method to increase the shared portion, as `split` $<$ `head`, unless all tasks have been stolen in which case lines 19–23 are not reachable. Memory fences and

```
9  def push(task):
10     if head = size : raise QueueFull
11     write task data at head
12     head ← head + 1
13     if oallstolen :
14         (tail,split) ← (head-1,head)
15         osplit ← head
16         allstolen ← false
17         oallstolen ← false
18         if movesplit : movesplit ← false
19     elif movesplit :
           // Grow shared portion
20         new_split ← (osplit + head + 1) / 2
21         split ← new_split
22         osplit ← new_split
23         movesplit ← false
```

Algorithm 3.4 The `push` algorithm.

```
24 def pop():
25     head ← head - 1
26 def pop-stolen():
27     head ← head - 1
28     if ¬ oallstolen :
29         allstolen ← true
30         oallstolen ← true
```

Algorithm 3.5 The `pop` and the `pop-stolen` algorithms.

```
24 def pop():
25     head ← head - 1
26     if head < osplit and ¬ oallstolen :
27         allstolen ← true
28         oallstolen ← true
```

Algorithm 3.6 An alternative to `pop` that avoids a separate `pop-stolen` method.

atomic operations are not necessary when increasing the shared portion, as there are no problematic race conditions.

Method `pop` (Algorithm 3.5) simply decreases the value of `head`. As per the discussion in Section 3.3 above, we use a separate method `pop-stolen` that sets `allstolen` in case `allstolen` was reset during leapfrogging. See Algorithm 3.6 for an alternative for `pop` that fixes the flag `allstolen` without requiring a

```
31 def peek():
32     if head=0 : raise QueueEmpty
33     if oallstolen : return Stolen(head-1)
34     if osplit = head :
35         if ¬ shrink-shared() :
36             allstolen ← true
37             oallstolen ← true
38             return Stolen(head-1)
39     if movesplit :
           // Grow public section (excluding head-1)
40         new_split ← (osplit + head) / 2
41         split ← new_split
42         osplit ← new_split
43         movesplit ← false
44     return Work(head-1)
```

Algorithm 3.7 The `peek` algorithm.

separate method `pop-stolen`. This requires one additional comparison between variables that are both on the private cacheline (`head` < `osplit`), however this would be done at every invocation of `pop` and not only those after leapfrogging.

Method `peek` (Algorithm 3.7) performs two main functions: check if the task on top of the stack is stolen, and if not, to ensure that it cannot be stolen. This is done by ensuring that the top task is in the private portion of the deque. This functionality is mostly delegated to a helper algorithm `shrink-shared` which is discussed below. The `shrink-shared` algorithm is called when the private portion is empty and returns **true** if the top task is now in the private portion and unstolen, and **false** if it is stolen. We first check if maybe all tasks are already stolen (line 33), in which case we can simply return this status. Then we check if the top task is in the shared portion or in the private portion, by comparing `osplit` and `head` (line 34). If the task is in the shared portion, then we try to move the split point with `shrink-shared` (line 35) and if this is not succesful, then we set the flag `allstolen` and return the status (lines 36–38). If either the top task was already in the private portion or if `shrink-shared` was successful, we continue at line 39 and check if there is a request to move the split point to increase the shared portion of the deque. This is implemented similar to increasing the shared portion in Algorithm 3.4, but we round down so that the top task stays in the private portion of the deque (lines 40–43). Finally, we return the status that the top task is not stolen (line 44).

The `shrink-shared` method (Algorithm 3.8) is called when the private portion is empty and returns **true** if the top task is now in the private portion

```
45 def shrink-shared():
46     t, s ← (tail, split)
47     if t = s : return false
48     new_s ← (t + s) / 2
49     split ← new_s
50     osplit ← new_s
51     memory fence
52     t ← tail
53     if t = s : return false
54     if t > new_s :
55         new_s ← (t + s) / 2
56         split ← new_s
57         osplit ← new_s
58     return true
```

Algorithm 3.8 The `shrink-shared` algorithm.

and unstolen, and **false** if it is stolen. First the algorithm loads the current values of `tail` and `split`. Note that the value of `split` equals `head` (see line 34), so if `tail` and `split` are equal at line 47, this means `tail` = `head` therefore all tasks are stolen and the algorithm immediately returns `false`. If this is not the case, then we set the new split point between `tail` and `split`, rounding down so that `new_s` < `split` (lines 48–50), which ensures that the task at the top of the deque will be in the private portion. As per the discussion in Section 3.3 above, we use a memory fence and then reload the value of `tail` (lines 51–52). We again check if `tail` = `head`, with s the original split value when loaded at line 46. If these are equal, then all tasks are stolen and the algorithm immediately returns `false` (line 53). If not, then we check if tasks have been stolen beyond the new split point. If this is the case, we move the split point again (lines 54–57), rounding down so the top task stays in the private portion. Since no thieves can steal tasks after the memory write of line 49 is globally visible (which is the case after the memory fence), we can be certain that no tasks have been stolen beyond the new split point this time, and we return **true** as the top task is not stolen and in the private portion.

3.5 Correctness

In this section, we aim to provide the reader with an intuitive proof of correctness. A proper full correctness proof should be performed using a proof checker.

We assume that the program correctly uses the keywords `SPAWN` and `SYNC`

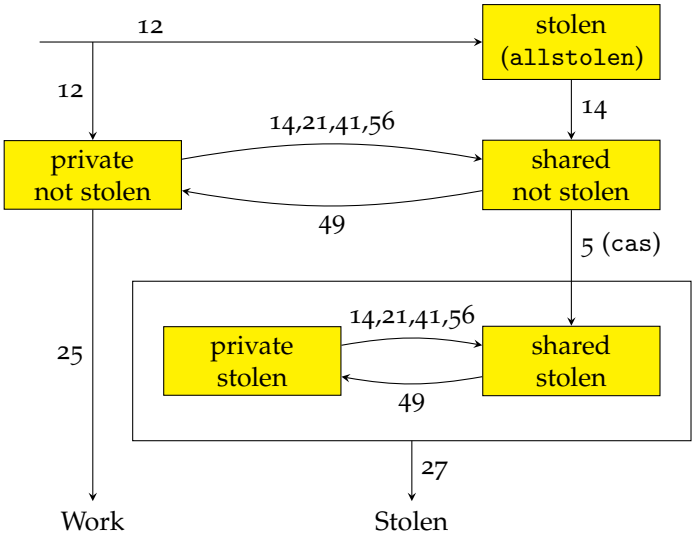


Figure 3.5 States and transitions of a task (only for tasks that exist). The labels of the transitions are line numbers in the algorithms.

to parallelize the algorithm, i.e., every `SPAWN` is matched by a `SYNC`, and that the spawned tasks can be performed independently, i.e., their execution order does not influence the result. The work-stealing algorithm is correct if each created task is executed once, and if the result of the parallel algorithm is the same as the original sequential algorithm, i.e., the `sync` call returns the same value as the return value of the matched `spawn` task.

We start with the following definitions and invariants:

1. A task x **exists** iff $x <$ `head`.
2. A task x is **stolen** iff $x <$ `tail`, and **not stolen** otherwise.
3. A task x is **shared** iff $x <$ `split`, and **private** otherwise.
4. If `allstolen` is not set, then `tail` $\leq$ `split` $\leq$ `head`.
5. If `allstolen` is set, then `tail` $\geq$ `head` and `tail` $\geq$ `split`.

We now establish that Figure 3.5 accurately describes the life-cycle of each task from creation until destruction. Considering all lines of code in Algorithms 3.3, 3.4, 3.5, 3.7 and 3.8, we determine all lines that modify the status of a task by manipulating `tail`, `split` or `head`.

- Line 5 increases `tail` by 1 if the atomic `cas` is successful, leaving `split` unmodified. This marks 1 task as **stolen**.
- Line 12 increases `head` by 1 and thus creates 1 new task. Note that if `allstolen` is set, then `tail` $\geq$ `head`, so the new state then starts as **stolen**.

- Line 14 decreases `tail` to `head` − 1. This only affects 1 existing task.
- Line 14 sets `split` to `head`. Note that the old value of `split` could be lower or higher than `head`; `allstolen` only implies that `tail` ≥ `head` and `tail` ≥ `split`. This can move some tasks from **private** to **shared**.
- Line 21 increases `split` (in `push`), moving tasks from **private** to **shared** when `movesplit` is set
- Line 25 decreases `head` by 1 and thus destroys 1 task.
- Line 27 decreases `head` by 1 and thus destroys 1 task.
- Line 41 increases `split` (in `peek`, after ensuring that the last task is **private**), moving tasks from **private** to **shared** when `movesplit` is set.
- Line 49 decreases `split`, moving tasks from **shared** to **private**.
- Line 56 increases `split` (to recover after the memory fence), moving tasks from **private** to **shared**.

Note that `peek` and `shrink-shared` not only move tasks from **shared** to **private**, but also from **private** to **shared**. The transitions between states in Figure 3.5 are labeled with the lines in the above list. In particular, note the following regarding the task created by `push`:

- If `allstolen` is set, then the newly created task might start as **stolen**, which is repaired at line 14.
- It is also possible that `push` is called during leapfrogging, in which case it is possible that `allstolen` is set, but the new task starts as **private** and as **not stolen** and line 14 moves it to the **shared** state.
- Line 14 might affect other existing states, moving them from **private** to **shared**, if `split` < `head`.
- If `allstolen` is not set, then by invariant, the new task starts as **private** at line 12.

By inspection of Algorithm 3.8 and the explanation in Section 3.4, we state that if `shrink-shared` returns **false**, then the top task is **stolen**, and if it returns **true**, then the top task is **private** and **not stolen**. Without the memory fence at line 51, it would be be possible that a task is **stolen** but `shrink-shared` returns `true`. Furthermore, by inspection of Algorithm 3.7 we see that `peek` correctly returns Stolen if a task is **stolen**, and Work if it is in the private portion of the deque and **not stolen**.

Based on the above discussion, we state the following:

1. Only `push` creates tasks and `push` creates one task.
2. Only `pop` and `pop-stolen` destroy tasks and they destroy one task.
3. A task can only be stolen once, because we use atomic `cas`.

```
31 def sync-fast():
32     if ¬ movesplit and ¬ oallstolen and osplit < head :
33         head ← head − 1
34         return task[head].execute()
35     else: return sync()
```

Algorithm 3.9 The `sync-fast` algorithm.

4. If and only if a task is stolen, `peek` returns that it is stolen.
5. After `peek` reports that a task is not stolen, it cannot be stolen until the next `peek` or `push`.

Then, by inspecting Algorithm 3.2 and the above properties, we conclude:

1. `spawn` creates one task and only `spawn` creates tasks.
2. `sync` destroys one task and only `sync` destroys tasks.
3. `sync` returns the correct result.
   - When a task is not stolen, `sync` returns the result of `execute`.
   - When a task is stolen, the thief writes the result of `execute` to `result`, which `sync` returns after synchronizing on the flag `done`.
4. A task is executed exactly once. It is either stolen once and executed by the thief, or it is executed by `sync`.
5. A thief never operates on a task that does not exist.

In particular, this implies that the desired properties (correct result, each task executed once) are true.

3.6 Implementation of the framework Lace

This section details our implementation of Lace, a C library that provides a work-stealing framework in a style similar to Wool and Cilk.

Lace creates one POSIX thread ("pthread") for each available core. The default configuration allocates two blocks of memory: the program stack for the new thread, and the memory for the work-stealing deque, including the cachelines for the metadata. Laces uses the `hwloc` library (available on various operating systems, including Linux, OSX and Windows) to pin each pthread to their processor core and to place the program stack and the work-stealing deque on the memory closest to their core.

3.6.1 Standard work-stealing functionality

To implement tasks, Lace provides C macros that require only few modifications of the original source code. See the table below.

Task declaration	`TASK_DECL_1(int, f, int, n);`
Task implementation	`TASK_IMPL_1(int, f, int, n) { ... }`
Task decl+impl	`TASK_1(int, f, int, n) { ... }`
Spawn a task	`SPAWN(f, 4);`
Synchronize a task	`int res = SYNC(f);`
Call a task directly	`int res = CALL(f, 4);`
Drop a task (if not stolen)	`DROP();`

The `TASK_...._n` macros are used for tasks with n input parameters. The first parameter of the macro is the return type, the second is the function name, and then alternating the type and name of the function parameters. The `DECL` and `IMPL` versions are used to separate the declaration in the header file from the implementation. The `SYNC` macro actually requires the identifier of the task, since each task gets a set of `_SPAWN` and `_SYNC` macros, but this is purely to allow for compiler optimization and for the correct return types. In addition, we use a "fast" version of `sync` (Algorithm 3.9) which helps for compiler optimization and significantly improves the performance due to how the compiler generates the program code. For programs that want the option to sometimes cancel the execution of a task rather than synchronizing, they can use `DROP` instead of `SYNC` which skips executing the task unless it has been stolen. The `DROP` macro is simply the `SYNC` macro, but without executing the task if it was not stolen, or returning the result if it was stolen. It is a convenience macro for programs that want to discard the result of a subtask that is not yet executed.

3.6.2 Interrupting tasks to run a new task tree

One helpful feature for garbage collection in Sylvan that we implemented in Lace is a feature that suspends all current tasks and starts a new task tree.

New task tree with 1 root task `t`	`NEWFRAME(t)`
All workers execute a task `t`	`TOGETHER(t)`

Both features work in a similar way. The difference is that the `NEWFRAME` macro starts 1 new task and all other workers help execute this task in parallel, while the `TOGETHER` macro lets all workers execute a local copy of the given task. Workers regularly check if they have to cooperate on a new "frame", e.g., manually by the user or during leapfrogging and work-stealing. This is done with a macro `YIELD_NEWFRAME` which checks a task pointer stored in a separate cacheline. This pointer normally is zero, but if set, it points to the task that all workers (except the initiating worker) execute. For `NEWFRAME` this

task is a helper task that performs work-stealing until the root task is done; for `TOGETHER` this is the task that all workers execute. Events play out as follows:

1. One worker initiates by setting the task pointer using atomic `cas`.
2. All other workers call the function `yield` after they see the task pointer.
3. `yield` creates a local copy of the task for each worker.
4. All workers wait in a barrier until they all finished step 3. This is the first barrier and it ensures that no workers are still busy stealing from other workers.
5. The initiating worker sets the task pointer to zero.
6. All workers create a backup copy of their `tail`, `split` and `allstolen` variables.
7. All workers wait in a barrier until they all finished step 6.
8. The initiating worker executes its root task; all other workers execute their local copy of the given task, which is work-stealing (for `NEWFRAME`) or the same task as the initiating worker (for `TOGETHER`).
9. All workers wait in a barrier until they all finished step 8.
10. All workers restore their copy of the `tail`, `split` and `allstolen` variables, and return from the procedure.

A barrier after step 10 is not necessary since after finishing their task, `tail` $\geq$ `split` for all workers, so workers that finish the last step early cannot accidentally steal invalid tasks from each other.

Sylvan uses the `NEWFRAME` macro as part of garbage collection, and the `TOGETHER` macro to perform thread-specific initialization. The parallelized decision diagram operations typically check whether garbage collection must be performed at the start of each operation using the `YIELD_NEWFRAME` macro.

3.7 Experimental evaluation

We evaluate Lace on several benchmarks and compare it to the work-stealing framework Wool [Fax10], which uses the classical leapfrogging strategy. This version of Wool (0.1.5alpha) has a dynamic split point and does not use locking. We compare the performance of Lace with Wool, for two reasons. Our implementation resembles the implementation of Wool, making a comparison easier. Also, [Fax10] and [PBF10] show that Wool is efficient compared to Cilk++, OpenMP and the Intel TBB framework, with a slight advantage for Wool. We also compare our algorithm to the receiver-initiated version (using the alternative `acquire` function) of the private deque of Acar et al. [ACR13], which we implemented as an alternative task pool in the Lace framework.

A core challenge in parallel processing is load balancing with minimal overhead. Available work must be distributed among the participating processors with maximal speedup. Overhead comes from two main sources. There is "sequential overhead" which is just the result of using parallelization techniques, i.e., measured by comparing the original sequential program with the parallelized program on one processor. There is "dynamic overhead" which is the overhead from adding processors, usually a result of extra communication. We evaluate the performance of the parallelized benchmark programs by looking at the sequential overhead and at the obtained speedup by the parallelization with N workers.

3.7.1 Benchmarks

For all benchmarks, we use the smallest possible granularity and do not use sequential cut-off points, since we are interested in measuring the overhead of the work-stealing algorithm. Using a larger granularity and sequential cut-off points may result in better scalability for some benchmarks, but we are not interested in obtaining the best scalability, rather we want the benchmarks to be challenging.

- *Fibonacci*. For a positive integer N, calculate the Fibonacci number by calculating the Fibonacci numbers $N-1$ and $N-2$ recursively and add the results. This benchmark generates a skewed task tree and is commonly used to benchmark work-stealing algorithms, since the actual work per task is minimal.
  Number of tasks: 20,365,011,073 (`fib 50`).
- *Queens*. For a positive integer N, calculate the number of solutions for placing N queens on a $N \times N$ chessboard so that no two queens attack each other. Each task at depth i spawns up to N new tasks, one for every correct board after placing a queen on row i.
  Number of tasks: 171,129,071 (`queens 15`).
- *Unbalanced Tree Search*. This benchmark by Olivier et al. [Oli+06] is designed to evaluate the performance for parallel applications requiring dynamic load balancing. The algorithm uses the SHA-1 algorithm to generate geometric and binomial trees. The generated binomial trees (T3L) have unpredictable subtree sizes and depths and are optimal adversaries for load balancing strategies [Oli+06]. The geometric tree (T2L) appears to be easy to load balance in practice.
  Number of tasks: 96,793,509 (`uts T2L`) and 111,345,630 (`uts T3L`).
- *Rectangular matrix multiplication*. Given N, compute the product of two (pseudo-)random rectangular $N \times N$ matrices A and B. We use the `matmul`

algorithm from the Cilk benchmark set.
Number of tasks: 3,595,117 (`matmul 4096`).

3.7.2 Results

Our test machine has four twelve-core AMD Opteron 6168 processors. The system has 128 GB of RAM and runs Scientific Linux 6.0 with kernel version 2.6.32. We considered using fewer than 48 cores to reduce the effects of operating system interference, but we did not see significant effects in practice. We compiled the benchmarks using gcc 4.7.2 with flag `-O3`.

See Table 3.2 for the results of the benchmark set. Each T_{48} data point is the average of 50 measurements. Each T_1 and T_S data point is the average of 20 measurements. We also included measurements marked with "LF+" and "TLF" that are discussed below in Section 3.7.3.

In general, Table 3.2 shows similar performance for all three algorithms. The three benchmarks `uts T2L`, `queens` and `matmul` are trivial to parallelize and have no extra overhead with 48 workers, i.e., $T_1/T_{48} \approx 48$.

Comparing T_S and T_1 for all benchmarks, we see that the overhead of work-stealing is small for all three work-stealing algorithms, with the exception of the `fib` benchmark. For benchmark `fib` with our algorithm, $T_1 < T_S$, which appears to be related to compiler optimizations. In general, we observed that variation in T_1 is often related to code generation by the compiler. In some cases, removing unused variables and other minor changes even increased T_1 by up to 20% during development. Inspection of the generated program code revealed that the compiler inlined execution of the `fib` "call". If we add this optimization manually in the sequential version of the Fibonacci algorithm, we see a significant performance improvement. This makes it difficult to compute or estimate the sequential overhead of parallelization, although the results with the other benchmarks suggest that this overhead is minor.

We measured the runtimes of `fib` and `uts T3L` using 4, 8, 16, 24, 32 and 40 workers to obtain the speedup graph in Figure 3.6. This graph suggests that the `fib` benchmark scales well and that similar results may be obtained using a higher number of processors in the future. The scalability of the `uts T3L` benchmark appears to be limited after 16 workers. We discuss this benchmark below.

We also measured the average number of steals during a parallel run with 48 workers. See Figure 3.7. We make a distinction between normal stealing when a worker is idle, and leapfrogging when a worker is stalled because of unfinished stolen work. We also measured the amount of split point changes. The number of 'grows' indicates how often thieves set `movesplit`. The number of 'shrinks' is equal to the number of memory fences. In the `uts T3L` benchmark, the high

	Benchmark time			Speedup	
Lace	T_S	T_1	T_{48}	T_S/T_{48}	T_1/T_{48}
fib 50	149.2	144	4.13	34.5	34.9
uts T2L	84.5	86.0	1.81	46.1	47.4
uts T3L	43.11	44.2	2.23	18.7	19.9
uts T3L (LF+)	43.11	44.26	1.154	37.4	38.3
queens 15	533	602	12.63	42.2	47.7
matmul 4096	773	781	16.46	47.0	47.5
Private deque					
fib 50	149.2	208	5.22	23.2	39.8
uts T2L	84.5	86.1	1.83	45.7	47.0
uts T3L	43.11	44.8	2.55	17.3	17.5
uts T3L (LF+)	43.11	44.83	1.240	34.8	36.2
queens 15	533	541	11.34	43.3	47.7
matmul 4096	773	774	16.34	47.3	47.4
Wool					
fib 50	149.2	185	4.38	34.1	42.2
uts T2L	84.5	85.1	2.00	42.5	42.5
uts T3L	43.11	44.3	2.12	19.4	20.9
uts T3L (TLF)	43.11	44.27	1.172	36.8	37.8
queens 15	533	539	11.23	47.5	48.0
matmul 4096	773	780	16.40	47.2	47.5

Table 3.2 Averages of running times (seconds) for all benchmarks. Speedups are calculated relative to both the time of the sequential version (T_S) and the parallel version with one worker (T_1). Each T_{48} data point is the average of 50 measurements. Each T_1 and T_S data point is the average of 20 measurements. The exception are the TLF and LF+ benchmarks (discussed in Section 3.7.3), which were performed over 200 times.

number of leaps and split point changes may indicate that the stolen subtrees were relatively small.

3.7.3 Extending leapfrogging

Benchmark `uts T3L` appears to be a good adversary for all three algorithms. This is partially related to the leapfrogging strategy, which forces workers that wait for the result of stolen tasks to steal from the thief. This strategy can result in chains of thieves waiting for work to trickle down the chain.

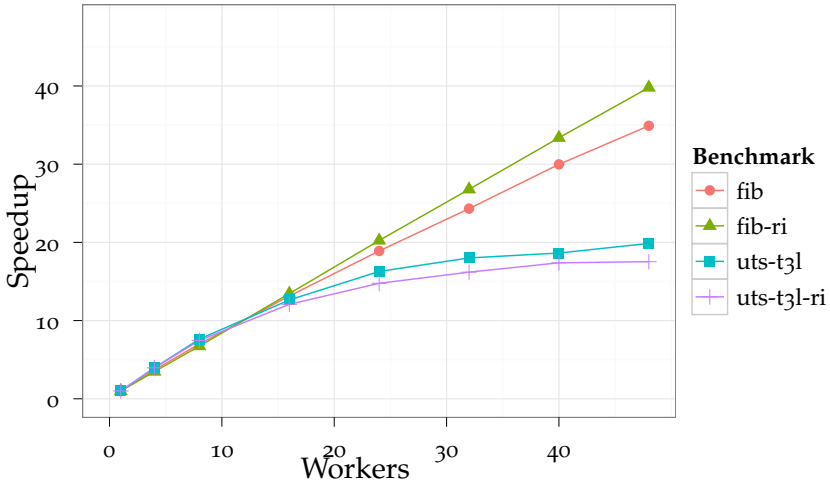


Figure 3.6 Absolute speedup graph (T_1/T_N) of the `fib` and `uts T3L` benchmarks using Lace with our algorithm and Lace with the private deque receiver initiated (`-ri`) algorithm. Each data point is based on the average of 20 measurements.

Benchmark	#steals	#leaps	#grows	#shrinks
fib 50	865	50,569	70,789	97,750
uts T2L	4,554	82,440	72,222	57,701
uts T3L	158,566	4,443,432	2,173,006	846,509
queens 15	1,964	6,053	5,694	6,618
matmul 4096	2,492	12,456	13,081	9,911

Figure 3.7 The average total number of steals, leaps, grows and shrinks over 7 runs with 48 workers.

For example, when worker 2 steals from worker 1, worker 1 will only steal from worker 2. If worker 3 steals from worker 2 and worker 4 steals from worker 3, new tasks will be generated by worker 4 and stolen by worker 3 first. Worker 3 then generates new work which can be stolen by workers 2 and 4. Worker 1 only acquires new work if the subtree stolen by worker 4 is large enough. The updated version of Wool [Fax10] implements an extension to leapfrogging, called transitive leapfrogging. This feature is documented in the distribution of Wool versions 0.1.5alpha and 0.1.7alpha, which are available at `http://www.sics.se/~kff/wool/`. Transitive leapfrogging enables workers

to steal from the thief of the thief, i.e., still steal subtasks of the original stolen task. There is no actual guarantee that the leapfrogging strategy steals subtasks of the stolen branch, since thieves may finish the stolen task and steal another branch of the subtree while the victim is about to steal from the thief. This is also the case with transitive leapfrogging.

Instead of implementing transitive leapfrogging like in Wool, we extended Lace to steal from a random worker whenever the thief has no available work to steal. We included the results of this extension ("LF+") in Table 3.2, compared to transitive leapfrogging ("TLF") in Wool. All benchmarks now have reasonable speedups, improving from a speedup of 20x to a speedup of 36x with 48 workers.

The disadvantage of our extension is that in practice transitive leapfrogging and classical leapfrogging require a smaller task stack than the random stealing extension. It is, however, very simple to implement, while resulting in similar performance. We measured the peak stack depth with the `uts T3L` benchmark for all 48 workers. We observed an increase from a peak stack depth of 6500–12500 tasks with classical leapfrogging to 17000–21000 tasks with the random stealing extension. Since every task descriptor for `uts T3L` is 64 bytes large (including padding), this strategy required at most 1 extra megabyte per worker for `uts T3L`. We also observed that the number of times the shared porting was increased ("grows") decreased by 50%.

3.8 Conclusion and Discussion

This chapter presented our work-stealing framework Lace, with features that we need for garbage collection in Sylvan. Lace has an intuitive interface similar to frameworks like Cilk and Wool, and has a small source code footprint. This chapter also presented the novel non-blocking split deque that Lace is built around. Our design has the advantage that it does not require memory fences for local deque operations, except when reclaiming tasks from the shared portion of the deque, and only needs the expensive atomic `cas` operation for stealing tasks.

Our experiments show that our work-stealing deque is competitive with Wool and with the private deque algorithm of Acar [ACR13]. We gain near optimal speedup for several benchmarks, with very limited overhead compared to the sequential program. The most challenging benchmark was the `uts T3L` benchmark, for which a speedup of 37.4x was obtained on 48 cores, compared to the sequential version. Extending leapfrogging with random stealing greatly improves the scalability for the `uts T3L` benchmark. Clearly, for this benchmark, the ideal speedup is not yet reached, although it is not clear whether improvements are possible in practice.

The other benchmark that did not result in the ideal speedup was the Fibonacci benchmark. This benchmark consists of nearly empty tasks, and suffered from variation in the runtime due to compiler optimizations. Although there is theoretical room for better performance, it depends on the application whether the bottleneck for speedup is the work-stealing framework or not.

Compared to the private deque algorithm, our split deque allows stealing of multiple tasks by different thieves in the shared deque without cooperation of the owner, while the private deque algorithm requires cooperation of the owner for every steal transaction.

Extensions There are several possible extensions to the work-stealing deque.

Resizing. Our work-stealing deque uses a fixed-size array. Given that virtual memory is several orders of magnitude larger than real memory and the ability of modern operating systems to allocate only used pages, we can avoid overflows by allocating an amount of virtual memory much higher than required. The deque could be extended for resizing, for example using linked lists of arrays, but we feel this is unnecessary in practice.

Steal-multiple strategies. One extension to work-stealing is the policy to steal more than one task at the same time, e.g., stealing half the tasks in the deque, which has been argued to be beneficial in the context of irregular graph applications [HS02; Din+09]. This is easily implemented by modifying line 5 to steal multiple tasks (increase `tail` by more than 1). However, in preliminary tests on a single NUMA machine, this did not sufficiently improve performance to be further investigated at the time.

Other memory models. The algorithm is designed for the standard x86 weak memory model, which only allows reordering loads before stores. Weaker memory models may for example allow reordering stores. Additional memory fences may be needed, for example to ensure in the `steal-and-run` algorithm that the result is set before the `done` flag is set.

Several open questions Several open questions remain. When growing the shared deque, the new split point is the average of `split` and `head`, and when shrinking the shared deque, the new split point is the average of `tail` and `head`. More optimal strategies may exist. A limitation of our approach is that tasks can only be stolen at the tail of the deque. This limits work-stealing strategies. Designs that allow stealing any task may be useful for some applications. Our benchmarks all consist of uniform small tasks. Benchmarking on larger or irregular sized tasks may be disadvantageous for the private deque algorithm, since it requires owner cooperation on every steal. Workers that spend a relatively long time performing a task, are unable to check if they need to communicate work to a thief. We mainly studied the leapfrogging strategy

and the extension that uses random stealing when leapfrogging does not work. It may be interesting to see what the performance characteristics and typical memory requirements are for purely random work-stealing, or perhaps for strategies that try to steal from workers that have most tasks. Finally, we performed our experiments on a single NUMA machine. On such machines, communication costs are low compared to distributed systems. It may be interesting to compare the work-stealing algorithms on a cluster of computers using a shared-memory abstraction. In fact, this work is being performed by Oortwijn et al. at the University of Twente. Especially steal-multiple strategies may be more beneficial when communication is more expensive, which is still open for further investigation.

Chapter 4

Concurrent nodes table and operation cache

As outlined in Chapter 2, to efficiently parallelize decision diagram operations, we need to (1) execute recursive subtasks in parallel, and (2) perform memory operations in a scalable manner, i.e., using optimized scalable data structures. This chapter is concerned with scalable data structures. Section 4.1 introduces the topic. We present three versions (one light-weight blocking and two wait-free variants) of the hash table that store the decision diagram nodes in Section 4.2. Section 4.3 then discusses the wait-free operation cache. Finally, Section 4.4 concludes this chapter.

4.1 Scalable data structures

The parallel efficiency or parallel speedup of a task-based parallelized algorithm depends largely on the contents of each task. Apart from the number of independent subtasks, which influences the amount of parallelism for work-stealing, an important metric for parallel speedup is the "arithmetic intensity" (or its dual "memory intensity"). Arithmetic intensity expresses the ratio of processor computation versus memory access, usually in clock cycles per transferred byte or transferred cacheline. Typically, processor computation is not a shared resource, while memory bandwidth is a scarce shared resource. Algorithms with a low arithmetic intensity (or high memory intensity) often have lower parallel speedup as the number of threads increases, as the time required to perform memory operations increases. This is especially so if the processors operate on the same cachelines in the memory, as then the number of messages increases with the number of threads. Therefore, for optimal parallel speedups, the memory intensity of the operations should be as low as possible. See also [WWP09] for a more thorough discussion of this subject.

Decision diagram operations are typically memory intensive, since they consist mainly of calls to the operation cache and the unique table and perform few calculations. Hence the design of scalable concurrent data structures for the cache and the unique table is crucial for the parallel performance of decision diagram implementations.

In parallel programs, memory accesses can result in race conditions or data corruption, for example when multiple threads write to the same location in memory. Typically data structures are protected against race conditions using locking techniques. While locks are relatively easy to implement and reason about, they often severely cripple parallel performance, especially as the number of threads increases. Threads have to wait until the lock is released, and locks can be a bottleneck when many threads try to acquire the same lock. Also, locks can sometimes cause spurious delays that smarter data structures could avoid, for example by recognizing that some operations do not interfere even though they access the same resource.

A standard technique that avoids locks uses the atomic `compare-and-swap` (`cas`) operation, which is supported by many modern processors.

```
1 def compare-and-swap(location, expected, newvalue):
2     value ← *location
3     if value ≠ expected : return False
4     *location ← newvalue
5     return True
```

This operation atomically compares the contents of a given location in shared memory to some given expected value and, if the contents match, changes the contents to a given new value. If multiple processors try to change the same bytes in memory using `cas` at the same time, then only one succeeds.

Datastructures that avoid locks are called non-blocking or lock-free. Such data structures often use the atomic `cas` operation to make progress in an algorithm, rather than protecting a part that makes progress. For example, when modifying a shared variable, an approach using locks would first acquire the lock, then modify the variable, and finally release the lock. A lock-free approach would use atomic `cas` to modify the variable directly. This requires only one memory write rather than three, but lock-free approaches are typically more complicated to reason about, and prone to bugs that are more difficult to reproduce and debug.

There is a distinction between different levels of lock freedom. In this thesis we are concerned with three levels:

- In **blocking** data structures, it may be possible that no threads make progress if a thread is suspended. If an operation may be delayed forever because another thread is suspended, then that operation is blocking.

Furthermore, we call data structures where mutexes are avoided (especially heavy-weight mutexes provided by the operating system, which can cause process switches) but that have light-weight fine-grained `cas`-locks **light-weight blocking**.

- In **lock-free** data structures, if any thread working on the data structure is suspended, then other threads must still be able to perform their operations. There is always at least one thread that makes progress. In other words, an operation may be delayed forever, but if this is because another thread is making progress and never because another thread is suspended, then that operation is lock-free.
- In **wait-free** data structures, regardless of the behavior of the other threads, every thread can complete its operation within a bounded number of steps; all threads make progress.

The atomic `cas` operation is often used in loops. Typically this loop consists of reading the current value of the variable, and using `cas` to change the variable from this current value to a new value, until the operation succeeds or is aborted. This can lead to different levels of lock freedom:

- If a thread may fail the atomic `cas` operation when other threads are suspended (often when using something else than the current value of the variable for `cas`, for example when implementing a spinlock), then the thread stays in the loop forever and the operation is blocking.
  For example, the loop "loop until $\texttt{cas}(\text{location}, 0, 1)$" only terminates when another thread (which may be suspended) sets `location` to 0.
- If a thread performs an atomic `cas` operation in a loop, and there are no spurious `cas` failures, i.e., every time the `cas` operation fails, it is because another thread made progress, then the operation is lock-free. It is however possible that a thread stays in the loop forever, while other threads make progress.
- If a thread performs an atomic `cas` operation in a loop, but the number of `cas` failures is bounded, then the thread can complete its operation within a bounded number of steps and the operation is wait-free.

This chapter presents several light-weight blocking and wait-free hash tables.

4.2 Unique table

This subsection describes the hash tables that we use to store the unique decision diagram nodes. The hash tables store fixed-size decision diagram nodes (16 bytes for each node) and strictly separates lookup and insertion of nodes from a stop-the-world garbage collection phase, during which the

table may be resized. From the perspective of the nodes table algorithms (and correctness), all threads of the program are in one of two phases:

1. During **normal operation**, threads only call the `find-or-insert` operation, which takes as input the 16-byte data and either returns a unique identifier for the data, or raises the TableFull signal if the algorithm fails to insert the data.
2. During **garbage collection**, the `find-or-insert` operation is never called. Instead, methods `clear`, `mark` and `rehash` (described in Section 2.2.5) are called to perform garbage collection.

This simplifies the requirements for the hash tables. The `find-or-insert` operation must have the following property: if the operation returns a value for some given data, then other `find-or-insert` operations may not return the same value for a different input, or return a different value for the same input. This property must hold between garbage collections; garbage collection obviously breaks the property for nodes that are not kept during garbage collection, as nodes are removed from the table to make room for new data.

The rest of this section is organized as follows. First, Section 4.2.1 describes the original hash table from [LPW10] that we based our designs on. Section 4.2.2 then presents variant 1 of our nodes table, an early design that incorporates a reference counter for each bucket. We used this design in the preliminary implementation of Sylvan and include it here for completeness. We continue in Section 4.2.3 with variant 2, which is the first version of the hash table that uses the current mark-and-sweep approach for garbage collection and also supports resizing the table. We finish in Section 4.2.4 with variant 3, which is the final version of the hash table. This hash table uses bit arrays to manage data allocation, reducing the number of expensive atomic `cas` operations per `find-or-insert` invocation. The designs presented in Section 4.2.3 and Section 4.2.4 are also wait-free rather than light-weight blocking. In Section 4.2.5 we compare the three variants by analyzing the number of memory writes and atomic `cas` operations that are necessary in normal operation. The performance of variants 2 and 3 is compared using the LTSmin toolset in Section 5.5.5 in the next chapter.

Related work Various shared-memory concurrent hash tables have been published in the literature. For an overview of such hash tables, see for example [SB14; MSD16]. The hash tables that we present here are based on the shared hash table by Laarman et al. [LPW10]. See the work by Gao, Groote and Hesselink [GGH05b; GGH05a; GGH07] for a line of research on lock-free hash tables with garbage collection, which is an important aspect also for decision diagrams operations. In particular [GGH07] studies mark-and-sweep garbage collection and uses computer-assisted theorem proving to prove the correctness

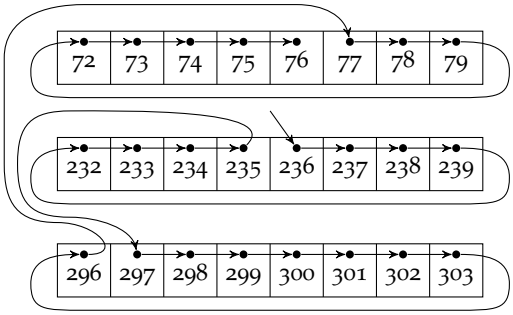


Figure 4.1 Example of the walking-the-line probe sequence, with the starting buckets 236, 297 and 77 based on the first three hash values of the data.

of their hash table algorithms, although their algorithms are designed to allow garbage collection concurrently with inserting data into the hash table, which we do not support.

4.2.1 Original hash table

Our hash tables are based on the hash table in [LPW10] that is designed to store visited states in model checking. This hash table only supports the `find-or-insert` operation and does not need to support resizing operations, as almost all the available memory is used for the hash table anyway, which simplifies its design. The hash table incorporates several ideas:

- Using a probe sequence called "walking-the-line" that is efficient with respect to transferred cachelines.
- Separating the stored data in a "data array" and the hash of the data in the "hash array" so directly comparing the data is often avoided.
- Using a light-weight parametrised local "writing lock" when inserting data, which almost always only delays threads that insert the same data.

Probe sequence Every hash table needs to implement a strategy to deal with hash table collisions, i.e., when different data hashes to the same location in the table. To find a location for the data in the hash table, some hash tables use open addressing: they visit buckets in the hash table in a deterministic order called the probe sequence, to either detect that the data is already in the hash table, or to find an empty bucket which indicates that the data can be inserted into that bucket. One of the simplest probe sequences is linear probing, where the data is hashed once to obtain the first bucket (e.g. bucket 61), and the probe sequence consists of all buckets from that first bucket (e.g. 61, 62, 63...).

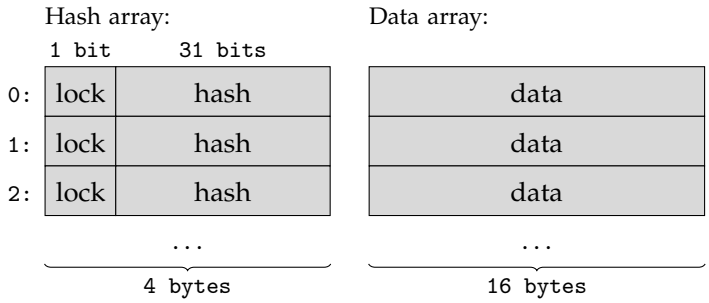


Figure 4.2 Layout of the lock-less hash table using a separate hash array and data array.

An alternative to linear probing is walking-the-line, proposed in [LPW10]. Since data in a computer is transferred in blocks called cachelines, it is more efficient to use the entire cacheline instead of only a part of the cacheline. For example, if there are 8 buckets per cacheline and we assume that the buckets are properly aligned so that the first cacheline starts with bucket 0, then linear probing starting at bucket 61 would only check buckets 61–63 of the first accessed cacheline. In walking-the-line, the other buckets in that cacheline are also checked, so after buckets 61–63, also buckets 56–60 would be checked. Then, a new hash value is obtained for the data using a hash function to obtain the next starting bucket. In theory, this procedure could be repeated forever; in practice, after a certain number of cachelines the procedure terminates with the result that the table is full. See also Figure 4.1 for an example of walking-the-line.

Writing lock When multiple workers simultaneously access the hash table to find or insert data, there must be some mechanism to avoid race conditions, such as inserting the same data twice, or trying to insert different data at the same location simultaneously. Rather than using a global lock on the entire hash table or regions of the hash table, or a non-specific local lock on each bucket, the hash table of [LPW10] combines a short-lived local lock with a hash value of the data that is inserted. This way, threads that are finding or inserting data with a different hash value know that they can skip the locked bucket in their search.

An empty bucket is first locked using an atomic `cas` operation that sets the lock with the hash value of the inserted data, then writes the data, and then releases the lock. Only workers that are finding or inserting data with the same hash as the locked bucket need to wait until the lock is released. This approach

```
1  def find-or-insert(data):
2      h ← hash(data)
3      for s ∈ probe-sequence(data) :
4          V ← harray[s]
5          if V = empty :
6              if cas(harray[s], empty, locked(h)) :
7                  darray[s] ← data
8                  harray[s] ← filled(h)
9                  return s
10             else: V ← harray[s]
11         if V.hash = h :
12             while V = locked(h) : V ← harray[s]
13             if darray[s] = data : return s
14     raise TableFull
```

Algorithm 4.1 Algorithm for parallel `find-or-insert` of the original lock-less hash table [LPW10].

is not lock-free. The authors state that a mechanism could be implemented that ensures local progress (making the algorithm wait-free), but that this is not needed, since the writing locks are rarely hit under normal operation [LPW10].

Separated arrays The hash table stores the hash of the data and the short-lived lock separated from the stored data. The idea is that the `find-or-insert` algorithm does not need to access the stored data if the stored hash does not match with the hash of the data given to `find-or-insert`. This reduces the number of accessed cachelines during `find-or-insert`. See also Figure 4.2. Each bucket *i* in the hash array matches with the bucket *i* in the data array. The hash that is stored in the hash array is independent of the hash value used to determine the starting bucket in the probe sequence, although in practice hash functions give a 64-bit or 128-bit hash that we can use both to determine the starting bucket in the probe sequence and the 31-bit hash for the hash array.

The `find-or-insert` algorithm The `find-or-insert` algorithm is given in Algorithm 4.1. Each bucket can be in three states: either `empty` (the fields "lock" and "hash" of the bucket are zero), or `locked` for a certain hash, indicating that data is being inserted in that bucket, or `filled` for a certain hash. A hash function is used that never hashes to zero.

The algorithm itself is straightforward. For every bucket in the probe sequence, we read the bucket (line 4) and check if it is empty (line 5). In this case, we know that the data is not yet in the hash table, so we try to claim the

empty bucket to insert the data (line 6). If the atomic `cas` succeeds, then we know for sure that we are the exclusive owner of the bucket (due to atomic `cas`) and we write the data (line 7) and release the lock (line 8). Since we assume a system architecture that has total store ordering (memory writes are not reordered, Section 1.3), we know that when the lock is released for other threads, then the data they read is also the correct data. If the atomic `cas` fails, then the bucket is already claimed by some other worker, since atomic `cas` can only fail if another worker succeeded in changing the state from `empty` to `locked`. In that case, we need to reload our knowledge of the bucket (line 10). Either this worker is inserting data with the same hash, or not, which we check at line 11. If this is not the case, then we can simply continue with the next bucket in the probe sequence. If this is the case, then we need to wait until the lock is released (line 12), after which we compare the data to see if the other worker inserted our data or not (line 13).

We provide a short informal proof of correctness, by showing that a) it is not possible that `find-or-insert` returns the same value for different data, and that b) it is not possible that `find-or-insert` returns a different value for the same data.

By inspection of Algorithm 4.1 we see that the algorithm only returns a value `s` if the value of `darray[s]` equals `data` and that `harray[s]` contains `filled(h)`. Furthermore, it is not possible that the contents of `darray[s]` change later. The contents of `darray[s]` can only be modified by a thread that succeeds the atomic `cas` at line 6, i.e., after `harray[s]` is originally `empty`, and `harray[s]` cannot be set to `empty` after an operation succeeded at line 6 and returned `s`. Therefore, it is not possible that `find-or-insert` returns the same value `s` for different data.

Furthermore, we assume that `probe-sequence` is always the same for given data, i.e., that every `find-or-insert` operation on the same data visits the same buckets in the same order. By inspection of Algorithm 4.1, we state that it is impossible for the algorithm to go to the next bucket unless it is certain that the bucket contains different data: the algorithm only proceeds to the next bucket if the hash values are different (thus the bucket contains different data) or if a direct comparison of the data after the lock is released reveals that the data is different. Consider two threads that insert the same data and the first thread succeeds. If the second `find-or-insert` operation returns a different value for the same data, then the second thread inserts the data in a bucket that is either before or after the bucket where the first thread inserted the data. If it is inserted in an earlier bucket, then the first thread skipped this bucket earlier, and if it is inserted in a later bucket, then the second thread skipped the existing bucket, which is not possible, since the algorithm cannot go to the next bucket unless it is certain that it contains different data. Therefore, it is

not possible that `find-or-insert` returns a different value for the same data.

Note that the loop at line 12 is blocking. Therefore the algorithm is not lock-free. However, the loop only blocks for concurrent threads that insert data with the same hash, which is a rare event in practice. [LPW10] states that a mechanism could be implemented that ensures local progress (making the algorithm wait-free), but that this is not needed, since the writing locks are rarely hit under normal operation.

4.2.2 Variant 1: Reference counter and tombstones

This subsection describes the lock-less hash table that we used in the initial implementation of Sylvan, presented in [DLP13]. The initial implementation of Sylvan used a different approach to garbage collection than is described in Section 2.2.5. The initial version used reference counting with the reference counter for each node embedded in the hash table, whereas the later versions use external data structures to manage which decision diagram nodes must be kept during garbage collection.

Implementations of decision diagrams like CUDD [Som15] have a reference count variable as a part of each decision diagram node, which is used both for internal referencing (within operations) and external referencing (by the user of the implementation). We therefore initially extended the hash table described in Section 4.2.1 with a reference counter in the hash array, reserving 1 bit for the wait lock, 15 bits for the hash, and 16 bits for the reference counter.

In addition, we extended the hash table with a mechanism for deleting decision diagram nodes, as garbage collection is essential for the manipulation of decision diagrams. Node deletion causes a challenge with probe sequences, as nodes earlier in the probe sequence of some node may be deleted. If we simply delete nodes by setting the bucket to `empty`, then later `find-or-insert` calls find this empty bucket and insert the node again, resulting in node duplication. One option is to reinsert all nodes into the hash table during garbage collection, which is problematic since we derive the node identities from their location in the hash table, which would change upon reinsertion. Another option is to use so-called tombstones for deleted buckets. When buckets are deleted, they are not set to `empty`, but to a state `tombstone`. The `find-or-insert` algorithm can overwrite the tombstones to insert data, but must follow the probe sequences until an actual empty bucket or the end of the probe sequence is found to determine whether the data is already in the table. Empty buckets are buckets that have never stored data, whereas tombstones are empty buckets that have at some time contained data.

In this extension of the hash table, presented in [DLP13], each bucket is in one the following states, manipulated using atomic `cas`:

```
1  def find-or-insert(data):
2      h ← hash(data)
3      for s ∈ probe-sequence(data) :
4          V ← harray[s]
5          if V = empty : return insert(h, data)
6          elif V.hash = h :
7              while V = locked(h) : V ← harray[s]
8              if darray[s] = data :
9                  while not cas(harray[s], V, filled(V.hash, V.refcount+1)) :
10                     V ← harray[s]
11                 return s
12     raise TableFull
13 def insert(h, data):
14     for s ∈ probe-sequence(data) :
15         V ← harray[s]
16         if V = empty ∨ V = tombstone :
17             if cas(harray[s], V, locked(h)) :
18                 darray[s] ← data
19                 harray[s] ← filled(h, 1)
20                 return s
21             else: V ← harray[s]
22         if V.hash = h :
23             while V = locked(h) : V ← harray[s]
24             if darray[s] = data :
25                 while not cas(harray[s], V, filled(V.hash, V.refcount+1)) :
26                     V ← harray[s]
27                 return s
28     raise TableFull
```

Algorithm 4.2 Algorithm for parallel `find-or-insert` of the lock-less hash table.

- `empty` for unused buckets.
- `locked(hash)` for buckets that are about to contain data with hash `hash`.
- `filled(hash, count)` for buckets that contain data with hash `hash` and with `count` references.
- `tombstone` for unused buckets that earlier stored data.

During the normal operation of the hash tables, buckets can only move from the `empty` and `tombstone` state to the `locked` state, and from the `locked` state to the `filled` state. During garbage collection, buckets can be moved to the `tombstone` state.

See Algorithm 4.2 for the implementation of the `find-or-insert` operation.

The algorithm consists of two phases, a "find" phase and an "insert" phase. First, the algorithm tries to find either an `empty` bucket which signifies that the node is not yet in the table, or a node with the same hash. This phase is similar to Algorithm 4.1. The main difference is that if we find the data in the table, then we increase the reference counter using atomic `cas` at line 9. If this atomic `cas` fails, we simply reload the contents of the bucket and try again. The second phase is similar to the first phase (and Algorithm 4.1), but now we insert the data when we encounter a bucket that is `empty` or a `tombstone`.

There is a separate stop-the-world phase for garbage collection, in which no nodes are inserted. No nodes are deleted outside of this separate phase. This means that no tombstones are created during the execution of `find-or-insert`. We also experimented with a version of this hash table where garbage collection and node insertion can occur simultaneously. This requires additional locking on the first bucket of the probe sequence to avoid the scenario where two threads want to insert the same node after finding that the node is not yet in the table, one thread succeeds first, then a third thread deletes a node earlier in the probe sequence, and the second thread inserts the node at that location, resulting in node duplication.

Algorithm 4.2 is correct for the same reason that Algorithm 4.1 is correct, with the additional distinction that the `tombstone` does not signify that certain data is not in the table, but it can be reused for inserting data. An additional property for this particular variant is that `find-or-insert` must always set the reference count to 1 for newly inserted data and increase it by 1 when finding existing data. The former is trivial to prove (line 19) and the latter follows from the loops at lines 9–10 and lines 25–26.

Note that the `cas`-loops in lines 9–10 and lines 25–26 are lock-free (always a thread progresses if the `cas` operation fails), whereas the loops in line 7 and line 23 are light-weight blocking: they are rarely hit, since they only block threads that insert data with the same hash.

4.2.3 Variant 2: Independent locations

In [DP15], we implemented mark-and-sweep garbage collection in Sylvan and designed a hash table without reference counting and with independent locations for the bucket in the hash array and in the data array. The idea is that the location of the decision diagram node in the data array is used for the node identifier and that nodes can be reinserted into the hash array without changing the node identifier.

If we have a separate garbage collection phase that clears the hash array and reinserts the nodes that we want to keep, then tombstones are no longer necessary. Tombstones have a significant disadvantage, as over time all empty

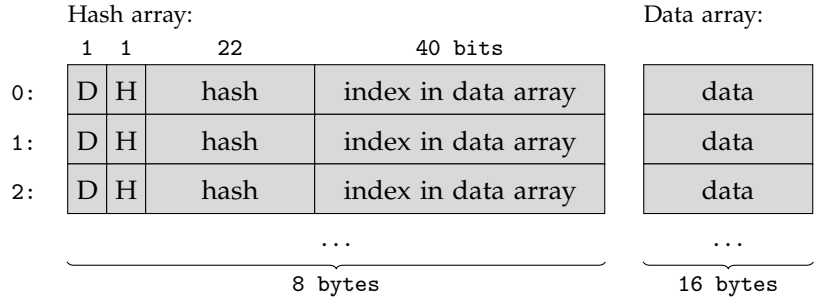


Figure 4.3 Layout of the hash table with hash array `h` and data array `d`. The field D of hash bucket i controls whether the data bucket i is used; the field H of hash bucket i controls whether the hash bucket i is used, i.e., the fields hash and index.

buckets become tombstones, which forces every `find-or-insert` operation to check the entire probe sequence even if the table is actually nearly empty, as most empty buckets are now tombstones. With operations on decision diagrams, often garbage collection removes many nodes (e.g. 80%-90% of the table), which all become tombstones.

See Figure 4.3 for the layout of this hash table. Note that we now require larger buckets in the hash array, mainly to store the location of the hashed data in the data array. The bucket in the hash array can be in the following states:

- `empty` (`D` $= 0$, `H` $= 0$) when both the bucket in the hash array and the corresponding bucket in the data array are not used.
- `dfilled` (`D` $= 1$, `H` $= 0$) when only the corresponding bucket in the data array is filled.
- `hfilled(hash, index)` (`D` $= 0$, `H` $= 1$) when only the bucket in the hash array is filled.
- `dhfilled(hash, index)` (`D` $= 1$, `H` $= 1$) when both buckets are filled.

All modifications to buckets in the hash array must be performed with atomic `cas` operations. Note that the field `D` can be set to 1 and to 0 by `find-or-insert` as discussed below, but the field `H` can only be set from 0 to 1.

See Algorithm 4.3 for `find-or-insert`. The `find-or-insert` algorithm now consists of three phases: a find phase, then a phase that inserts the data in the data array, and finally a phase that inserts the data into the hash array. Data is inserted in the data array first and then in the hash array. If some other worker concurrently inserted the same data after we inserted the data in the data array, then we undo this operation by setting the field D back to 0 using atomic `cas` (lines 28–31). Note that this implementation no longer needs a writing lock.

The algorithm first searches through the probe sequence to either find an

```
1  def find-or-insert(data):
2      h ← hash(data)
3      for s ∈ probe-sequence(data) :
4          V ← harray[s]
5          if V.H = 0 : return insert(h, data)
6          elif V.hash = h ∧ darray[V.index] = data :  return V.index
7      raise TableFull
8  def insert(h, data):
9      for d ∈ data-sequence(data) :
10         V ← harray[d]
11         if V.D = 0 ∧ cas(harray[d], V, V with [D:=1]) :
12             darray[d] ← data
13             return insert2(h, d, data)
14     raise TableFull
15 def insert2(h, d, data):
16     for s ∈ probe-sequence(data) :
17         V ← harray[s]
18         if V.H = 0 :
19             loop:
20                 if cas(harray[s], V, ⟨V.D, 1, h, d⟩) :  return d
21                 V ← harray[s]
22                 if V.H = 1 : break
23         if V.h = h ∧ darray[V.index] = data :
24             uninsert(d)
25             return V.index
26     uninsert(d)
27     raise TableFull
28 def uninsert(d):
29     loop:
30         V ← harray[d]
31         if cas(harray[d], V, V with [D:=0]) : return
```

Algorithm 4.3 Algorithm for parallel `find-or-insert` of the lock-less hash table.

empty bucket (H = 0) signifying the data is not yet in the table, or a matching bucket. If the data is not yet in the table, then the `insert` function continues the procedure by finding an empty bucket in the data array and claiming that bucket using atomic `cas` setting `D` to 1. If this is successful, then we continue with `insert2` to insert the data into the hash array, which uses an atomic `cas` to insert the data in the hash array. We perform this atomic `cas` in a loop because other threads might be manipulating the field `D`. In case another worker has inserted the same data, we rollback the operation by unsetting the field D from the bucket in the data array. We also need to rollback if we exhaust the probe sequence.

The `insert` method that inserts the data into the data array uses a different probe sequence. In particular this data sequence does not need to be deterministic. In our implementation, we simply let every thread start at a different position in the hash table and search linearly (with a bounded number of buckets). Also, small experiments with random probing resulted in performance deterioration. Our hypothesis is that the operating system automatically allocates new memory pages on the memory closest to the processor on which the thread resides that is writing to the memory page for the first time. However, we did not further investigate this.

The informal proof of correctness follows along the same lines as for the previous variants. Again, it is not possible to continue to the next bucket in the probe sequence of the hash array unless it is determined that the bucket contains other data. It is also not possible that data is changed after its "definitive" insertion in the hash array (line 20), even though the same bucket in the data array can be used many times for speculative insertion.

The `find-or-insert` operation of Algorithm 4.3 is wait-free (although with a high bound). To support this, we consider all loops in the algorithm.

- The for-loops at line 3, line 9 and line 16 are bounded as the probe sequence and the data sequence are bounded.
- The `cas`-loop at lines 19–22 is technically wait-free, although the bound is very high. The `cas` operation fails when another thread has modified the fields H, hash and index (also at line 20), but this can occur at most once and it breaks the loop. The `cas` operation also fails when another thread has modified the field D, which can occur at most twice as often as there are buckets in the hash table, since the number of times field D can change from 1 to 0 is bounded by the size of the hash table.
- The `cas`-loop at lines 29–31 is wait-free, as the number of times the `cas` operation can fail is at most once, which is when another thread succeeds writing H, hash and index (at line 20).

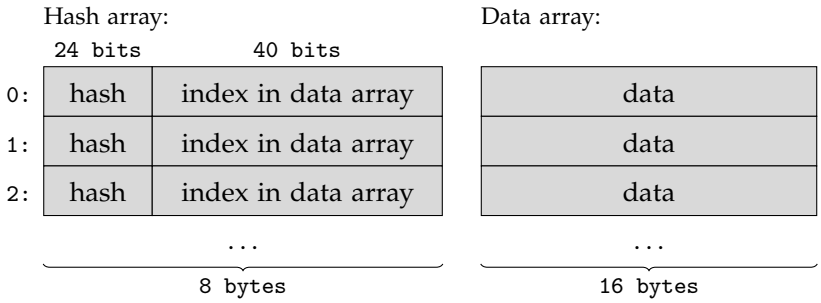


Figure 4.4 Layout of the hash array and data array in the final hash table design.

4.2.4 Variant 3: Using bit arrays to manage the data array

The hash table above has the drawback that the speculative insertion and uninsertion into the data array requires atomic `cas` operations, once for the insertion, once for the uninsertion. We present the final version of this hash table in [DP16b]. Instead of using a field D in the hash array, we use a separate bit array `databits` to implement a parallel allocator for the data array. Furthermore, to avoid having to use `cas` for every change to `databits`, we divide this bit array into regions, such that every region matches exactly with one cacheline of the `databits` array, i.e., 512 buckets per region if there are 64 bytes in a cacheline, which is the case for most current architectures. Every worker has exclusive access to one region, which is managed with a second bit array `regionbits`. Only changes to `regionbits` (to claim a new region) require an atomic `cas`. The new version therefore only uses normal writes for insertion and uninsertion into the data array, and only occasionally an atomic `cas` during speculative insertion to obtain exclusive access to the next region of 512 buckets.

A claimed region is not given back until garbage collection, which resets claimed regions. On startup and after garbage collection, the `regionbits` array is cleared and all threads claim an initial region using the `claim-next-region` method in Algorithm 4.4. All threads start at a different position (distributed over the entire table) for their first claimed region, to minimize the interactions between threads. The `databits` array is empty at startup and during garbage collection threads use atomic `cas` to set the bits in `databits` of decision diagram nodes that must be kept in the table. In addition, the bit of the first bucket is always set to 1 to avoid using the index 0 since this is a reserved value in Sylvan.

The layout of the hash array and the data array is given in Figure 4.4. We

```
1  def find-or-insert(data):
2      index ← 0
3      h ← hash(data)
4      for s ∈ probe-sequence(data) :
5          V ← harray[s]
6          if V = 0 :
7              if index = 0 :
8                  index ← reserve-data-bucket()
9                  darray[index] ← data
10             if cas(harray[s], 0, {h, index}) : return index
11             else: V ← harray[s]
12         if V.hash = h ∧ darray[V.index] = data :
13             if index ≠ 0 : free-data-bucket(index)
14             return V.index
15     raise TableFull
16 def reserve-data-bucket():
17     loop:
18         if myregion has a bit set to 0 :
19             i ← first bit in myregion that is 0
20             set-bit(databits, 512 × myregion + i, 1)
21             return 512 × myregion + i
22         else: myregion ← claim-next-region(myregion)
23 def free-data-bucket(d):
24     set-bit(databits, d, 0)
25 def claim-next-region(oldregion):
26     newregion ← (oldregion + 1) mod (tablesize/512)
27     while newregion ≠ oldregion :
28         loop:
29             if the bit for newregion is 1 : break
30             if set-bit-cas(regionbits, newregion, 0, 1) : return newregion
31         newregion ← (newregion + 1) mod (tablesize/512)
32     raise TableFull
```

Algorithm 4.4 Algorithm for parallel `find-or-insert` of the hash table, with 512 buckets per region. The variable `myregion` is a thread-specific variable.

also remove the field H, which is obsolete as we use a hash function that never hashes to 0 and we forbid nodes with the index 0 because 0 is a reserved value in Sylvan. The fields hash and index are therefore never 0, unless the hash bucket is `empty`, so the field H to indicate that hash and index have valid values is not necessary. Manipulating the hash array bucket is also simpler, since we no longer need to take into account changes to the field D.

Inserting data in the hash table consists of three steps. First the algorithm tries to find whether the data is already in the table. If this is not the case, then a new bucket in the data array is reserved in the current region of the thread with the `reserve-data-bucket` function. If the current region is full, then the thread claims a new region with the `claim-next-region` function. Note that it may be possible that the next region contains used buckets, if there has been a garbage collection earlier. Afterwards the new bucket is inserted in the hash array. Sometimes, the data has been inserted concurrently (by another thread) and then the bucket in the data array is freed again with the `free-data-bucket` function, so it is available the next time the thread wants to insert data.

The main method of the hash table is `find-or-insert`. See Algorithm 4.4. The algorithm uses the local variable "index" to keep track of whether the data is inserted into the data array. This variable is initialized to 0 (line 2) which signifies that data is not yet inserted in the data array. For every bucket in the probe sequence, we first check if the bucket is empty (line 6). In that case, the data is not yet in the table. If we did not yet write the data in the data array, then we reserve the next bucket and write the data (lines 7–9). We use atomic `cas` to insert the hash and index into the hash array (line 10). If this is succesful, then the algorithm is done and returns the location of the data in the data array. If the `cas` operation fails, some other thread inserted data here and we refresh our knowledge of the bucket (line 11) and continue at line 12. If the bucket is not or no longer empty, then we compare the stored hash with the hash of our data, and if this matches, we compare the data in the data array with the given input (line 12). If this matches, then we may need to free the reserved bucket (line 13) and we return the index of the data in the data array (line 14). If we finish the probe sequence without inserting the data, we raise the TableFull signal (line 15).

The `find-or-insert` method relies on the methods `reserve-data-bucket` and `free-data-bucket` which are also given in Algorithm 4.4. They are fairly straightforward.

The `claim-next-region` method searches for the first 0-bit in the `regionbits` array. The value `tablesize` here represents the size of the entire table. We use a simple linear search and a `cas`-loop to actually claim the region. Note that we may be competing with threads that are trying to set the bit of a different region, since the smallest range for the atomic `cas` operation is 1 byte or 8 bits.

The informal proof of correctness of this algorithm follows along the same lines as before. We establish that threads truly have exclusive access to regions by inspecting `claim-next-region`. From this we can argue that once a thread inserts new data and returns, the data will not change. Furthermore, we can again establish that `find-or-insert` only proceeds to the next bucket in the probe sequence if establishes that either the current bucket has a different hash or that the associated data bucket contains different data.

The algorithms in Algorithm 4.4 are wait-free. The `claim-next-region` method is wait-free, since the number of `cas` failures is bounded: regions are only claimed and not released (until garbage collection), and the number of regions is bounded, so in principle the maximum number of cas failures is the number of regions. The `free-data-bucket` is trivially wait-free: there are no loops. The `reserve-data-bucket` method contains a loop, but since `claim-next-region` is wait-free and the number of times `claim-next-region` returns a value instead of raising the TableFull signal is bounded by the number of regions, `reserve-data-bucket` is also wait-free. Finally the `find-or-insert` method only relies on wait-free methods and has only one for-loop (line 4) which is bounded by the number of items in the probe sequence. It is therefore also wait-free.

4.2.5 Comparing the three variants

This section contains an informal analysis of the number of memory writes and atomic `cas` operations that are minimally needed for various operations on the nodes table. The real-world performance of variants 2 and 3 is compared using the LTSmin toolset in Section 5.5.5 in the next chapter.

Expected cost in number of operations The main difference between the variants of the hash tables is in the number of cachelines they read and write and the number of atomic `cas` operations. See Table 4.1. We consider the costs of finding existing nodes and inserting new nodes. Since inserting data is a two-step process in variants 2 and 3, we also consider the cost of inserting a node, then discovering it is already added simultaneously by another thread, and undoing the insertion.

For variant 1, we also add the cost of dereferencing nodes in the future, which uses atomic `cas` to decrease the reference count by 1. This is necessary for the vast majority of created decision diagram nodes, for typical applications. The cost of dereferencing nodes for variants 2 and 3 is not part of the table, but depends on the application. For example, node lists can be maintained on the program stack (relatively cheap) or using BDD "protected" variables (only memory write for the initialization of the variable, see Section 2.2.5).

	Cost of finding existing node
Original	Only memory reads, at least 2 cachelines (1+ in the hash array, 1 in the data array)
Variant 1	In addition to memory reads: • 1 `cas` to increase the reference count • 1 `cas` in the future to decrease the reference count
Variant 2	Only memory reads, at least 2 cachelines (1+ in the hash array, 1 in the data array)
Variant 3	Only memory reads, at least 2 cachelines (1+ in the hash array, 1 in the data array)
	Cost of inserting new node
Original	• 1 `cas` in the hash array • 1 write (16 bytes) in the data array • 1 write on the same cacheline in the hash array **Total**: 2 cachelines, 1 `cas`, 2 write
Variant 1	• 1 `cas` in the hash array • 1 write (16 bytes) in the data array • 1 write on the same cacheline in the hash array • 1 `cas` in the future to decrease the reference count **Total**: 2 cachelines, 2 `cas`, 2 write
Variant 2	• 1 `cas` in the hash array for the data bucket • 1 write (16 bytes) in the data array • 1 `cas` in the hash array for the hash bucket • possibly more cachelines searching for an empty data bucket **Total**: 3 cachelines, 2 `cas`, 1 write
Variant 3	• 1 write in the array `databits` • 1 write (16 bytes) in the data array • rarely 1 `cas` in the array `regionbits` • 1 `cas` in the hash array for the hash bucket **Total**: 3 cachelines, 1 `cas`, 2 write, rarely +1 `cas` on +1 cacheline
	Cost of undoing inserting a node
Original	–
Variant 1	–
Variant 2	• 1 `cas` in the hash array field D to undo insertion
Variant 3	• 1 write in the array `databits` to undo insertion

Table 4.1 Comparison of cost in accessed cachelines, memory writes and atomic `cas` operations for typical calls to `find-or-insert`

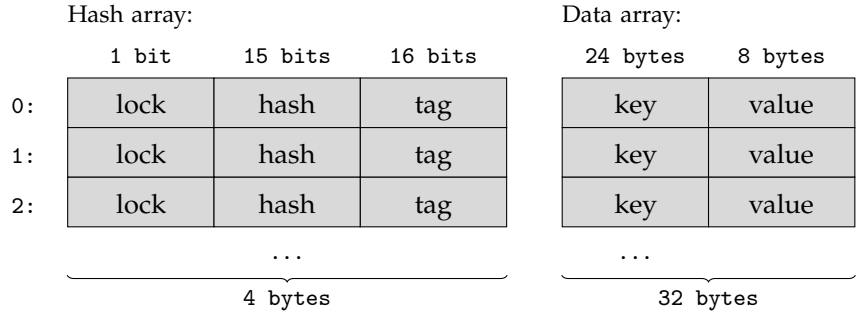


Figure 4.5 Layout of the operation cache.

Table 4.1 assumes that all atomic `cas` operations are successful; in all cases when the `cas` fails, the `cas` operation is performed again until it is successful. There is also some additional cost from following the probe sequence that is the same for all variants and not included here. As we see in Table 4.1, variants 2 and 3 touch an additional cacheline to either set the field D in the hash array, or to set the bit in the `databits` array. Typically we want to avoid `cas` operations, as they are not only more expensive than normal writes and imply a memory fence, but also because they need to be restarted when the `cas` operation fails. The main difference between these two variants is that a `cas` operation is replaced by a normal write (which is cheaper), both for inserting the node and for undoing that insertion if needed, and that only sometimes a new region is claimed using `cas` in variant 3. An application-specific performance comparison of variants 2 and 3 is compared in Section 5.5.5.

Memory cost The original hash table requires 20 bytes per bucket (16 bytes data, 4 bytes in the hash array). Variant 1 also requires 20 bytes per bucket. Variant 2 requires 24 bytes per bucket (8 bytes in the hash array). Variant 3 requires 24 bytes and $1 + 1/512$ bits for the bit array per bucket.

4.3 Operation cache

The operation cache is a hash table that stores intermediate results of BDD operations. It is well known that an operation cache is required to reduce the worst-case time complexity of BDD operations from exponential time to polynomial time. See also Section 2.2.4.

We use an operation cache which, like the hash tables described in Section 4.2, consists of two arrays: the hash array and the data array. See Figure 4.5 for the layout.

```
1 def cache-put(key, value):
2     h, location ← hash(key)
3     s ← harray[location]
4     if s.lock : return
5     if s.hash = h : return
6     if not cas(harray[location], s, {1, h, s.tag + 1}) : return
7     darray[location] ← {key, value}
8     harrray[location] ← {0, h, s.tag + 1}
```

Algorithm 4.5 The `cache-put` algorithm.

Since we implement a lossy cache, the design of the operation cache is extremely simple. We do not implement a special strategy to deal with hash collisions, but simply overwrite the old results. There is a trade-off between the cost of recomputing operations and the cost of synchronizing with the cache. For example, the caching granularity (see Section 2.2.4) increases the number of recomputed operations but improves the performance in practice.

The most important concern for correctness is that every result obtained via `cache-get` was inserted earlier with `cache-put`, and the most important concern for performance is that the number of memory accesses is as low as possible. To ensure this, we use a 16-bit "tag" counter that increments (modulo 4096) with every update to the bucket, and check this value before reading the cache and after reading the cache to check if the obtained result is valid. The chance that this tag counter is the same for a different result is astronomically small, as this requires exactly 4096 `cache-put` operations on the same bucket by other workers between the first and the second time the tag is read in `cache-get`, and the last of these 4096 other operations must have the same hash value but different data.

We reserve 24 bytes of the bucket for the operation and its parameters. We use the first 64-bit value to store a BDD parameter and the operation identifier. The remaining 128 bits store other parameters, such as up to two 64-bit values, or up to three BDDs (123 bits, with 41 bits per BDD with a complement edge). The same holds for MTBDDs and LDDs. The result of the operation can be any 64-bit value or a BDD. Note that with 32 bytes per bucket and a properly aligned array, accessing a bucket requires only 1 cacheline transfer.

See Algorithms 4.5 and 4.6 for the `cache-put` and `cache-get` algorithms.

The algorithms are quite straight-forward. We use a 64-bit hash function that returns sufficient bits for the 15-bit `h` value and the `location` value. The `h` value is used for the hash in the hash array, and the `location` for the location of the bucket in the table. The `cache-put` operation aborts as soon as some problem arises, i.e., if the bucket is locked (line 4), or if the hash of the stored

```
1 def cache-get(key):
2     h, location ← hash(key)
3     s ← harray[location]
4     if s.lock : return ⊥
5     if s.hash ≠ h : return ⊥
6     storedkey, value ← darray[location]
7     if storedkey ≠ key : return ⊥
8     if s ≠ harray[location] : return ⊥
9     return value
```

Algorithm 4.6 The `cache-get` algorithm.

key matches the hash of the given key (line 5), or if the `cas` operation fails (line 6). If the `cas` operation succeeds, then the bucket is locked. The key-value pair is written to the cache array (line 7) and the bucket is unlocked (line 8, by setting the locked bit to 0).

In the `cache-get` operation, when the bucket is locked (line 4), we abort instead of waiting for the result. We also abort if the hashes are different (line 5). We read the result (line 6) and compare the key to the requested key (line 7). If the keys are identical, then we verify that the cache bucket has not been manipulated by a concurrent operation by comparing the "tag" counter (line 8).

It is theoretically possible that between lines 6–8 of the `cache-get` operation, exactly 4096 `cache-put` operations are performed on the same bucket by other workers, with at least one of these such that the comparison at line 7 succeeds. The chances of this occurring are astronomically small. The reason we choose this design is that this implementation of `cache-get` only reads from memory and never writes. Memory writes cause additional communication between processors and with the memory when writing to the cacheline, and also force other processor caches to invalidate their copy of the bucket. We also want to avoid locking buckets for reading, because locking often causes bottlenecks. Since there are no loops in either algorithm, both algorithms are wait-free.

4.4 Conclusion and Discussion

This chapter presented the data structures that we have implemented in Sylvan for the unique table and the operation cache. Because Sylvan strictly separates garbage collection and possible table resizing from the normal operation with `find-or-insert`, the table designs can be kept simple.

The designs presented in this chapter follow the evolution of Sylvan as a research implementation of parallel decision diagrams, and were mainly evaluated with the model checking toolset LTSmin (see Chapter 5). We have

a comparison of variants 2 and 3 using a large benchmark set for the model checking toolset LTSmin in Section 5.5.5.

One direction for future research is to look at different architectures. We focus on multi-core systems, but distributed systems are also popular. Recently, Oortwijn et al. [ODP15] studied the performance of a distributed hash table design, which uses a shared memory abstraction with Infiniband and remote direct memory access. This paper is part of the research by Oortwijn into parallelizing BDD operations on distributed systems, similar to our approach on multi-core systems.

There are many options to further improve or study the hash tables presented here. An open question is how large exactly the cost is of not waiting in `cache-get` until a thread that is writing a result is finished. Also, we did not really look at the performance of the hash function, while this may be very important for good performance. It may be interesting to consider an operation cache that looks at multiple buckets (maybe several buckets on the same cacheline) and if none of them contain the data, uses the bucket with the smallest value of the tag counter to write the result. Currently, garbage collection removes all results from the cache, but it may be interesting to study whether some results can be kept in the table if garbage collection only removes a small number of nodes. Another interesting question would be whether more intelligent garbage collection, for example generational garbage collection might be useful with some support from the nodes table. We tried to keep the data structures in this chapter as "stupid" and simple as possible, but maybe more intelligent approaches could improve the performance further in the future.

Chapter 5

Application: State space exploration

The main application for which we developed parallel decision diagram operations is symbolic model checking. In model checking, systems are modeled as sets of possible states of the system and transitions between these states. System states are typically represented by Boolean vectors. In symbolic model checking, rather than treating and storing these states individually, sets of states are represented by Boolean functions stored using decision diagrams.

Fixed point algorithms, which are procedures that repeatedly apply some operation until a fixed point is reached, play a central role in many model checking algorithms. An example of a fixed point algorithm is state space exploration ("reachability"), which computes all states reachable from some initial state of the system, i.e., the transitive closure of the transition relation on some initial set of states. The algorithms in Figure 5.1 are well-known methods to compute this transitive closure, with the `closure-fs` algorithm using a so-called frontier set that only contains the newly discovered states in each iteration. Model checking algorithms depend on state space exploration to determine the number of states, to check if an invariant is always true, to


```
1 def closure(S, T):
2     S' ← ∅
3     while S ≠ S' :
4         S' ← S
5         S ← S ∪ relnext(S, T)
6     return S
```

```
1 def closure-fs(S, T):
2     F ← S
3     while F ≠ ∅ :
4         next ← relnext(F, T)
5         S, F ← next ∪ S, next \ S
6     return S
```

Figure 5.1 Algorithms `closure` and `closure-fs` compute the transitive closure of the transition relation $\mathcal{T}$ on the initial set of states $\mathcal{S}$.

find cycles and deadlocks, and so forth.

This chapter describes the implementation of on-the-fly state space exploration in the model checking toolset LTSmin (Section 5.1), and shows how we parallelize symbolic state space exploration using

1. the parallel decision diagram operations of Sylvan, including a specialised operation `relnext` (Section 5.2),
2. parallel transition learning with a special decision diagram operation `collect` that combines enumeration with set union (Section 5.3), and
3. high-level parallelism, using the disjunctive partitioning of transitions offered by LTSmin (Section 5.4).

We study the effects of these three consecutive steps on the parallel speed-up (Section 5.5) using benchmarks from the BEEM database [Pel07]. Section 5.6 concludes this chapter.

This chapter and the experimental results in this chapter are mainly based on the publications [DP15] and [DP16b]. For this thesis, we added Sections 5.1, 5.2 and 5.6, and also added new experiments in Section 5.5.2.

5.1 On-the-fly state space exploration in LTSmin

The Pins interface The model checking toolset LTSmin provides a language independent Partitioned Next-State Interface (Pins), which connects various input languages to model checking algorithms [BPW10; LPW11; DLP12; Kan+15; Mei+14]. In Pins, the states of a system are represented by vectors of N integer values. Furthermore, transitions are distinguished in K disjunctive "transition groups", i.e., each transition in the system belongs to one of these transition groups. The transition relation of each transition group usually only depends on a subset of the entire state vector called the "short vector". This enables the efficient encoding of transitions that only affect some integers of the state vector. Variables in the short vector are further distinguished by the notions of read dependency and write dependency [Mei+14]: the variables that are inspected or read to obtain new transitions are in the "read vector" of the transition group, and the variables that can be modified by transitions in the transition group are in the "write vector". An example of a variable that is only in the read vector is a guard; when a variable is only in the write vector, then its original value is irrelevant. Computing short vectors from long vectors is called "projection" in LTSmin and is similar to existential quantification.

Learning transitions Initially, LTSmin does not have knowledge of the transitions in each transition group, and only the initial state is known. As the model is explored, new transitions of each transition group are learned via the Pins interface and their projection is added to the transition relation. Every

```
1 def learn-transitions(states, k):
2     shorts ← project(states, readvariables[k])
3     shorts ← minus(shorts, visited[k])
4     visited[k] ← union(visited[k], shorts)
5     enumerate (shorts, readvariables[k], next-state-wrapper, k)
6 def next-state-wrapper(state, k):
7     next-state(state, k, add-transition)
8 def add-transition(source, target, k):
9     relations[k] ← union(relations[k], transition(source, target))
```

Algorithm 5.1 The algorithm for on-the-fly learning in LTSmin.

Pins language module implements a `next-state` function. This `next-state` function takes as input the source state (read vector), a transition group, a callback function, and optional parameters that are given to the callback function. For every transition from the short state, the callback function is called with as input the source state (read vector), the target state (write vector), the transition group, and the extra parameters that were given to the `next-state` function. In addition, for each transition group, LTSmin stores the set of read vectors for which `next-state` has been called. Algorithms in LTSmin thus learn new transitions on-the-fly. Internally, LTSmin offers various backends to store discovered states and transitions, including binary decision diagrams and list decision diagrams from Sylvan. With list decision diagrams, integers from the state vector can be used directly in the decision diagram nodes. With binary decision diagrams, the integers must be represented in binary form, typically using a fixed number of bits per integer.

The algorithm `learn-transitions` that learns new transitions is given in Algorithm 5.1. This algorithm is given a set of states, i.e., as a BDD or LDD, and the transition group k, and also uses global variables:

- `relations` is an array of BDDs or LDDs with the transition relation of each transition group;
- `readvariables` is an array that encodes for each transition group the variables that are in the read vector;
  - for BDDs, this is a cube of BDD variables
  - for LDDs, this is a singleton vector with the value 1 for every variable in the read vector, e.g., with 4 state variables, the singleton $\{\langle 0,1,1,0\rangle\}$ encodes that the 2nd and 3rd variable are in the read vector.
- `visited` is an array of BDDs or LDDs with the read vectors for which the transitions have already been computed.

First the set of (new) states is reduced to the short states (read vector) by abstracting from variables that are not in the read vector of transition group k (line 2). In LTSmin, this is called the projection of the set of long vectors onto the variables in the read vector, and it is similar to existential quantification. We remove the states from `shorts` that have been seen before (line 3) and update the visited set with the new states (lines 4). Finally, we enumerate all new short states (line 5). The `enumerate` function takes four parameters (three for LDDs): the set of states, the variables in the BDD (omitted for LDDs), the callback function, and a parameter for the callback function. `enumerate` calls the wrapper function `next-state-wrapper` for every state in the given set. This wrapper function calls the `next-state` function of the Pins language module (line 7), which calls the callback `add-transition` for every discovered transition. The `add-transition` callback then converts the transition to a singleton set with the method `transition` and adds this transition to the set `relation[k]` (line 9).

The advantage of this technique is that on-the-fly transition learning uses short vectors via Pins, implicitly learning many "long transitions" with every learned "short transition". While Pins is in essence an explicit-state interface, using the short (read and write) vectors makes Pins quite efficient for symbolic model checking. The learned transitions (explicit-state) are added to the transition relation first and then used to compute the successor states (as long vectors) symbolically.

The reachability algorithm The symbolic reachability algorithm with K transition groups and on-the-fly learning is given in Algorithm 5.2. This algorithm is an extension of the standard breadth-first-search (BFS) reachability algorithm with a frontier set (Figure 5.1). Algorithm 5.2 iteratively discovers new states until no new states are found (line 4). For every transition group (line 5), the transition group is updated with new transitions learned from the frontier set (line 6). The updated transition relation `relations[k]` is then used to symbolically find all successors of the states in the frontier set (line 7). This uses the `relnext` operation that is described in Section 2.3.3. The sets of new states discovered for every transition group are pair-wise merged into the new set `frontier` (line 8). Successors that have been found in earlier iterations are removed (line 9). All new states are then added to the set of discovered states `states` (line 10). When no new states are discovered, the set of discovered states is returned (line 11).

The algorithm `big-union` that pair-wise merges sets is given in Algorithm 5.3.

```
1  def reachable(initial):
       // global variables:  relations[K], K
2      states ← initial
3      frontier ← initial
4      while frontier ≠ ∅ :
5          for k ∈ {0,...,K−1} :
6              learn-transitions (frontier, k)
7              next[k] ← relnext (frontier, relations[k])
8          frontier ← big-union (next, 0, K)
9          frontier ← minus (frontier, states)
10         states ← union (states, frontier)
11     return states
```

Algorithm 5.2 Symbolic on-the-fly reachability algorithm (using a frontier set) with K transition groups. Computes the set of states reachable from the initial state. The transition relations are updated with on-the-fly learning (line 6).

```
1  def big-union(array, i, k):
2      if k = 1 : return array[i]
3      else:
4          left ← big-union(array, i, k/2)
5          right ← big-union(array, i + k/2, k − k/2)
6          return union(left, right)
```

Algorithm 5.3 Implementation of pair-wise merge algorithm `big-union`, which given an array of BDDs or LDDs, merges the k consecutive sets, starting at the ith element in the array.

5.2 Parallel operations in a sequential algorithm

To parallelize the symbolic reachability algorithm in LTSmin, we can simply use the parallel operations of Sylvan, without special modifications to the original algorithm. However, we use a non-parallel `enumerate` method (Algorithm 5.1, line 5), because the `add-transitions` callback (Algorithm 5.1, lines 8–9) is not thread-safe. If we use a parallel `enumerate` method, then multiple threads update the same global transition relation at the same time, resulting in a race condition.

5.3 Parallel learning

In order to exploit the automatic parallelization offered by Sylvan with a parallel `enumerate` method to parallelize transition learning, we need to modify the

```
1 def collect(states, variables, callback, vec={}, k):
2     if variables = ∅ : return callback(vec, k)
3     if states = false : return ∅
4     v, variables ← head(variables), tail(variables)
5     do in parallel:
6         low ← collect(states_{v=0}, variables, callback, vec+{0}, k)
7         high ← collect(states_{v=1}, variables, callback, vec+{1}, k)
8     return or(low, high)
```

Algorithm 5.4 The parallel `collect` algorithm for BDDs combining set enumeration and set union. The algorithm expects the BDD states, a cube of variables used for the enumeration, and a callback function. The callback is called for every element of the set and returns a BDD. The returned BDDs are pairwise merged (using `or`) and returned.

```
1 def collect(states, callback, vec={}, k):
2     if states = 1 : return callback(vec, k)
3     if states = 0 : return ∅
4     do in parallel:
5         right ← collect(states.right, callback, vec, k)
6         down ← collect(states.down, callback, vec+{states.value}, k)
7     return union(right, down)
```

Algorithm 5.5 The parallel `collect` algorithm for LDDs combining set enumeration and set union. The algorithm expects the LDD states and a callback function. The callback is called for every element of the set and returns an LDD. The returned LDDs are pairwise merged (using `union`) and returned.

design of the Pins callback wrappers in LTSmin. The `next-state` method implemented by the language modules remains the same, but we modify both the wrapper around the `next-state` function and the callback called by the `next-state` function to enable parallel learning. We modify the callback to put learned transitions in a temporary set and return this set, instead of updating a global transition relation. We implement a custom decision diagram operation `collect` that combines set enumeration with set union. See Algorithm 5.4 for BDDs and Algorithm 5.5 for LDDs.

By using this `collect` algorithm with the modified `next-state` wrapper instead of the sequential `enumerate` operation, we parallelize on-the-fly transition learning. See Algorithm 5.6 for the modified `learn-transitions` method that uses `collect` instead of `enumerate` (line 5) and adds the learned relations to the global transition relation (line 6). The modified `next-state-wrapper` uses a local temporary set `learned` (lines 8–10) that is updated by the `add-transition` method (line 12).

```
1  def learn-transitions(states, i, learned):
2      shorts ← project(states, readvariables[i])
3      shorts ← minus(shorts, visited[i])
4      visited[i] ← union(visited[i], shorts)
5      learned ← collect(shorts, readvariables[i], next-state-wrapper, i)
6      relations[i] ← union(relations[i], learned)
7  def next-state-wrapper(state, k):
8      learned ← ∅
9      next-state(state, k, add-transition, &learned)
10     return learned
11 def add-transition(source, target, k, learned):
12     learned ← union(learned, encode-transition(source, target, k))
```

Algorithm 5.6 Transition learning with the `collect` operation.

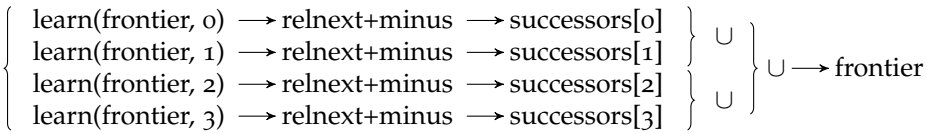


Figure 5.2 Schematic overview of parallel on-the-fly reachability.

5.4 Fully parallel on-the-fly symbolic reachability

Even with parallel decision diagram operations and parallel learning with the `collect` operation, the parallel speedup of model checking in LTSmin is limited, especially for smaller benchmark models. See Section 5.5.2 and Section 5.5.3 for benchmark results that show speedups of up to 20x with 48 cores for a few benchmark models, but slowdowns for many other benchmark models. We expect that performing reachability on these benchmark models results in mostly small "work units" (between sequential points in the algorithm) and insufficient parallelism inside each work unit.

Since LTSmin partitions the transition relation in transition groups, many small operations are executed in sequence, for each transition group. To improve the parallel speedups, we execute lines 5–8 of Algorithm 5.2 in parallel, as in Figure 5.2. We still perform the same decision diagram operations, but the number of sequential points is decreased and this increases the size of each work unit and also the amount of parallelism in the task tree. We therefore expect improved parallel speedup.

The fully parallel on-the-fly symbolic reachability algorithm is given in Algorithm 5.7. The new `par-next` method implements the parallelization for

```
1  def par-next(frontier, i, k):
2      if k = 1 :
3          learn-transitions (frontier, i)
4          next ← relnext(frontier, relations[i])
5          return next
6      else:
7          do in parallel:
8              left ← par-next(frontier, i, k/2)
9              right ← par-next(frontier, i + k/2, k − k/2)
10         return union(left, right)
11 def reachable(initial):
12     states ← initial
13     frontier ← initial
14     while frontier ≠ ∅ :
15         frontier ← par-next(frontier, 0, K)
16         frontier ← minus(frontier, states)
17         states ← union (states, frontier)
18     return states
```

Algorithm 5.7 Parallel symbolic on-the-fly reachability with K transition groups.

all transition groups. Lines 3–5 correspond with lines 6–7 in Algorithm 5.2. Lines 7–10 correspond with the `big-union` method.

5.5 Experimental evaluation

5.5.1 Experimental setup

We evaluate the application of parallelization to LTSmin. The experimental evaluation is based on the BEEM model database [Pel07]. Of these 300 benchmark models, 269 were successfully explored in [DP15]. The `plc` and `train-gate` models had an unfortunate parsing error in the Pins wrapper and we ignore these models in our evaluation. Several other models timed out (we use a timeout of 1200 seconds). We perform the experiments on a 48-core machine, consisting of 4 AMD Opteron™ 6168 processors with 12 cores each and 128 GB of internal memory.

We perform symbolic reachability using the LTSmin toolset using the following command:

```
dve2lts-sym -rgs --order=<order>
            --vset=<dd> <model>.dve
```

Most experiments use the LDD implementation of Sylvan (selected with `--vset=lddmc`) but in Section 5.5.6 we compare these results with the BDD implementation (selected with `--vset=sylvan`). We also select as size of the unique table 2^{30} buckets and as size of the operation cache also 2^{30} buckets. The parameter `--order` selects the variation of the algorithm that we use: `bfs-prev` for the sequential algorithm with parallel operations and `par-prev` for the fully parallel version. The `-prev` versions use a frontier set, whereas `bfs` and `par` do not use a frontier set. We perform the following experiments:

No	Section	Experiment
1	5.5.2	parallel operations, sequential learning with `enumerate` algorithm described in Section 5.2 (table variant 3, newest ltsmin)
2	5.5.3	parallel operations, parallel learning with `collect` algorithm described in Section 5.3 (table variant 2, oldest ltsmin)
3	5.5.4	fully parallel reachability algorithm algorithm described in Section 5.4 (table variant 2, oldest ltsmin)
4	5.5.5	fully parallel with nodes table variant 3 compare tables of Section 4.2.3 and Section 4.2.4 (table variant 3, old ltsmin)
5	5.5.6	fully parallel but with binary decision diagrams compare BDDs with LDDs (table variant 3, old ltsmin)

The experiments with the "oldest" version of LTSmin were performed for [DP15], whereas the experiments with the "old" version of LTSmin were performed for [DP16b]. The experiments with the "new" version of LTSmin were performed specifically for this thesis.

In all cases, we measure the time spent to execute symbolic reachability, excluding time spent initializing LTSmin, loading the models and computing the heuristic for the variable ordering (the `-rgs` parameter), which is the same for all models. The speedups given are the absolute speedups, but we also present the total time to run all experiments.

5.5.2 Experiment 1: Only parallel LDD operations

In Section 5.2 we discuss parallelizing LTSmin by only performing the decision diagram operations in parallel, but without modifying the sequential algorithm

Experiment 1	T_1	T_8	T_{48}	T_1/T_8	T_1/T_{48}
`firewire_link.1`	2.62	3.44	6.39	0.7	0.4
`anderson.1`	6.67	7.01	13.76	1.0	0.5
`firewire_tree.1`	2.57	2.73	3.67	0.9	0.7
`blocks.4`	555.97	98,23	39.08	5.7	14.2
`collision.5`	271.55	46.92	18.12	5.8	15.0
`lifts.8`	315.41	53.01	18.64	5.6	16.9
`exit.4`	418.25	67.03	20.33	6.2	20.6
`telephony.8`	737.76	116.61	34.96	6.3	21.1
Sum of all 269 models	12917	3649	2970	3.5	4.3

Table 5.1 Benchmark results (runtimes in seconds) for experiment 1. Each data point of T_1 and T_{48} is the average of at least 25 measurements; each data point of T_8 is the average of at least 2 measurements.

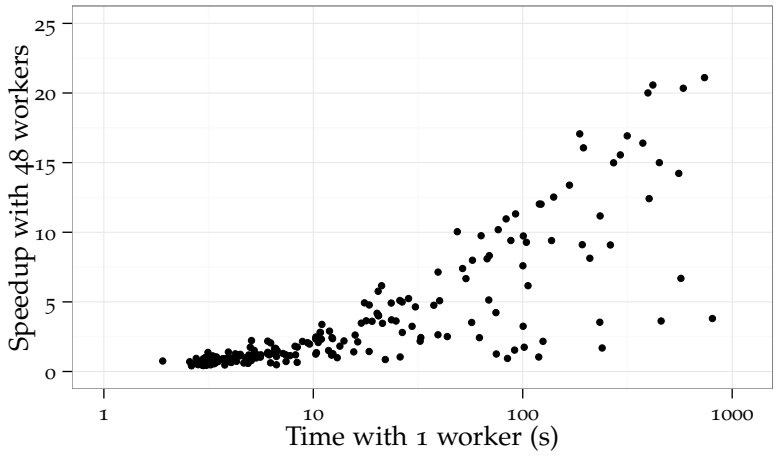


Figure 5.3 Plot of logarithmic time spent with 1 worker versus the obtained (absolute) speedup with 48 workers for the 269 benchmark models for experiment 1 (only parallel operations, but no parallel learning and no parallel reachability).

Experiment 2	T_1	T_{48}	T_1 / T_{48}
`firewire_link.1`	17.93	15.23	1.2
`anderson.1`	24.44	30.78	0.8
`firewire_tree.1`	16.43	11.35	1.4
`blocks.4`	630.04	21.69	29.0
`collision.5`	401.26	20.03	20.0
`lifts.8`	377.36	26.11	14.5
`exit.4`	440.66	20.67	21.3
`telephony.8`	843.06	31.10	27.1
Sum of all 269 models	20745	3737	5.6

Table 5.2 Benchmark results (runtimes in seconds) for experiment 2 (parallel operations and parallel learning, but no parallel reachability). Each data point is the average of at least 3 measurements.

in LTSmin, and using the sequential `enumerate` method for transition learning.

Table 5.1 summarizes the results for all 269 benchmark models and shows in particular the results of several selected models. We selected a few models that performed badly and a few models that performed well. The highest speedup was obtained with the `telephony.8` model, with a speedup of 21.1x. Unfortunately, 125 of the 269 benchmark models have a slowdown when executed with 48 workers, and 31 of the 269 benchmarks models have a slowdown already with 8 workers.

See also Figure 5.3. This plot shows that for this particular benchmark set, small models (under 10 seconds) have low parallel speedups (under 3x) or even slowdowns. Of course, this also means that many of these benchmarks are not very interesting for parallel execution on a 48-core machine, as they are done quickly enough (under 10 seconds) that they are not very interesting for parallelism anyway.

5.5.3 Experiment 2: Parallel learning

This subsection presents the experimental results using the `bfs-prev` variation in LTSmin, which uses `collect` that parallizes on-the-fly transition learning as described in Section 5.3.

Table 5.2 summarizes the results for all 269 benchmark models. These results were earlier published in [DP15] and were obtained with using an older version of LTSmin than the results in Section 5.5.2. Also, this version used the nodes table variant 2 presented in Section 4.2.3, while the results in Section 5.5.2 were

Experiment 3	T_1	T_{48}	T_1/T_{48}
`firewire_link.1`	17.96	1.57	11.5
`anderson.1`	24.43	15.45	1.6
`firewire_tree.1`	16.40	0.99	16.5
`blocks.4`	629.54	16.58	38.0
`collision.5`	401.38	12.97	31.0
`lifts.8`	377.52	12.03	31.4
`exit.4`	441.06	12.71	34.7
`telephony.8`	843.70	24.68	34.2
Sum of all 269 models	20756	1298	16.0

Table 5.3 Benchmark results (runtimes in seconds) for experiment 3 (parallel operations, parallel learning and parallel reachability). Each data point is the average of at least 3 measurements.

obtained with the nodes table variant 3 (with bitmaps) presented in Section 4.2.4. For this reason, the obtained runtimes are not directly comparable to the results in Section 5.5.2. However, compared to those results, we obtain better speedups here. The `blocks.4` model resulted in the highest speedup of 29.0x with 48 workers.

5.5.4 Experiment 3: Fully parallel reachability

This subsection presents the experimental results using the `par-prev` variation in LTSmin, which fully parallelizes on-the-fly symbolic reachability as described in Section 5.4.

Table 5.3 summarizes the results for all 269 benchmark models. These results were earlier published in [DP15] and were obtained using the same version of LTSmin and Sylvan as the results in Section 5.5.3. Model `blocks.4` results in the highest speedup of 38.0x. We also highlight the model `lifts.8` which has a speedup of 14.5x with `bfs-prev` and more than twice as high with `par-prev`. Also, models `firewire_link.1` and `firewire_tree.1` have much better speedup with the fully parallelized version. One of the largest improvements was obtained with the `firewire_tree.1` model, which went from 1.4x to 16.5x. The overhead (difference in T_1) between the "sequential" `bfs-prev` and "parallel" `par-prev` versions is negligible. For all models, the speedup either improves with the `par-prev` strategy, or stays the same.

For an overview of the obtained speedups on the entire benchmark set, see Figure 5.4. Here we see that "larger" models (higher T_1) are associated with a

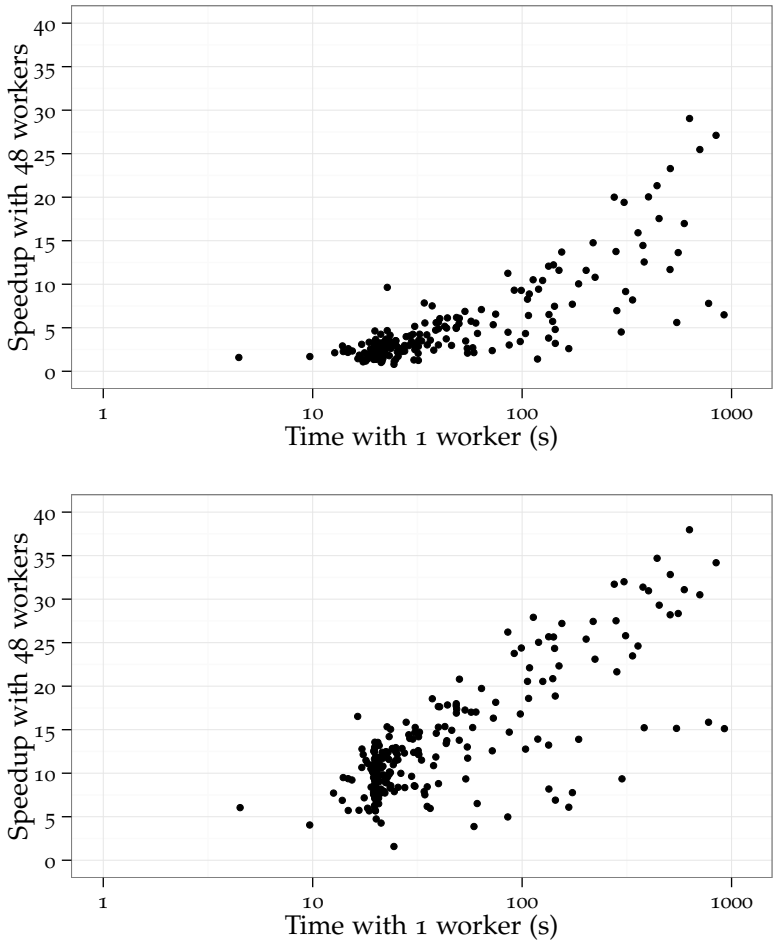


Figure 5.4 Plot of logarithmic time spent with 1 worker versus the obtained (absolute) speedup with 48 workers for the 269 benchmark models for experiment 2 (parallel learning, but no parallel reachability, above) and 3 (fully parallel, below).

Experiment 4	T_1	T_{48}	T_1/T_{48}
`firewire_link.1`	4.24	0.48	8.8
`anderson.1`	8.93	6.21	1.4
`firewire_tree.1`	4.23	0.30	14.1
`blocks.4`	635.86	17.27	36.8
`collision.5`	341.57	10.99	31.1
`lifts.8`	416.04	13.05	31.9
`exit.4`	494.85	13.95	35.5
`telephony.8`	915.61	28.18	32.5
Sum of all 269 models	16231	896	18.1

Table 5.4 Benchmark results (runtimes in seconds) for experiment 4 (fully parallel, with nodes table variant 3). Each data point is the average of at least 5 measurements.

higher parallel speedup. This plot also shows the benefit of adding parallellism on the algorithmic level, as many models in the fully parallel version have higher speedups. We conclude that the lack of parallelism is a bottleneck, which can be alleviated by exploiting the disjunctive partitioning of the transition relation.

5.5.5 Experiment 4: Comparing nodes table variants 2 and 3

We repeated the benchmarks on all 269 benchmark models using the `par-prev` variation with variant 2 of the nodes table presented in Section 4.2.3 and variant 3 presented in Section 4.2.4. See Table 5.4. We observe that the total benchmark set is run faster and result in a slightly improved parallel speedup. However, the results for individual models varies, for examples the model `collision.5` and the firewire models are performed faster, but the `telephony.8` and `exit.4` models are performed slower.

For a more insightful comparison of these results, see Figure 5.5. The results suggest that smaller benchmark models benefit more from the new hash table design. For the larger models, the difference between variants 2 and 3 is not very large, which is disappointing.

See Figure 5.6 for a speedup graph of a selection of the models with the highest speedups with variant 3 of the nodes table. The point of this speedup graph is that most likely further speedups would be obtained after 48 cores for the selected models.

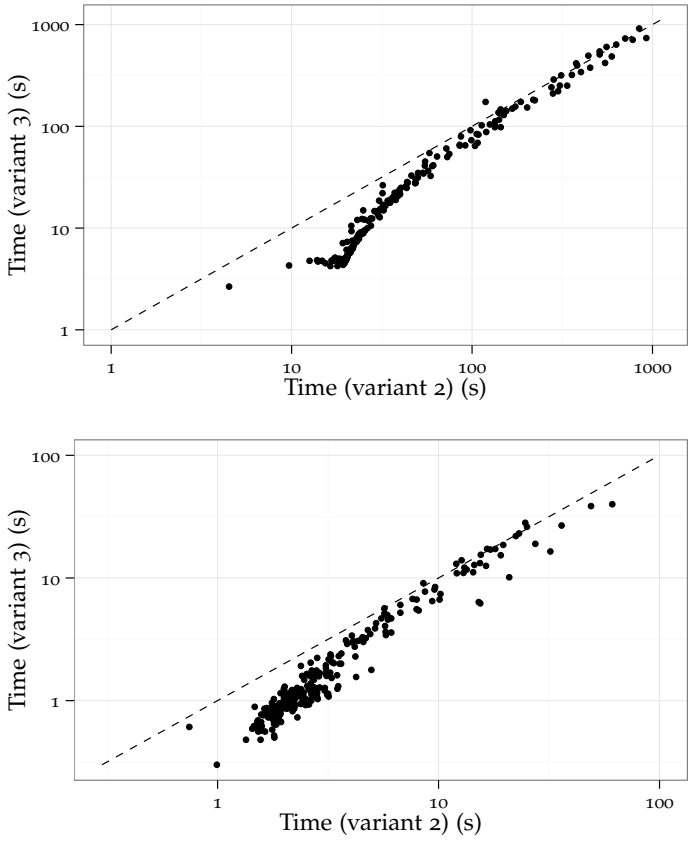


Figure 5.5 Comparison between the benchmark results of experiment 3 (with variant 2 of the nodes table) and experiment 4 (with variant 3 of the nodes table), for 1 worker (above) and for 48 workers (below).

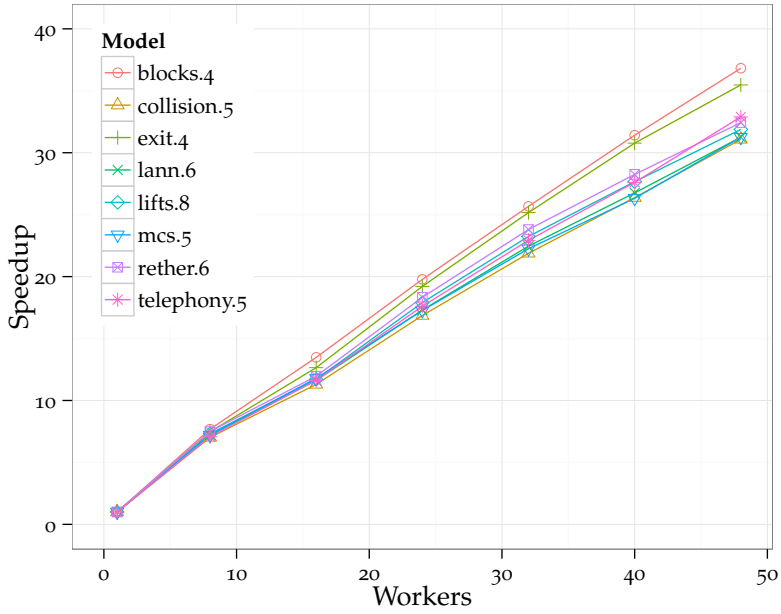


Figure 5.6 Speedup graphs of several well-performing models (experiment 4), using LDDs, the fully parallel strategy and nodes table variant 3. Each data point is an average of at least 5 measurements.

5.5.6 Experiment 5: Comparing BDDs and LDDs

Finally, we compared the performance of experiment 3 (fully parallel, using nodes table variant 2) with the same setup but using binary decision diagrams instead of list decision diagrams. One of the problems when using binary decision diagrams is selecting how many bits we need per integer in the state vector. For each model, we obtained the smallest number of bits per integer by experimentation and we used that for the experiments.

Figure 5.7 shows that the majority of models, especially larger models, are performed up to several orders of magnitude faster using LDDs. The most extreme example is model `frogs.3`, which has for BDDs $T_1 = 989.40$, $T_{48} = 1005.96$ and for LDDs $T_1 = 61.01$, $T_{48} = 9.36$. The large difference suggests that LDDs are a more efficient representation for the models of the BEEM database. Some models are missing that timed out for BDDs but did not time out for LDDs, for example model `blocks.4`. Perhaps better results for BDDs would be obtained by obtaining the smallest number of bits for

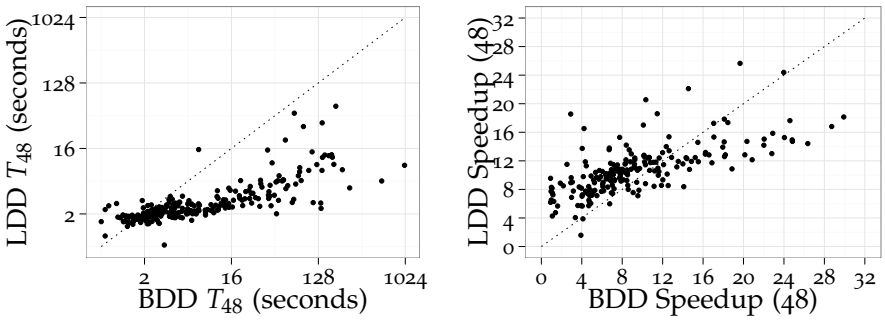


Figure 5.7 Results of the models that did not time out for both BDDs and LDDs, comparing time with 48 workers (left) and obtained (absolute) speedup (right).

each specific integer of the state vector, instead of for all integers. When we performed these experiments, LTSmin did not support this feature.

5.6 Conclusion and Discussion

The following table summarizes the experiments of Sections 5.5.2–5.5.5.

Experiment	T_1	T_{48}	T_1/T_{48}
parallel operations, sequential learning (table variant 3, newest ltsmin)	12917	2970	4.3
parallel operations, parallel learning (table variant 2, oldest ltsmin)	20745	3737	5.6
fully parallel reachability algorithm (table variant 2, oldest ltsmin)	20756	1298	16.0
fully parallel reachability algorithm (table variant 3, old ltsmin)	16231	896	18.1

The evolution of LTSmin and Sylvan is shown in the above table. The results with the "oldest" version of LTSmin were obtained for [DP15], whereas the results with the "old" version of LTSmin were obtained for [DP16b]. The results with the "new" version of LTSmin were obtained specifically for this thesis. It is clear that newer versions have better performance, although the "oldest" and "old" versions were relatively similar in terms of performance; the difference here is caused by the new design of the nodes table (variant 3).

We have seen that the best obtained speedups were obtained with the `telephony.8` model in the first experiment (21.1x), and with the `blocks.4` model in the second (29.0x), third (38.0x) and fourth (36.8x) experiments, all with 48 workers. Not all models result in good speedup, as we have seen that the speedup greatly depends on the model of which we compute the reachable state space.

The majority of the work to parallelize on-the-fly symbolic reachability was implementing parallel decision diagram operations in Sylvan. Using the framework offered by Sylvan, parallel on-the-fly symbolic reachability is then quite straightforward to implement. We use the Lace work-stealing framework that is a part of Sylvan to implement Algorithm 5.7. The `collect` method was the only custom decision diagram operation needed and only required about 25 lines of code, including some overhead to manage internal references for garbage collection.

We conclude that parallelizing on-the-fly symbolic state space exploration using Sylvan is a success. Furthermore, disjunctive partitioning of the state space contributed significantly to the obtained parallel speedup reported here.

Recent experiments in related work Sylvan has also been used as a symbolic backend in the model checker IscasMC, a probabilistic model checker [Hah+14] written in Java. A recent study [Dij+15] compared the performance of the BDD libraries CUDD, BuDDy, CacBDD, JDD, Sylvan, and BeeDeeDee when used as the symbolic backend of IscasMC and performing symbolic reachability. This result is quite important, as we did not directly compare our decision diagram library with other such libraries in LTSmin.

They summarize the overall runtimes by the following table [Dij+15]:

backend	time (s)	backend	time (s)
sylvan-7	608	buddy	2156
cacbdd	1433	jdd	2439
cudd-bdd	1522	beedeedee	2598
sylvan-1	1838	cudd-mtbdd	2837

This result was produced with variant 2 of the nodes table in Sylvan. As the results show, Sylvan is competitive with other BDD implementations when used sequentially (with 1 worker) and benefits from parallelism (with 7 workers).

More sophisticated algorithms The work in this chapter and the performed experiments are based on one of the most basic methods to perform state space exploration: a set-based breadth first search. Based on the results we obtained, we expect that the application of parallel decision diagram operations in more

sophisticated model checking algorithms and for other types of transition systems also results in parallel speedup. It is quite likely though that some more sophisticated algorithms can be challenging to parallelize in this way and may require some creativity to obtain good speedup. One particular example is the saturation algorithm proposed and advocated by Ciardo et al. [CLS01; CMS03; CZJ12]. They write on the parallelization of state space exploration with the saturation algorithm [ELC07; CZJ09], suggesting in [CZJ09, Section 3] that on shared-memory systems, calls to relational products could be parallelized. They write here that operations on decision diagrams often share computation paths and that parallelization may result in recomputation, and suggest to use the operation cache to record the *intention* of computation might alleviate this problem. In our experience, the amount of recomputation due to parallelization is small and the benefits outweigh the extra work that is done.

In [ELC07], they report parallel speedup for some models on a dual-core machine, and slowdowns in other cases. We suggest that one of the reasons may be the use of mutexes to protect the nodes tables (one table for each variable level) against race conditions. Perhaps the nodes tables that we propose in this thesis and that are fundamental to our success with breadth first search could alleviate some of the problems they encountered.

Additionally, they report that many operations are very small and therefore difficult to parallelize. This may be similar to the limited speedup that we obtained with the sequential breadth first search algorithm (only parallel decision diagram operations) and perhaps a similar strategy using disjunctive transition relations of some kind could improve the parallel performance.

It may well be possible that the techniques that allowed good scalability for LTSmin, especially the scalable lock-free nodes table and operation cache, may result in much better scalability for parallel saturation in the future.

Chapter 6

Application: Bisimulation minimisation

One of the main challenges for model checking is that the space and time requirements of model checking algorithms increase exponentially with the size of the models. One technique that helps combat this challenge is called bisimulation minimisation. Given an input model, bisimulation minimisation computes the smallest equivalent model, also called the maximal bisimulation, under some notion of equivalence. This can significantly reduce the number of states. This technique is also used to abstract models from internal behavior, when only observable behavior is relevant.

The maximal bisimulation of a model is typically computed using partition refinement. Starting with an initially coarse partition (e.g., all states are equivalent), the partition is refined until states in each equivalence class can no longer be distinguished. The result is the maximal bisimulation with respect to the initial partition. Blom et al. [BO03] introduced a signature-based method, which assigns states to equivalence classes according to a characterizing signature. This method easily extends to various types of bisimulation.

In the literature, symbolic methods have been applied to bisimulation minimisation in several ways. Bouali and De Simone [BS92] refine the equivalence relation $R \subseteq S \times S$, by iteratively removing all "bad" pairs from R, i.e., pairs of states that are no longer equivalent. For strong bisimulation, Mumme and Ciardo [MC13] apply saturation-based methods to compute R. Wimmer et al. [WHB07; Wim+06] use signatures to refine the partition, represented by the assignment to equivalence classes $P\colon S \to C$. Symbolic bisimulation based on signatures has also been applied to Markov chains by Derisavi [Der07b] and Wimmer et al. [WB10; WDH10].

The symbolic representation of the maximal bisimulation, when effective, often tends to be much larger than the original model. One particular applica-

tion of symbolic bisimulation minimisation is as a bridge between symbolical models and explicit-state analysis algorithms. Such models can have very large state spaces that are efficiently encoded using BDDs. If the minimised model is sufficiently small, then it can be analyzed efficiently using explicit-state algorithms.

Symbolic techniques mainly reduce the memory requirements of model checking. To take advantage of computer systems with multiple processors, developing scalable parallel algorithms is the way forward. Parallelization has been applied to explicit-state bisimulation minimisation. Blom et al. [Blo+08; BO03] introduced a parallel, signature-based algorithm for various types of bisimulation, especially strong and branching bisimulation. Also, [Kul13] proposed a concurrent algorithm for bisimulation minimisation which combines signatures with the approach by Paige and Tarjan [PT87]. Recently, Wijs [Wij15] implemented highly parallel strong and branching bisimilarity checking on GPGPUs. As far as we are aware, no earlier work combines symbolic bisimulation minimisation and parallelism.

This chapter studies bisimulation minimisation for labeled transition systems (LTSs), continuous-time Markov chains (CTMCs) and interactive Markov chains (IMCs), which combines the features of LTSs and CTMCs. These allow the analysis of quantitative properties, e.g., performance and dependability. We concentrate on strong bisimulation and branching bisimulation. Strong bisimulation preserves both internal behavior (τ-transitions) and observable behavior, while branching bisimulation abstracts from internal behavior. The advantage of branching bisimulation compared to other variations of weak bisimulation is that it preserves the branching structure of the LTS, thus preserving certain interesting properties such as CTL* without next-state operator [DV95].

The current chapter contains the following contributions. After preliminaries (Section 6.1), we introduce the notion of partition refinement with partial signatures in Section 6.2. Section 6.3 discusses how we extend Sylvan to parallelize signature-based partition refinement. In particular, we develop two specialized BDD algorithms. We implement a new `refine` algorithm, that refines a partition according to a signature, but maximally reuses the block number assignment of the previous partition (Section 6.3.2). This algorithm improves the operation cache use for the computation of the signatures of stable blocks, and enables partition refinement with partial signatures. We also present the `inert` algorithm, which, given a transition relation and a partition, removes all transitions that are not inert (Section 6.3.3). This algorithm avoids an expensive intermediate result reported in the literature [Wim+06]. We present the implementation of these algorithms as a versatile tool that can be customized for bisimulation minimisation in various contexts (Section 6.4). Section 6.5 discusses experimental data based on benchmarks from the literature

to demonstrate a speedup of up to 95x sequentially. In addition, we find parallel speedups of up to 17x due to parallelisation with 48 cores. Finally, we end the chapter with conclusions in Section 6.6.

This chapter is based on the following publication:
[DP16a] Tom van Dijk and Jaco van de Pol. "Multi-Core Symbolic Bisimulation Minimisation." In: *TACAS*. vol. 9636. LNCS. Springer, 2016, pp. 332–348

6.1 Definitions

We recall the basic definitions of partitions, of LTSs, of CTMCs, of IMCs, and of various bisimulations as in [BO03; HK09; WHB07; Wim+06; Wim+07].

Definition 6.1.1. Given a set S, a partition π of S is a subset $\pi \subseteq 2^S$ such that

$$\bigcup_{C \in \pi} C = S \qquad \text{and} \qquad \forall C, C' \in \pi \colon (C = C' \vee C \cap C' = \emptyset).$$

If π' and π are two partitions, then π' is a refinement of π, written $\pi' \sqsubseteq \pi$, if each block of π' is contained in a block of π. The elements of π are called equivalence classes or blocks. Each equivalence relation $\equiv$ is associated with a partition $\pi = S/\equiv$. In this chapter, we use π and $\equiv$ interchangeably.

Definition 6.1.2. A labeled transition system (LTS) is a tuple $(S, \mathsf{Act}, \rightarrow)$, consisting of a set of states S, a set of labels Act that may contain the non-observable action τ, and transitions $\rightarrow \subseteq S \times \mathsf{Act} \times S$.

We write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. and $s \xrightarrow{\tau}$ when s has no outgoing τ-transitions. We use $\xrightarrow{a*}$ to denote the transitive reflexive closure of $\xrightarrow{a}$. Given an equivalence relation $\equiv$, we write $\underset{\equiv}{\xrightarrow{a}}$ for $\xrightarrow{a} \cap \equiv$, i.e., transitions between equivalent states, called *inert* transitions. We use $\underset{\equiv}{\xrightarrow{a*}}$ for the transitive reflexive closure of $\underset{\equiv}{\xrightarrow{a}}$.

Definition 6.1.3. A continuous-time Markov chain (CTMC) is a tuple $(S, \Rightarrow)$, consisting of a set of states S and Markovian transitions $\Rightarrow \subseteq S \times \mathbb{R}^{>0} \times S$.

We write $s \overset{\lambda}{\Rightarrow} t$ for $(s, \lambda, t) \in \Rightarrow$. The interpretation of $s \overset{\lambda}{\Rightarrow} t$ is that the CTMC can switch from s to t within d time units with probability $1 - e^{-\lambda \cdot d}$. For a state s, let $\mathbf{R}(s)(s') = \sum\{\lambda \mid s \overset{\lambda}{\Rightarrow} s'\}$ be the rate to move from state s to state s', and let $\mathbf{R}(s)(C) = \sum_{s' \in C} \mathbf{R}(s)(s')$ be the cumulative rate to reach a set of states $C \subseteq S$ from state s.

Definition 6.1.4. An interactive Markov chain (IMC) is a tuple $(S, \mathsf{Act}, \rightarrow, \Rightarrow)$, consisting of a set of states S, a set of labels Act that may contain the non-observable action τ, transitions $\rightarrow \subseteq S \times \mathsf{Act} \times S$, and Markovian transitions $\Rightarrow \subseteq S \times \mathbb{R}^{>0} \times S$.

An IMC basically combines the features of an LTS and a CTMC. One feature of IMCs is the *maximal progress assumption*. Internal interactive transitions, i.e. τ-transitions, can be assumed to take place immediately, while the probability that a Markovian transition executes immediately is zero. Therefore, we may remove all Markovian transitions from states that have outgoing τ-transitions: $s \xrightarrow{\tau}$ implies $\mathbf{R}(s)(S) = 0$. We call IMCs to which this operation has been applied *maximal-progress-cut* (mp-cut) IMCs.

In this chapter, we look at strong and branching bisimulation.

For LTSs, strong and branching bisimulation are typically defined as follows [Wim+06]:

Definition 6.1.5. An equivalence relation $\equiv_S$ is a strong bisimulation on an LTS iff for all states s, t, s' with $s \equiv_S t$ and for all $s \xrightarrow{a} s'$, there is a state t' with $t \xrightarrow{a} t'$ and $s' \equiv_S t'$.

Definition 6.1.6. An equivalence relation $\equiv_B$ is a branching bisimulation on an LTS iff for all states s, t, s' with $s \equiv_B t$ and for all $s \xrightarrow{a} s'$, either

- $a = \tau$ and $s' \equiv_B t$, or
- there are states t', t'' with $t \xrightarrow{\tau*} t' \xrightarrow{a} t''$ and $t \equiv_B t'$ and $s' \equiv_B t''$.

For CTMCs, strong bisimulation is defined as follows [Der07b; WB10]:

Definition 6.1.7. An equivalence relation $\equiv_S$ is a strong bisimulation on a CTMC iff for all $(s, t) \in \equiv_S$ and for all classes $C \in S/\equiv_S$, $\mathbf{R}(s)(C) = \mathbf{R}(t)(C)$.

6

For mp-cut IMCs, strong and branching bisimulation are defined as follows [HK09; Wim+07]:

Definition 6.1.8. An equivalence relation $\equiv_S$ is a strong bisimulation on an mp-cut IMC iff for all $(s, t) \in \equiv_S$ and for all classes $C \in S/\equiv_S$

- $s \xrightarrow{a} s'$ for some $s' \in C$ implies $t \xrightarrow{a} t'$ for some $t' \in C$
- $\mathbf{R}(s)(C) = \mathbf{R}(t)(C)$

Definition 6.1.9. An equivalence relation $\equiv_B$ is a branching bisimulation on an mp-cut IMC iff for all $(s, t) \in \equiv_B$ and for all classes $C \in S/\equiv_B$

- $s \xrightarrow{a} s'$ for some $s' \in C$ implies
  - $a = \tau$ and $(s, s') \in \equiv_B$, or
  - there are states $t', t'' \in S$ with $t \xrightarrow{\tau*} t' \xrightarrow{a} t''$ and $(t, t') \in \equiv_B$ and $t'' \in C$.
- $\mathbf{R}(s)(C) > 0$ implies
  - $\mathbf{R}(s)(C) = \mathbf{R}(t')(C)$ for some $t' \in S$ such that $t \xrightarrow{\tau*} t' \xrightarrow{\tau} \not\rightarrow$ and $(t, t') \in \equiv_B$.
- $s \xrightarrow{\tau} \not\rightarrow$ implies $t \xrightarrow{\tau*} t' \xrightarrow{\tau} \not\rightarrow$ for some t'

6.2 Signature-based bisimulation minimisation

Blom and Orzan [BO03] introduced a signature-based approach to compute the maximal bisimulation of an LTS, which was further developed into a symbolic method by Wimmer et al. [Wim+06]. Each state is characterized by a *signature*, which is the same for all equivalent states in a bisimulation. These signatures are used to refine a partition of the state space until a fixed point is reached, which is the maximal bisimulation.

In the literature, multiple signatures are sometimes used that together fully characterize states, for example based on the state labels, based on the rates of continous-time transitions, and based on the enabled interactive transitions. We consider these multiple signatures as elements of a single signature that fully characterizes each state.

Definition 6.2.1. A signature $\sigma(\pi)(s)$ is a tuple of functions $f_i(\pi)(s)$, that together characterize each state s with respect to a partition π.

Two signatures $\sigma(\pi)(s)$ and $\sigma(\pi)(t)$ are equivalent, if and only if for all f_i, $f_i(\pi)(s) = f_i(\pi)(t)$.

The signatures of five bisimulations from Section 6.1 are known from the literature. For all actions $a \in \mathsf{Act}$ and equivalence classes $C \in \pi$, we define

- $\mathbf{T}(\pi)(s) = \{(a, C) \mid \exists s' \in C \colon s \xrightarrow{a} s'\}$
- $\mathbf{B}(\pi)(s) = \{(a, C) \mid \exists s' \in C \colon s \xrightarrow[\pi]{\tau*} \xrightarrow{a} s' \wedge \neg(a = \tau \wedge s \in C)\}$
- $\mathbf{R}^s(\pi)(s) = C \mapsto \mathbf{R}(s)(C)$
- $\mathbf{R}^b(\pi)(s) = C \mapsto \max(\{\mathbf{R}(s')(C) \mid \exists s' \colon s \xrightarrow[\pi]{\tau*} s' \overset{\tau}{\not\to}\})$

The five bisimulations are associated with the following signatures:

Strong bisimulation for an LTS	$(\mathbf{T})$	[Wim+06]
Branching bisimulation for an LTS	$(\mathbf{B})$	[Wim+06]
Strong bisimulation for a CTMC	$(\mathbf{R}^s)$	[WB10]
Strong bisimulation for an mp-cut IMC	$(\mathbf{T}, \mathbf{R}^s)$	[Wim+07]
Branching bisimulation for an mp-cut IMC	$(\mathbf{B}, \mathbf{R}^b, s \xrightarrow{\tau*} \overset{\tau}{\not\to})$	[Wim+07]

Functions $\mathbf{T}$ and $\mathbf{B}$ assign to each state s all actions a and equivalence classes $C \in \pi$, such that state s can reach C by an action a either directly ($\mathbf{T}$) or via any number of inert τ-steps ($\mathbf{B}$). $\mathbf{R}^s$ equals $\mathbf{R}$ but with the domain restricted to the equivalence classes $C \in \pi$, and represents the cumulative rate with which state s can go to states in C. $\mathbf{R}^b$ equals $\mathbf{R}^s$ for states $s \overset{\tau}{\not\to}$, and takes the highest "reachable rate" for states with inert τ-transitions. In branching bisimulation for mp-cut IMCs, the "highest reachable rate" is by definition the rate that all

states $s \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow$ in C have. The element $s \xrightarrow{\tau*} \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow$ distinguishes time-convergent states from time-divergent states [Wim+07], and is independent of the partition.

For the bisimulations of Definitions 6.1.5–6.1.9, we state:

Lemma 6.2.2. *A partition π is a bisimulation, if and only if for all s and t that are equivalent in π, $\sigma(\pi)(s) = \sigma(\pi)(t)$.*

For the above definitions it is fairly straightforward to prove that they are equivalent to the classical definitions of bisimulation. See e.g. [BO03; Wim+06] for the bisimulations on LTSs and [Wim+07] for the bisimulations on IMCs.

6.2.1 Partition refinement

The definition of signature-based partition refinement is as follows.

Definition 6.2.3 (Partition refinement with full signatures)**.**

$$\begin{aligned} \text{sigref}(\pi, \sigma) &:= \{\{t \in S \mid \sigma(\pi)(s) = \sigma(\pi)(t)\} \mid s \in S\} \\ \pi^0 &:= \{S\} \\ \pi^{n+1} &:= \text{sigref}(\pi^n, \sigma) \end{aligned}$$

The algorithm iteratively refines the initial coarsest partition $\{S\}$ according to the signatures of the states, until some fixed point $\pi^{n+1} = \pi^n$ is obtained. This fixed point is the maximal bisimulation for "monotone signatures":

Definition 6.2.4. A signature is monotone if for all π, π' with $\pi \sqsubseteq \pi'$, whenever $\sigma(\pi)(s) = \sigma(\pi)(t)$, also $\sigma(\pi')(s) = \sigma(\pi')(t)$.

For all monotone signatures, the sigref operator is monotone: $\pi \sqsubseteq \pi'$ implies $\text{sigref}(\pi, \sigma) \sqsubseteq \text{sigref}(\pi', \sigma)$. Hence, following Kleene's fixed point theorem, the procedure above reaches the greatest fixed point.

In Definition 6.2.3, the full signature is computed in every iteration. We propose to apply partition refinement using parts of the signature. By definition, $\sigma(\pi)(s) = \sigma(\pi)(t)$ if and only if for all parts $f_i(\pi)(s) = f_i(\pi)(t)$.

Definition 6.2.5 (Partition refinement with partial signatures)**.**

$$\begin{aligned} \text{sigref}(\pi, f_i) &:= \{\{t \in S \mid f_i(\pi)(s) = f_i(\pi)(t) \wedge s \equiv_\pi t\} \mid s \in S\} \\ \pi^0 &:= \{S\} \\ \pi^{n+1} &:= \text{sigref}(\pi^n, f_i) \qquad (\text{select } f_i \in \sigma) \end{aligned}$$

We always select some f_i that refines the partition π. A fixed point is reached only when no f_i refines the partition further: $\forall f_i \in \sigma\colon\ \text{sigref}(\pi^n, f_i) = \pi^n$. The extra clause $s \equiv_\pi t$ ensures that every application of sigref refines the partition.

Theorem 6.2.6. *If all parts f_i are monotone, Def. 6.2.5 yields the greatest fixed point.*

Proof. The procedure terminates since the chain is decreasing ($\pi^{n+1} \sqsubseteq \pi^n$), due to the added clause $s \equiv_\pi t$. We reach some fixed point π^n, since $\forall f_i \in \sigma$: $\text{sigref}(\pi^n, f_i) = \pi^n$ implies $\text{sigref}(\pi^n, \sigma) = \pi^n$. Finally, to prove that we get the *greatest* fixed point, assume there exists another fixed point $\xi = \text{sigref}(\xi, \sigma)$. Then also $\xi = \text{sigref}(\xi, f_i)$ for all i. We prove that $\xi \sqsubseteq \pi^n$ by induction on n. Initially, $\xi \sqsubseteq S = \pi^0$. Assume $\xi \sqsubseteq \pi^n$, then for the selected i, $\xi = \text{sigref}(\xi, f_i) \sqsubseteq \text{sigref}(\pi^n, f_i) = \pi^{n+1}$, using monotonicity of f_i. □

There are several advantages to this approach due to its flexibility. First, for any f_i that is independent of the partition, refinement with respect to that f_i only needs to be applied once. Furthermore, refinements can be applied according to different strategies. For instance, for the strong bisimulation of an mp-cut IMC, one could refine w.r.t. $\mathbf{T}$ until there is no more refinement, then w.r.t. $\mathbf{R}^s$ until there is no more refinement, then repeat until neither $\mathbf{T}$ nor $\mathbf{R}^s$ refines the partition. Finally, computing the full signature is the most memory-intensive operation in symbolic signature-based partition refinement. If the partial signatures are smaller than the full signature, then larger models can be minimised.

6.3 Symbolic signature refinement

This section describes the (MT)BDDs and (MT)BDD operations required for signature-based partition refinement. We describe how we encode partitions and signatures for signature-based partition refinement. We present a new parallelized `refine` function that maximally reuses block numbers from the old partition. Finally, we present a new BDD algorithm that computes inert transitions, i.e., restricts a transition relation such that states s and s' are in the same block.

6.3.1 Encoding of signature refinement

We implement symbolic signature refinement similar to [Wim+06]. However, we do not refine the partition with respect to a single block, but with respect to all blocks simultaneously. We use a binary encoding with variables s for the current state, s' for the next state, a for the action labels and b for the blocks. We order BDD variables a and b after s and s', since this is required to efficiently replace signatures (a, b) by new block numbers b (see below). Variables s and s' are interleaved, which is common in the context of transition systems.

To perform symbolic bisimulation we represent a number of sets by their characteristic functions. See also Figure 6.1.

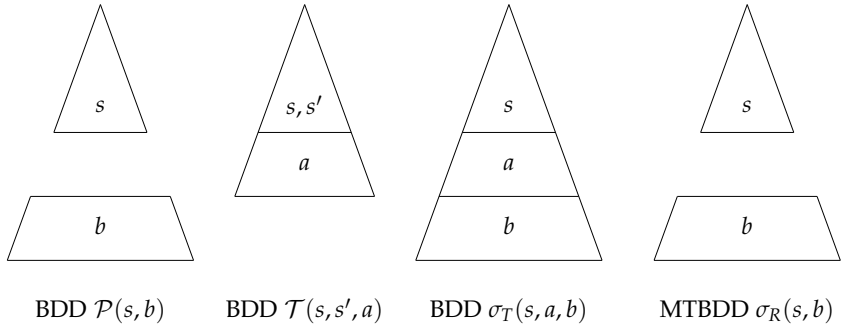


Figure 6.1 Schematic overview of the BDDs in signature refinement

- A set of states is represented by a BDD $\mathcal{S}(s)$;
- Transitions are represented by a BDD $\mathcal{T}(s,s',a)$;
- Markovian transitions are represented by an MTBDD $\mathcal{R}(s,s')$, with leaves containing rational numbers ($\mathbb{Q}$) that represent the transition rates;
- Signatures $\mathbf{T}$ and $\mathbf{B}$ are represented by a BDD $\sigma_T(s,a,b)$;
- Signatures $\mathbf{R}^s$ and $\mathbf{R}^b$ are represented by an MTBDD $\sigma_R(s,b)$, with leaves containing rational numbers ($\mathbb{Q}$) that represent the rates in the signature.

In the literature, three methods have been proposed to represent π.

1. As an equivalence relation, using a BDD $\mathcal{E}(s,s') = 1$ iff $s \equiv_\pi s'$ [BS92; MC13].
2. As a partition, by assigning each block a unique number, encoded with variables b, using a BDD $\mathcal{P}(s,b) = 1$ iff $s \in C_b$ [Der07b; Wim+06; Wim+07].
3. Using $k = \lceil \log_2 n \rceil$ BDDs $\mathcal{P}_0, \ldots, \mathcal{P}_{k-1}$ such that $\mathcal{P}_i(s) = 1$ iff $s \in C_b$ and the i^{th} bit of b is 1. This requires significant time to restore blocks for the refinement procedure, but can require less memory [Der07a].

We choose to use method 2, since in practice the BDD of $\mathcal{P}(s,b)$ is smaller than the BDD of $\mathcal{E}(s,s')$. Using $\mathcal{P}(s,b)$ also has the advantage of straightforward signature computation. The logarithmic representation is incompatible with our approach, since we refine all blocks simultaneously. Their approach involves restoring individual blocks to the $\mathcal{P}(s,b)$ representation, performing a refinement step, and compacting the result to the logarithmic representation. Restoring all blocks simply computes the full $\mathcal{P}(s,b)$.

We represent Markovian transitions using rational numbers, since they offer better precision than floating-point numbers. The manipulation of floating-point numbers typically introduces tiny rounding errors, resulting in different

results of similar computations. This significantly affects bisimulation reduction, often resulting in finer partitions than the maximal bisimulation [WB10], which is unacceptable.

6.3.2 The `refine` algorithm

In this section, we present a new BDD algorithm to refine partitions according to a signature, which maximally preserves previously assigned block numbers.

Partition refinement consists of two steps: computing the signatures and computing the next partition. Given the signatures σ_T and/or σ_R for the current partition π, the new partition can be computed as follows.

Since the chosen variable ordering has variables s, s' before a, b, each path in σ ends in a (MT)BDD representing the signature for the states encoded by that path. For σ_T, every path that assigns values to s ends in a BDD on a, b. For σ_R, every path that assigns values to s ends in a MTBDD on b with rational leaves.

Wimmer et al. [Wim+06] present a BDD operation `refine` that "replaces" these sub-(MT)BDDs by the BDD representing a unique block number for each distinct signature. The result is the BDD of the next partition. They use a global counter and a hash table to associate each signature with a unique block number. This algorithm has the disadvantage that block number assignments are unstable. There is no guarantee that a stable block has the same block number in the next iteration. This has implications for the computation of the new signatures. When the block number of a stable block changes, cached results of signature computation in earlier iterations cannot be reused.

We modify the `refine` algorithm to use the current partition to reuse the previous block number of each state. This also allows refining a partition with respect to only a part of the signature, as described in Section 6.2. The modification is applied such that it can be parallelized in Sylvan. See Algorithm 6.1.

The algorithm has two input parameters: the (MT)BDD σ which encodes the (partial) signature for the current partition, and the BDD $\mathcal{P}$ which encodes the current partition. The algorithm uses a global counter iter, which is the current iteration of partition refinement. This is necessary since the cached results of the previous iteration cannot be reused. It also uses and updates an array `blocks`, which contains the signature of each block in the new partition. This array is cleared between iterations of partition refinement.

The implementation is similar to other BDD operations, featuring the use of the operation cache (lines 2 and 15) and a recursion step for variables in s (lines 3–7), with the two recursive operations executed in parallel. `refine` simultaneously descends in σ and $\mathcal{P}$ (lines 5–6), matching the valuation of s_i in σ and $\mathcal{P}$. Block assignment happens at lines 9–14. We rely on the well-known atomic operation `compare_and_swap` (`cas`), which atomically compares and

```
1  def refine(σ, P):
2      if (σ, P, iter) ∈ cache : return cache[(σ, P, iter)]
3      v = topVar(σ, P)
4      if v equals s_i for some i :
           // match paths on s in σ and P
5          low ← refine(σ_{s_i=0}, P_{s_i=0})
6          high ← refine(σ_{s_i=1}, P_{s_i=1})
7          result ← lookupBDDnode(s_i, low, high)
8      else:
           // σ now encodes the state signature
           // P now encodes the previous block
9          B ← decodeBlock(P)
           // try to claim block B if still free
10         if blocks[B].sig = ⊥ : cas(blocks[B].sig, ⊥, σ)
11         if blocks[B].sig = σ : result ← P
12         else:
13             B ← search_or_insert(σ, B)
14             result ← encodeBlock(B)
15     cache[(σ, P, iter)] ← result
16     return result
```

Algorithm 6.1 `refine`, the (MT)BDD operation that assigns block numbers to signatures, given a signature σ and the previous partition $\mathcal{P}$.

6

modifies a value in memory. This is necessary so the algorithm is still correct when parallelized. We use `cas` to claim a block number for the signature (line 10). If the block number is already used for a different signature, then this block is being refined and we call a method `search_or_insert` to assign a new block number.

Different implementations of `search_and_insert` are possible. We implemented a parallel hash table that uses a global counter for the next block number when inserting a new pair (σ, B), similar to [Wim+06]. An alternative implementation that performed better in our experiments integrates the `blocks` array with a skip list. A skip list is a probabilistic multi-level ordered linked list. See [Pug90].

Skip list implementation Our implementation of the skip list is restricted to at most 5 levels and supports only the `insert` operation. We use a short-lived local lock at the lowest level to insert buckets, and lock-free insertions using atomic `cas` at higher levels. Furthermore, by restricting the number of blocks

```
1  def search-or-insert(sig, pb):
       // Uses global variable next_block
2      loc ← 0
3      level ← 4
4      loop:
5          cur ← buckets[loc]
6          locnext ← cur.next[level].value
7          next ← buckets[locnext]
8          if locnext ≠ 0 ∧ ⟨next.sig, next.prev_block⟩ = ⟨sig, pb⟩ :
9              return locnext
10         elif locnext ≠ 0 ∧ ⟨next.sig, next.prev_block⟩ < ⟨sig, pb⟩ :
11             loc ← locnext
12         elif level > 0 :
13             trace[level] ← loc
14             level ← level − 1
15         elif not cur.next[0].locked :
16             if cas(cur.next[0], locnext, locked+locnext) : break
17     locnew ← fetch_and_add(next_block, 1)
18     new ← buckets[locnew]
19     new.sig ← sig
20     new.prev_block ← pb
21     new.next[0] ← locnext
22     cur.next[0] ← locnew
23     level ← 1
24     h ← some random height (geometric distribution) from 1 to 5
25     while level < h :
26         loc ← trace[level]
27         loop:
28             cur ← buckets[loc]
29             locnext ← cur.next[level].value
30             next ← buckets[locnext]
31             if locnext ≠ 0 ∧ ⟨next.sig, next.prev_block⟩ < ⟨sig, pb⟩ :
32                 loc ← locnext
33             else:
34                 new.next[level] ← locnext
35                 if cas(cur.next[level], locnext, locnew) : break
36         level ← level + 1
37     return locnew
```

Algorithm 6.2 The `search-or-insert` algorithm with a skiplist.

to at most 2^{31}, we only need 32 bytes for each bucket in the skip list:

```
struct { uint64_t sig; uint32_t prev_block; uint32_t next[5]; }
```

Each bucket in the skip list contains the pair (σ, B) (`sig` and `prev_block`) and the 31-bit indices of the next bucket at each level. The highest bit of `next[0]` is used as a lock, which is released when setting `next[0]` to a new value.

We implement `search_or_insert` as follows. We traverse the skip list until either a bucket with (σ, B) is found (and returned), or the bucket B' that would immediately precede a bucket with (σ, B). We use `cas` to set the highest bit of `next[0]` and lock bucket B'. This ensures that no other thread can insert (σ, B) (or any other pair directly preceded by bucket B') simultaneously. A new block number B'' is generated using a global counter `next_block`, which is increased atomically with `cas`. Bucket B'' is initialized and inserted into the skip list by updating `next[0]` of B' with B'', which also releases the lock on B'. Finally, the new bucket is inserted at a random number of higher levels using `cas`.

See Algorithm 6.2 for this algorithm in more detail. The first loop (lines 4–16) traverses the skip list and performs the atomic `cas` to lock the `next` pointer where we insert the new bucket. The new bucket is then acquired and initialized (lines 17–21) and the insertion is completed at line 22. At that point, the new bucket is done, but we still want to insert the bucket in higher levels, depending on a randomly generated "height" (line 24). For every level where we want to insert the bucket, the loop (lines 27–35) traverses the level (it is possible other buckets have been added) and use atomic `cas` (line 35) to insert our bucket.

6.3.3 Computing inert transitions

To compute the set of inert τ-transitions for branching bisimulation, i.e., $s \xrightarrow{\tau}_{\pi} s'$, or more generally, to compute any inert transition relation $\rightarrow \cap \equiv$ where $\equiv$ is the equivalence relation corresponding to π computed by $\mathcal{E}(s,s') = \exists b\colon \mathcal{P}(s,b) \wedge \mathcal{P}(s',b)$, the expression $\mathcal{T}(s,s') \wedge \exists b\colon \mathcal{P}(s,b) \wedge \mathcal{P}(s',b)$ must be computed. [Wim+06] writes that the intermediate BDD of $\exists b\colon \mathcal{P}(s,b) \wedge \mathcal{P}(s',b)$, obtained by first computing $\mathcal{P}(s',b)$ using variable renaming from $\mathcal{P}(s,b)$ and then $\exists b\colon \mathcal{P}(s,b) \wedge \mathcal{P}(s',b)$ using `and_exists`, is very large. This makes sense, since this intermediate result is indeed the BDD $\mathcal{E}(s,s')$, which we were avoiding by representing the partition using $\mathcal{P}(s,b)$.

The solution in [Wim+06] was to avoid computing $\mathcal{E}$ by computing the signatures and the refinement only with respect to one block at a time, which also enables several optimizations in [WHB07].

We present an alternative solution, which computes $\rightarrow \cap \equiv$ directly using a custom BDD algorithm. The `inert` algorithm takes parameters $\mathcal{T}(s,s')$ ($\mathcal{T}$ may contain other variables ordered after s,s') and two copies of $\mathcal{P}(s,b)$: $\mathcal{P}^s$

```
1  def inert(T, P^s, P^s'):
2      if (T, P^s, P^s') ∈ cache : return cache[(T, P^s, P^s')]
       // find highest variable, interpreting s_i in P^s' as s'_i
3      v = topVar(T, P^s, P^s')
4      if v equals s_i for some i :
           // match s_i in T with P^s
5          low ← inert(T_{s_i=0}, P^s_{s_i=0}, P^s')
6          high ← inert(T_{s_i=1}, P^s_{s_i=1}, P^s')
7          result ← lookupBDDnode(s_i, low, high)
8      elif v equals s'_i for some i :
           // match s'_i in T with s_i in P^s'
9          low ← inert(T_{s'_i=0}, P^s, P^{s'}_{s_i=0})
10         high ← inert(T_{s'_i=1}, P^s, P^{s'}_{s_i=1})
11         result ← lookupBDDnode(s'_i, low, high)
12     else:
           // match the blocks P^s and P^s'
13         if P^s ≠ P^s' : result ← False
14         else: result ← T
15     cache[(T, P^s, P^s')] ← result
16     return result
```

Algorithm 6.3 Computes the inert transitions of a transition relation $\mathcal{T}$ according to the block assignments to current states ($\mathcal{P}^s$) and next states ($\mathcal{P}^{s'}$).

and $\mathcal{P}^{s'}$. The algorithm matches $\mathcal{T}$ and $\mathcal{P}^s$ on valuations of variables s, and $\mathcal{T}$ and $\mathcal{P}^{s'}$ on valuations of variables s'. See Algorithm 6.3, and also Figure 6.2 for a schematic overview. When in the recursive call all valuations to s and s' have been matched, with $S_s, S_{s'} \subseteq S$ the sets of states represented by these valuations, then $\mathcal{T}$ is the set of actions that label the transitions between states in S_s and $S_{s'}$, $\mathcal{P}^s$ is the block that contains all S_s and $\mathcal{P}^{s'}$ is the block that contains all $S_{s'}$. Then if $\mathcal{P}^s \neq \mathcal{P}^{s'}$, the transitions are not inert and `inert` returns `False`, removing the transition from $\mathcal{T}$. Otherwise, $\mathcal{T}$ (which may still contain other variables ordered after s, s', such as action labels), is returned.

6.4 Implementation

We implemented multi-core signature-based partition refinement in a tool called SigrefMC. The tool supports LTSs, CTMCs and IMCs delivered in two input formats, the XML format used by the original Sigref tool, and the BDD format

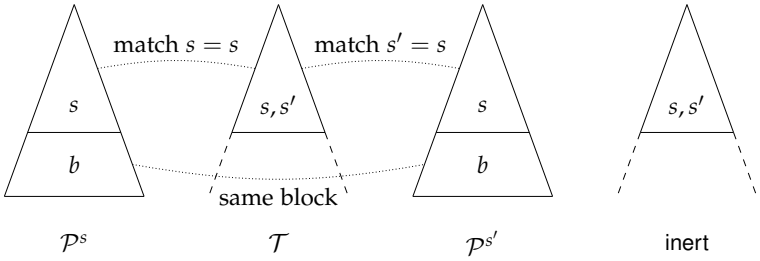


Figure 6.2 Schematic overview of the BDDs in the `inert` algorithm

that the tool LTSmin [Kan+15] generates for various model checking languages. SigrefMC supports both the floating-point and the rational representation of rates in continuous-time transitions.

One of the design goals of this tool is to encourage researchers to extend it for their own file formats and notions of bisimulation, and to integrate it in other toolsets. Therefore, SigrefMC is freely available online and licensed with the MIT license. Documentation is available and instructions for extending the tool for different input/output formats and types of bisimulation are included.

6.5 Experimental evaluation

6.5.1 Experiments

To study the improvements presented in the current chapter, we compared our results (using the skip list variant of `refine`) to Sigref 1.5 [WHB07] for LTS and IMC models, and to a version of Sigref used in [WB10] for CTMC models. For the CTMC models, we used Sigref with rational numbers provided by the GMP library and SigrefMC with rational number support by Sylvan. For the IMC models, version 1.5 of Sigref does not support the GMP library and the version used in [WB10] does not support IMCs. We used SigrefMC with floating points for a fairer comparison, but the tools give a slightly different number of blocks.

For the experiments, we restrict ourselves to the models presented in [WB10; Wim+06] and an IMC model that is part of the distribution of Sigref. These models have been generated from PRISM benchmarks using a custom version of the PRISM toolset [KNP11]. We refer to the literature for a description of these models.

We perform experiments on the three tools using the same 48-core machine, containing 4 AMD Opteron™ 6168 processors with 12 cores each. We measure

the runtimes for partition refinement using SIGREF, SIGREFMC with only 1 worker, and SIGREFMC with 48 workers.

Note that apart from the new `refine` and `inert` algorithms presented in the current chapter, there are several other differences. The first is that the original SIGREF uses the CUDD implementation of BDDs, while SIGREFMC obviously uses Sylvan, along with some extra BDD algorithms that avoid explicitly computing variable renaming of some BDDs. The second is that SIGREF has several optimizations [WHB07] that are not available in SIGREFMC.

6.5.2 Results

See Table 6.1 for the results of these experiments. These results were obtained by repeating each benchmark at least 15 times and taking the average. The timeout was set to 3600 seconds. The column "States" shows the number of states before bisimulation minimisation, and "Blocks" the number of equivalence classes after bisimulation minimisation. We show the wallclock time using SIGREF (T_w), using SIGREFMC with 1 worker (T_1) and using SIGREFMC with 48 workers (T_{48}). We compute the sequential speedup T_w/T_1, the parallel speedup T_1/T_{48} and the total speedup T_w/T_{48}.

We restrict ourselves to larger models in the presentation of the results here. In the full set of results, excluding executions that take less than 1 second, SIGREFMC is always faster sequentially and always benefits from parallelism.

The results show a clear advantage for larger models. One interesting result is for the p2p-7-5 model. This model is ideal for symbolic bisimulation with a large number of states (2^{35}) and very few blocks after minimisation (336). For this model, our tool is 95x faster sequentially and has a parallel speedup of 8x, resulting in a total speedup of 767x. The best parallel speedup of 17x was obtained for the kanban05 model.

In almost all experiments, the signature computation dominates the execution time in the sequential case with 70%–99%. We observe that the refinement step sometimes benefits more from parallelism than signature computation, with speedups up to 29.9x. We also find that reusing block numbers for stable blocks causes a major reduction in computation time towards the end of the procedure. The kanban LTS models and the larger polling CTMC models are an excellent case study to demonstrate this. See Figure 6.3.

6.6 Conclusion and Discussion

Originally we intended to investigate parallelism in symbolic bisimulation minimisation. To our surprise, we obtained a much higher sequential speedup using specialized BDD operations, as demonstrated by the results in Table 6.1

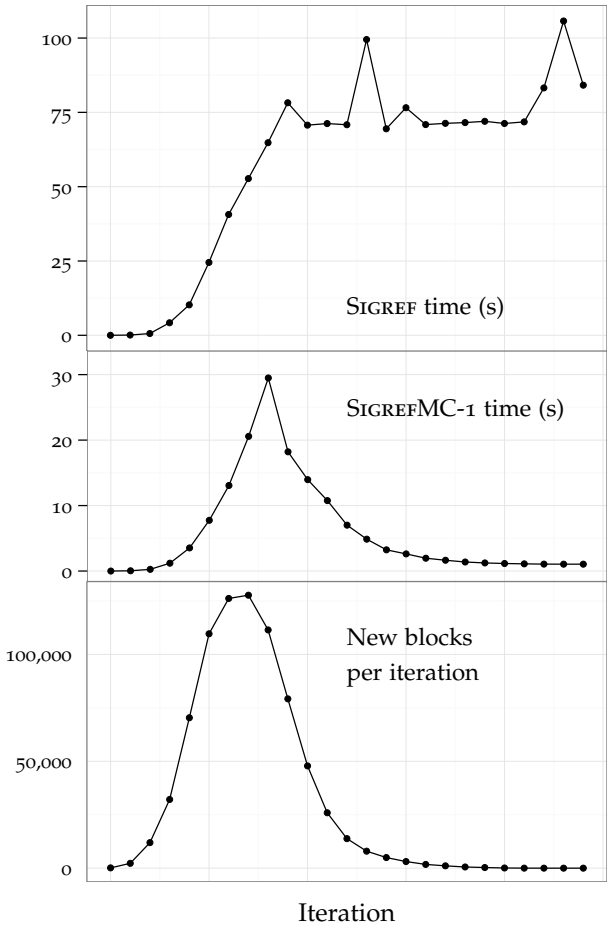


Figure 6.3 Time per iteration for SIGREF and SIGREFMC (1 worker), and the number of new blocks per iteration for strong bisimulation of the kanban04 LTS model.

LTS models (strong)			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
kanban03	1024240	85356	92.16	10.09	0.88	9.14	11.52	105.29
kanban04	16020316	778485	1410.66	148.15	11.37	9.52	13.03	124.06
kanban05	16772032	5033631	–	1284.86	73.57	–	17.47	–
kanban06	264515056	25293849	–	–	2584.23	–	–	–

LTS models (branching)			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
kanban04	16020316	2785	8.47	0.52	0.24	16.39	2.11	34.60
kanban05	16772032	7366	34.11	1.48	0.43	22.98	3.47	79.81
kanban06	264515056	17010	118.19	3.87	0.83	30.55	4.65	142.20
kanban07	268430272	35456	387.16	8.83	1.66	43.86	5.31	232.71
kanban08	4224876912	68217	1091.67	17.91	2.98	60.96	6.02	366.72
kanban09	4293193072	123070	3186.48	34.23	5.51	93.10	6.21	578.59

CTMC models			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
cycling-4	431101	282943	220.23	26.72	2.60	8.24	10.29	84.84
cycling-5	2326666	1424914	1249.23	170.28	19.42	7.34	8.77	64.34
fgf	80616	38639	71.62	8.86	0.88	8.08	10.04	81.20
p2p-5-6	2^{30}	336	750.29	26.96	2.99	27.83	9.03	251.24
p2p-6-5	2^{30}	266	248.17	9.49	1.21	26.15	7.82	204.47
p2p-7-5	2^{35}	336	2280.76	24.01	2.97	94.99	8.08	767.12
polling-16	1572864	98304	792.82	118.50	10.18	6.69	11.64	77.85
polling-17	3342336	196608	1739.01	303.65	22.58	5.73	13.45	77.03
polling-18	7077888	393216	–	705.22	49.81	–	14.16	–
robot-020	31160	30780	28.15	3.21	0.60	8.78	5.36	47.04
robot-025	61200	60600	78.48	6.78	0.95	11.58	7.11	82.39
robot-030	106140	105270	174.30	12.26	1.47	14.21	8.33	118.44

IMC models (strong)			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
ftwc01	2048	1133	1.26	1.14	0.2	1.11	5.76	6.38
ftwc02	32768	16797	154.55	102.07	15.85	1.51	6.44	9.75

IMC models (branching)			Time			Speedups		
Model	States	Blocks	T_w	T_1	T_{48}	Seq.	Par.	Total
ftwc01	2048	430	1.12	0.77	0.13	1.45	6.07	8.83
ftwc02	32786	3886	152.9	50.39	4.89	3.03	10.3	31.26

Table 6.1 Results for the benchmark experiments. Each data point is an average of at least 15 runs. The timeout was 3600 seconds.

and Figure 6.3. The specialized BDD operations offer a clear advantage sequentially and the integration with Sylvan results in decent parallel speedups. Our best result had a total speedup of 767x. Similar to our experiments in symbolic reachability [DP15], further parallel speedups might be obtained by disjunctively partitioning the transition relations.

Chapter 7

Conclusions

We studied the multi-core implementation of decision diagram operations, using work-stealing and scalable data structures, for the applications symbolic model checking and symbolic bisimulation minimisation.

7.1 The multi-core decision diagram package Sylvan

The main contribution of this thesis is the reusable multi-core decision diagram library Sylvan. Sylvan implements parallelized operations on various types of decision diagrams.

One of its particular strengths is that it can replace existing non-parallel decision diagram libraries to bring the processing power of multi-core machines to non-parallel applications. We demonstrated this for state space exploration in LTSmin, where we obtained a reasonable parallel speedup of up to 21x with 48 cores (Experiment 1, Section 5.5.2) when we did not modify LTSmin, and 29x with 48 cores (Experiment 2, Section 5.5.3) when we slightly modified LTSmin to make transition learning thread-safe. Also, symbolic bisimulation minimisation is a sequential algorithm that we parallelized using Sylvan. We did not introduce additional parallelism to this application and obtained the reasonable parallel speedup of up to 17x with 48 cores (Section 6.5, Table 6.1).

As discussed in Section 5.6, Sylvan has also been used as a symbolic backend in the probabilistic model checker IscasMC [Hah+14]. A recent study [Dij+15] compared the performance of the BDD libraries CUDD, BuDDy, CacBDD, JDD, Sylvan, and BeeDeeDee when used as the symbolic backend of IscasMC and performing symbolic reachability. The results show that Sylvan is competitive with other BDD implementations when used sequentially (with 1 worker) and significantly faster when using multiple cores (with 7 workers).

Sylvan supports binary decision diagrams, list decision diagrams and multi-terminal binary decision diagrams with various leaf types. It has been designed

with customization in mind, so there is extensive support for adding different leaf types and custom decision diagram operations. This is demonstrated by the application of bisimulation minimisation, as it is very beneficial to develop specialised BDD operations there. Compared to the state of the art tool Sigref [Wim+06] that relies on a version of CUDD [Som15], we obtained a sequential speedup of up to 95x, mainly due to the specialised BDD operations. We also implemented specialised algorithms for state space exploration, such as the algorithms `relnext` and `relprev`, the operations on list decision diagrams, and the `collect` operation for parallel transition learning.

Using the framework offered by Sylvan and Lace to further parallelize applications that use the parallelized decision diagram operations is also fairly straightforward, as is demonstrated in Chapter 5 for LTSmin. Parallelizing LTSmin is only a few lines of code, since the difficult task of performing load-balancing is solved by the work-stealing in Lace. Experimentally, we demonstrated a speedup of up to 38x for fully parallel on-the-fly symbolic reachability in LTSmin.

Sylvan is freely available online[1] and licensed with the Apache 2.0 license.

7.2 The work-stealing framework Lace

At the heart of the parallel implementation of decision diagram operations lies the work-stealing framework Lace, which we developed for Sylvan. We implemented this framework as a research vehicle and for features that are particularly useful for parallel decision diagrams, such as a feature where all workers cooperatively suspend their current tasks and start a new task tree. This is used to implement stop-the-world garbage collection in Sylvan. Lace uses the non-blocking split deque for work-stealing that we introduced in Chapter 3 and that shows good results on a number of typical benchmarks.

Lace is freely available online[2] and licensed with the Apache 2.0 license.

7.3 The symbolic bisimulation minimisation tool SigrefMC

We implemented multi-core signature-based partition refinement in a tool called SigrefMC. The tool supports LTSs, CTMCs and IMCs delivered in two input formats, the XML format used by the original Sigref tool, and the BDD format that LTSmin generates for various model checking languages. SigrefMC supports both the floating-point and the rational representation of rates in continuous-time transitions.

[1] `https://github.com/utwente-fmt/sylvan`
[2] `https://github.com/utwente-fmt/lace`

For this application, we developed two specialised BDD algorithms. The algorithm `refine` refines a partition based on the signatures of the states, and reuses the block numbers assigned in the previous partition. The algorithm `inert` computes the inert transition relation in one step.

One of the design goals of SigrefMC is to encourage researchers to extend it for their own file formats and notions of bisimulation, and to integrate it in other toolsets. SigrefMC is freely available online[3] and licensed with the Apache 2.0 license. Documentation is available and instructions for extending the tool for different input/output formats and types of bisimulation are included.

7.4 Future directions

In this section we summarize possible future research directions that we discussed in earlier chapters and some ideas that did not fit in any particular chapter.

7.4.1 Scalable data structures

In Chapter 4, we implemented several different variants of the hash table used to store decision diagram nodes, and we presented the data structure of the operation cache. Our focus was on keeping these data structures simple. Their performance was evaluated in Chapter 5 using LTSmin.

There are many options to further improve or study these hash tables. We discussed several ideas in Chapter 4. Different variations on the operation cache are possible, for example versions that wait in `cache-get` when another thread has locked the bucket, or versions that look at multiple buckets. It may be interesting to consider more intelligent garbage collection, which keeps results in the cache if the decision diagrams are kept during garbage collection. We tried to keep the data structures as simple as possible, but more intelligent approaches may improve the performance and scalability in the future.

7.4.2 Other decision diagrams and operations

BDD minimization Decision diagram operations that we did not discuss in this thesis but that are interesting for model checking and related fields include BDD minimization algorithms such as `restrict` and `compose`, which we implemented but did not study in great detail. A possibility is to use `restrict` and `compose` or the leaf-identifying compaction algorithm [Hon+97] after exploring the state space with LTSmin to produce a minimal BDD. This minimal BDD could then be used as input for symbolic bisimulation minimisation.

[3] `https://github.com/utwente-fmt/sigrefmc`

Dynamic variable reordering Another interesting algorithm is dynamic variable reordering using sifting [Rud93], which we did not implement in Sylvan. There are various applications for which dynamic reordering is interesting, for example when there are no known heuristics for a good static reordering. Considering the typical cost of performing dynamic reordering, we think it should be a priority to research heuristics for good static reordering whenever reordering is desired. The only true case where dynamic reordering is inherently superior is when the optimal reordering changes during the computation and the performance difference outweighs the cost of performing dynamic reordering, or if the structure of the input problem is not known or understood.

Other types of decision diagrams We could also consider other types of decision diagrams. For example, zero-suppressed binary decision diagrams [Min93] are also relevant for current research [Min13] and relatively easy to implement (they are very similar to normal BDDs in their implementation) and show good performance for model checking in LTSmin [Haj14]. Other types of decision diagrams that are used in current research include hierarchical decision diagrams [CT05] and hierarchical set decision diagrams [HTK08], which are used in model checking petri nets.

Operation nodes Another idea is to use special operation nodes in the decision diagrams that signify operations in progress, similar to [HDB96]. Decision diagram operations could perform an operation by creating an operation node, and multiple workers could work on "pushing down" the nodes in the decision diagrams in order to compute the results of the operations. This might also enable other optimizations when certain operations could be rewritten on-the-fly, for example combinations of variable substitution, $\wedge$, $\neg$, $\exists$ and $\forall$.

Arithmetic intensity of the operations Currently each suboperation in Sylvan is one task. Maybe fewer tasks that are larger might improve the performance of the binary decision diagram operations. However, Lace is already quite efficient for fine-grained operations. In addition, the cache granularity described in Chapter 4 has a similar effect. It may be difficult to obtain further improvements this way.

7.4.3 Applications

Parallel saturation Chapter 5 concentrated on the parallelization of a standard breadth first search algorithm to explore the state space. The toolset LTSmin also implements a "chaining" strategy (fire the transition groups after each other, and already include new states from one transition group as input for the

next group), and Ciardo et al. have proposed and advocated an optimal iteration strategy called saturation [CLS01; CMS03; CZJ12], which is also implemented in LTSmin [Sia12]. Ciardo and Ezekiel have also written on the parallization of the saturation algorithm [ELC07; CZJ09], especially [CZJ09] has the explicit title that parallel symbolic state space exploration is difficult.

Hardware model checking Another promising future direction may be hardware model checking. In hardware models the composed subsystems are often not asynchronous as in various protocols and in software model checking, but are bound to a common clock. Transition relations are not disjunctively defined but conjunctively. For example, for and-inverter graphs where the state of the model is defined as the contents of gates, we could define a transition relation for each gate, and then combine all these transition relations to form implicitly the entire transition system. Rather than parallelizing on the set of disjunctive transition relations, we could parallelize on the set of conjunctive transition relations, which has additional challenges such as the order in which transitions are fired and subresults are combined.

7.4.4 Formal verification of the algorithms

In this thesis, we have presented a number of algorithms on data structures, namely several different versions of the unique table, the operation cache, various algorithms on decision diagrams, in particular two specialised algorithms for bisimulation minimisation, and a novel non-blocking split deque for work-stealing.

It is well known that non-blocking algorithms, especially wait-free algorithms, are prone to bugs and unexpected corner cases. Although we provide an informal proof for most of these data structures, a formal proof, for example using an automated verification tool or computer aided verification, would be insightful and useful. Especially the work-stealing deque would be an interesting case study for verification tools, due to its complexity.

7.5 Multi-core and beyond

Finally, we believe that research into parallel processing will only be more relevant in the future. The number of cores on multi-core systems increases, which enables studying larger problems than is now feasible. While it is certainly true that adding cores only improves the computation time linearly to the number of cores, it seems for the moment that parallel processing is here to stay. For some applications, graphics processors (many-core systems) have been a great success, improving the performance of specific algorithms by several

orders of magnitude. Networks of workstations can combine the power of many multi-core systems to obtain speedup, although relatively slow connections between these workstations form a considerable challenge for memory intensive computations like operations on decision diagrams.

New algorithms may be discovered for existing problems that have better time and space complexity, but these new algorithms can often also be parallelized. This thesis shows that at least for algorithms that use decision diagrams, there is the potential to greatly reduce the computation time. Better algorithms with more favorable properties are merely a new challenge for parallelization. There will always be computations that are so large that parallelizing them reduces their computation time from weeks to days, or from hours to seconds.

Bibliography

[ABP01] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. "Thread Scheduling for Multiprogrammed Multiprocessors." In: *Theory Comput. Syst.* 34.2 (2001), pp. 115–144.

[ACM96] Prakash Arunachalam, Craig M. Chase, and Dinos Moundanos. "Distributed Binary Decision Diagrams for Verification of Large Circuit." In: *ICCD*. 1996, pp. 365–370.

[ACR13] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. "Scheduling parallel programs by work stealing with private deques." In: *PPOPP*. ACM, 2013, pp. 219–228.

[Ake78] S.B. Akers. "Binary Decision Diagrams." In: *IEEE Trans. Computers* C-27.6 (June 1978), pp. 509–516.

[AR86] Magdy S. Abadir and Hassan K. Reghbati. "Functional Test Generation for Digital Circuits Described Using Binary Decision Diagrams." In: *IEEE Trans. Computers* 35.4 (1986), pp. 375–379.

[BAA95] Debashis Bhattacharya, Prathima Agrawal, and Vishwani D. Agrawal. "Test Generation for Path Delay Faults Using Binary Decision Diagrams." In: *IEEE Trans. Computers* 44.3 (1995), pp. 434–447.

[Bah+93] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. "Algebraic decision diagrams and their applications." In: *ICCAD 1993*. 1993, pp. 188–191.

[BCT07] Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. "Symbolic Fault Tree Analysis for Reactive Systems." In: *ATVA 2007*. Vol. 4762. LNCS. Springer, 2007, pp. 162–176.

[Bia+97] F. Bianchi, Fulvio Corno, Maurizio Rebaudengo, Matteo Sonza Reorda, and Roberto Ansaloni. "Boolean Function Manipulation on a Parallel System Using BDDs." In: *HPCN Europe*. 1997, pp. 916–928.

[Blo+08] Stefan Blom, Boudewijn R. Haverkort, Matthias Kuntz, and Jaco van de Pol. "Distributed Markovian Bisimulation Reduction aimed at CSL Model Checking." In: *ENTCS* 220.2 (2008), pp. 35–50.

[Blu+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System." In: *J. Parallel Distrib. Comput.* 37.1 (1996), pp. 55–69.

[Blu94] Robert D. Blumofe. "Scheduling Multithreaded Computations by Work Stealing." In: *FOCS*. IEEE Computer Society, 1994, pp. 356–368.

[BO03] Stefan Blom and Simona Orzan. "Distributed Branching Bisimulation Reduction of State Spaces." In: *ENTCS* 89.1 (2003), pp. 99–113.

[BP08] Stefan Blom and Jaco van de Pol. "Symbolic Reachability for Process Algebras with Recursive Data Types." In: *ICTAC*. Vol. 5160. LNCS. Springer, 2008, pp. 81–95.

[BPW10] Stefan Blom, Jaco van de Pol, and Michael Weber. "LTSmin: Distributed and Symbolic Reachability." In: *CAV*. Vol. 6174. LNCS. Springer, 2010, pp. 354–359.

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. "Efficient Implementation of a BDD Package." In: *DAC*. 1990, pp. 40–45.

[Bry86] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." In: *IEEE Trans. Computers* C-35.8 (Aug. 1986), pp. 677–691.

[BS92] Amar Bouali and Robert de Simone. "Symbolic Bisimulation Minimisation." In: *Computer Aided Verification, 4th Int. Workshop*. Vol. 663. LNCS. Springer, 1992, pp. 96–108.

[Bur+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. "Symbolic Model Checking: 10^20 States and Beyond." In: *Inf. Comput.* 98.2 (1992), pp. 142–170.

[Bur+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. "Symbolic model checking for sequential circuit verification." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.4 (Apr. 1994), pp. 401–424.

[CB99] Jer-Sheng Chen and P. Banerjee. "Parallel construction algorithms for BDDs." In: *ISCAS 1999*. IEEE, 1999, pp. 318–322.

[CC04] Ming-Ying Chung and Gianfranco Ciardo. "Saturation NOW." In: *QEST*. IEEE Computer Society, 2004, pp. 272–281.

[CGS92] G.P. Cabodi, S. Gai, and M. Sonza Reorda. "Boolean Function Manipulation on Massively Parallel Computers." In: *Proc. of 4th Symp. on Frontiers of Massively Parallel Computation*. IEEE, Oct. 1992, pp. 508–509.

[CL05] David Chase and Yossi Lev. "Dynamic circular work-stealing deque." In: *SPAA*. ACM, 2005, pp. 21–28.

[Cla+93] Edmund M. Clarke, Kenneth L. McMillan, Xudong Zhao, Masahiro Fujita, and J. Yang. "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping." In: *DAC*. 1993, pp. 54–60.

[CLS01] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. "Saturation: An Efficient Iteration Strategy for Symbolic State-Space Generation." In: *TACAS*. Vol. 2031. LNCS. 2001, pp. 328–342.

[CM90] Olivier Coudert and Jean Christophe Madre. "A Unified Framework for the Formal Verification of Sequential Circuits." In: *ICCAD 1990*. IEEE Computer Society, 1990, pp. 126–129.

[CMS03] Gianfranco Ciardo, Robert M. Marmorstein, and Radu Siminiceanu. "Saturation Unbound." In: *TACAS 2003*. 2003, pp. 379–393.

[CT05] Jean-Michel Couvreur and Yann Thierry-Mieg. "Hierarchical Decision Diagrams to Exploit Model Structure." In: *FORTE*. Ed. by Farn Wang. Vol. 3731. Lecture Notes in Computer Science. Springer, 2005, pp. 443–457.

[CZJ09] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. "Parallel symbolic state-space exploration is difficult, but what is the alternative?" In: *PDMC*. 2009, pp. 1–17.

[CZJ12] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. "Ten Years of Saturation: A Petri Net Perspective." In: *T. Petri Nets and Other Models of Concurrency*. Lecture Notes in Computer Science 5 (2012). Ed. by Kurt Jensen, Susanna Donatelli, and Jetty Kleijn, pp. 51–95.

[Der07a] Salem Derisavi. "A Symbolic Algorithm for Optimal Markov Chain Lumping." In: *TACAS 2007*. Vol. 4424. LNCS. 2007, pp. 139–154.

[Der07b] Salem Derisavi. "Signature-based Symbolic Algorithm for Optimal Markov Chain Lumping." In: *QEST 2007*. IEEE Computer Society, 2007, pp. 141–150.

[Dij+15] Tom van Dijk, Ernst Moritz Hahn, David N. Jansen, Yong Li, Thomas Neele, Mariëlle Stoelinga, Andrea Turrini, and Lijun Zhang. "A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking." In: *SETTA*. Vol. 9409. LNCS. Springer, 2015, pp. 35–51.

[Din+09] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. "Scalable work stealing." In: *SC*. ACM, 2009.

[DLP12] Tom van Dijk, Alfons W. Laarman, and Jaco van de Pol. "Multi-core and/or Symbolic Model Checking." In: *ECEASST* 53 (2012).

[DLP13] Tom van Dijk, Alfons Laarman, and Jaco van de Pol. "Multi-Core BDD Operations for Symbolic Reachability." In: *ENTCS* 296 (2013), pp. 127–143.

[DP14] Tom van Dijk and Jaco van de Pol. "Lace: Non-blocking Split Deque for Work-Stealing." In: *MuCoCoS*. Vol. 8806. LNCS. Springer, 2014, pp. 206–217.

[DP15] Tom van Dijk and Jaco van de Pol. "Sylvan: Multi-Core Decision Diagrams." In: *TACAS*. Vol. 9035. LNCS. Springer, 2015, pp. 677–691.

[DP16a] Tom van Dijk and Jaco van de Pol. "Multi-Core Symbolic Bisimulation Minimisation." In: *TACAS*. Vol. 9636. LNCS. Springer, 2016, pp. 332–348.

[DP16b] Tom van Dijk and Jaco van de Pol. "Sylvan: Multi-core Framework for Decision Diagrams." In: *STTT* (2016). Accepted.

[DV95] Rocco De Nicola and Frits W. Vaandrager. "Three Logics for Branching Bisimulation." In: *J. ACM* 42.2 (1995), pp. 458–487.

[ELC07] Jonathan Ezekiel, Gerald Lüttgen, and Gianfranco Ciardo. "Parallelising Symbolic State-Space Generators." In: *CAV*. Vol. 4590. LNCS. 2007, pp. 268–280.

[Fax08] Karl-Filip Faxén. "Wool-A work stealing library." In: *SIGARCH Computer Architecture News* 36.5 (2008), pp. 93–100.

[Fax10] Karl-Filip Faxén. "Efficient Work Stealing for Fine Grained Parallelism." In: *ICPP 2010*. IEEE Computer Society, 2010, pp. 313–322.

[Fis+05] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. "Verification and change-impact analysis of access-control policies." In: *ICSE 2005*. ACM, 2005, pp. 196–205.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The Implementation of the Cilk-5 Multithreaded Language." In: *PLDI*. ACM, 1998, pp. 212–223.

[GGH05a] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. "Lock-free dynamic hash tables with open addressing." In: *Distributed Computing* 18.1 (2005), pp. 21–42.

[GGH05b] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. "Lock-Free Parallel Garbage Collection." In: *ISPA*. Vol. 3758. Lecture Notes in Computer Science. Springer, 2005, pp. 263–274.

[GGH07] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. "Lock-free parallel and concurrent garbage collection by mark&sweep." In: *Sci. Comput. Program.* 64.3 (2007), pp. 341–374.

[GHS06] Orna Grumberg, Tamir Heyman, and Assaf Schuster. "A work-efficient distributed algorithm for reachability analysis." In: *Formal Methods in System Design* 29.2 (2006), pp. 157–175.

[GRS95] S. Gai, M. Rebaudengo, and M. Sonza Reorda. "An improved data parallel algorithm for Boolean function manipulation using BDDs." In: *Proc. Euromicro Workshop on Par. and Distrib. Processing*. IEEE, Jan. 1995, pp. 33–39.

[Hah+14] Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. "iscasMc: A Web-Based Probabilistic Model Checker." In: *FM*. Vol. 8442. LNCS. Springer, 2014, pp. 312–317.

[Haj14] Maryam Haji Ghasemi. "Symbolic model checking using Zero-suppressed Decision Diagrams." MA thesis. University of Twente, Dept. of C.S., Nov. 2014.

[HDB96] A. Hett, R. Drechsler, and B. Becker. "MORE: an alternative implementation of BDD packages by multi-operand synthesis." In: *Design Automation Conference, EURO-DAC '96, Geneva*. IEEE, Sept. 1996, pp. 164–169.

[Hen+06] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. "A dynamic-sized nonblocking work stealing deque." In: *Distributed Computing* 18.3 (2006), pp. 189–207.

[Hey+00] Tamir Heyman, Danny Geist, Orna Grumberg, and Assaf Schuster. "Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits." In: *Computer Aided Verification*. Vol. 1855. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2000, pp. 20–35.

[HK09] Holger Hermanns and Joost-Pieter Katoen. "The How and Why of Interactive Markov Chains." In: *8th Intl. Symp. Formal Methods for Components and Objects*. Vol. 6286. LNCS. Springer, 2009, pp. 311–337.

[Hon+97] Youpyo Hong, Peter A. Beerel, Jerry R. Burch, and Kenneth L. McMillan. "Safe BDD Minimization Using Don't Cares." In: *DAC*. 1997, pp. 208–213.

[HS02] Danny Hendler and Nir Shavit. "Non-blocking steal-half work queues." In: *PODC*. ACM, 2002, pp. 280–289.

[HTK08] Alexandre Hamez, Yann Thierry-Mieg, and Fabrice Kordon. "Hierarchical Set Decision Diagrams and Automatic Saturation." In: *PETRI NETS*. Ed. by Kees M. van Hee and Rüdiger Valk. Vol. 5062. Lecture Notes in Computer Science. Springer, 2008, pp. 211–230.

[ISM11] Masakazu Ishihata, Taisuke Sato, and Shin-ichi Minato. "Compiling Bayesian Networks for Parameter Learning Based on Shared BDDs." In: *AI 2011*. Vol. 7106. Lecture Notes in Computer Science. Springer, 2011, pp. 203–212.

[Kam+98] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto L. Sangiovanni-vincentelli. "Multi-valued decision diagrams: theory and applications." In: *Multiple-Valued Logic* 4.1 (1998), pp. 9–62.

[Kan+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. "LTSmin: High-Performance Language-Independent Model Checking." In: *TACAS 2015*. Vol. 9035. LNCS. Springer, 2015, pp. 692–707.

[KC90] S. Kimura and E.M. Clarke. "A Parallel Algorithm for Constructing Binary Decision Diagrams." In: *Proc. of IC on Computer Design: VLSI in Computers and Processors ICCD*. Sept. 1990, pp. 220–223.

[KIH92] S. Kimura, T. Igaki, and H. Haneda. "Parallel Binary Decision Diagram Manipulation." In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science* E75-A.10 (Oct. 1992), pp. 1255–62.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems." In: *CAV*. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[Kul13] Konrad Kulakowski. "Concurrent bisimulation algorithm." In: *CoRR* abs/1311.7635 (2013).

[Laa14] Alfons W. Laarman. "Scalable Multi-Core Model Checking." PhD thesis. Enschede, The Netherlands, 2014.

[LB06] Elsa Loekito and James Bailey. "Fast mining of high dimensional expressive contrast patterns using zero-suppressed binary decision diagrams." In: *SIGKDD 2006*. ACM, 2006, pp. 307–316.

[LMS14] Alberto Lovato, Damiano Macedonio, and Fausto Spoto. "A Thread-Safe Library for Binary Decision Diagrams." In: *SEFM*. Vol. 8702. LNCS. Springer, 2014, pp. 35–49.

[LPW10] Alfons Laarman, Jaco van de Pol, and Michael Weber. "Boosting multi-core reachability performance with shared hash tables." In: *FMCAD 2010*. IEEE, 2010, pp. 247–255.

[LPW11] Alfons W. Laarman, Jaco van de Pol, and Michael Weber. "Multi-Core LTSmin: Marrying Modularity and Scalability." In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Vol. 6617. LNCS. Springer, 2011, pp. 506–511.

[Mal+88] Sharad Malik, Albert R. Wang, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. "Logic verification using binary decision diagrams in a logic synthesis environment." In: *ICCAD 1998*. 1988, pp. 6–9.

[MC13] Malcolm Mumme and Gianfranco Ciardo. "An Efficient Fully Symbolic Bisimulation Algorithm for Non-Deterministic Systems." In: *Int. J. Found. Comput. Sci.* 24.2 (2013), pp. 263–282.

[MD02] D. Michael Miller and Rolf Drechsler. "On the Construction of Multiple-Valued Decision Diagrams." In: *32nd IEEE International Symposium on Multiple-Valued Logic (ISMVL 2002)*. 2002, pp. 245–253.

[Mei+14] Jeroen Meijer, Gijs Kant, Stefan Blom, and Jaco van de Pol. "Read, Write and Copy Dependencies for Symbolic Model Checking." In: *HVC*. Ed. by Eran Yahav. Vol. 8855. Lecture Notes in Computer Science. Springer, 2014, pp. 204–219.

[MF89] Yusuke Matsunaga and Masahiro Fujita. "Multi-level logic optimization using binary decision diagrams." In: *ICCAD 1989*. IEEE, 1989, pp. 556–559.

[MH98] Kim Milvang-Jensen and Alan J. Hu. "BDDNOW: A Parallel BDD Package." In: *FMCAD*. 1998, pp. 501–507.

[Min13] Shin-ichi Minato. "Techniques of BDD/ZDD: Brief History and Recent Activity." In: *IEICE Transactions* 96-D.7 (2013), pp. 1419–1429.

[Min93] Shin-ichi Minato. "Zero-suppressed BDDs for set manipulation in combinatorial problems." In: *Proceedings of the 30th international Design Automation Conference*. DAC '93. New York, NY, USA: ACM, 1993, pp. 272–277.

[Moo65] Gordon E Moore. "Cramming more components onto integrated circuits." In: *Proceedings of the IEEE* 38.10 (1965), pp. 114–117.

[MSD16] Tobias Maier, Peter Sanders, and Roman Dementiev. "Concurrent Hash Tables: Fast and General?(!)" In: *CoRR* abs/1601.04017 (2016).

[MSS07] Shin-ichi Minato, Ken Satoh, and Taisuke Sato. "Compiling Bayesian Networks by Symbolic Probability Calculation Based on Zero-Suppressed BDDs." In: *IJCAI 2007*. 2007, pp. 2550–2555.

[MVS09] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. "Idempotent work stealing." In: *PPOPP*. ACM, 2009, pp. 45–54.

[ODP15] Wytse Oortwijn, Tom van Dijk, and Jaco van de Pol. "A Distributed Hash Table for Shared Memory." In: *Parallel Processing and Applied Mathematics*. Vol. 9574. LNCS. Springer, 2015, pp. 15–24.

[OIY91] Hiroyuki Ochi, Nagisa Ishiura, and Shuzo Yajima. "Breadth-First Manipulation of SBDD of Boolean Functions for Vector Processing." In: *DAC*. 1991, pp. 413–416.

[Oli+06] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. "UTS: An Unbalanced Tree Search Benchmark." In: *LCPC*. Vol. 4382. LNCS. Springer, 2006, pp. 235–250.

[Oor15] Wytse Oortwijn. "Distributed Symbolic Reachability Analysis." MA thesis. University of Twente, Dept. of C.S., 2015.

[Oss10] Jörn Ossowski. "JINC – A Multi-Threaded Library for Higher-Order Weighted Decision Diagram Manipulation." PhD thesis. Rheinischen Friedrich-Wilhelms-Universität Bonn, Oct. 2010.

[PBF10] Artur Podobas, Mats Brorsson, and Karl-Filip Faxen. "A Comparison of some recent Task-based Parallel Programming Models." In: *3rd Workshop on Programmability Issues for Multi-Core Computers* (2010).

[Pel07] Radek Pelánek. "BEEM: benchmarks for explicit model checkers." In: *SPIN*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 263–267.

[PSC94] Yegnashankar Parasuram, Edward P. Stabler, and Shiu-Kai Chin. "Parallel implementation of BDD Algorithms using a Distributed Shared Memory." In: *HICSS (1)*. 1994, pp. 16–25.

[PT87] Robert Paige and Robert Endre Tarjan. "Three Partition Refinement Algorithms." In: *SIAM J. Comput.* 16.6 (1987), pp. 973–989.

[Pug90] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees." In: *Commun. ACM* 33.6 (1990), pp. 668–676.

[RA02] Karen A. Reay and John D. Andrews. "A fault tree analysis strategy using binary decision diagrams." In: *Rel. Eng. & Sys. Safety* 78.1 (2002), pp. 45–56.

[Rud93] R. Rudell. "Dynamic variable ordering for ordered binary decision diagrams." In: *ICCAD*. 1993, pp. 42–47.

[Sak+11] Yuko Sakurai, Suguru Ueda, Atsushi Iwasaki, Shin-ichi Minato, and Makoto Yokoo. "A Compact Representation Scheme of Coalitional Games Based on Multi-Terminal Zero-Suppressed Binary Decision Diagrams." In: *PRIMA 2011*. Vol. 7047. Lecture Notes in Computer Science. Springer, 2011, pp. 4–18.

[San+96] Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. "High Performance BDD Package By Exploiting Memory Hiercharchy." In: *DAC*. 1996, pp. 635–640.

[SB14] Julian Shun and Guy E. Blelloch. "Phase-Concurrent Hash Tables for Determinism." In: *SPAA*. ACM, 2014, pp. 96–107.

[SB96] Tony Stornetta and Forrest Brewer. "Implementation of an Efficient Parallel BDD Package." In: *DAC*. 1996, pp. 641–644.

[Sew+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors." In: *Commun. ACM* 53.7 (2010), pp. 89–97.

[Sia12] Tien Loong Siaw. "Saturation for LTSmin." MA thesis. University of Twente, Dept. of C.S., Feb. 2012.

[Soe+16] Mathias Soeken, Laura Tague, Gerhard W. Dueck, and Rolf Drechsler. "Ancilla-free synthesis of large reversible functions using binary decision diagrams." In: *J. Symb. Comput.* 73 (2016), pp. 1–26.

[Som01] Fabio Somenzi. "Efficient manipulation of decision diagrams." In: *STTT* 3.2 (2001), pp. 171–181.

[Som15] Fabio Somenzi. "CUDD: CU Decision Diagram Package Release 3.0.0." `http://vlsi.colorado.edu/~fabio/CUDD/`. 2015.

[ST09] Håkan Sundell and Philippas Tsigas. "Brushing the Locks out of the Fur: A Lock-Free Work Stealing Library Based on Wool." In: *2nd Swedish Workshop on Multi-Core Computing MCC09*. University of Borås. School of Business and Informatics, 2009, pp. 126–130.

[WB10] Ralf Wimmer and Bernd Becker. "Correctness Issues of Symbolic Bisimulation Computation for Markov Chains." In: *MMB&DFT*. Vol. 5987. LNCS. Springer, 2010, pp. 287–301.

[WC93] David B. Wagner and Brad Calder. "Leapfrogging: A Portable Technique for Implementing Efficient Futures." In: *PPOPP*. ACM, 1993, pp. 208–217.

[WDH10] Ralf Wimmer, Salem Derisavi, and Holger Hermanns. "Symbolic partition refinement with automatic balancing of time and space." In: *Perform. Eval.* 67.9 (2010), pp. 816–836.

[WHB06] Ralf Wimmer, Marc Herbstritt, and Bernd Becker. "Minimization of Large State Spaces using Symbolic Branching Bisimulation." In: *Proceedings of the 9th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2006), Prague, Czech Republic, April 18-21, 2006*. IEEE Computer Society, 2006, pp. 9–14.

[WHB07] Ralf Wimmer, Marc Herbstritt, and Bernd Becker. "Optimization techniques for BDD-based bisimulation computation." In: *17th GLSVLSI*. ACM, 2007, pp. 405–410.

[Wij15] Anton Wijs. "GPU Accelerated Strong and Branching Bisimilarity Checking." In: *TACAS 2015*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 368–383.

[Wim+06] Ralf Wimmer, Marc Herbstritt, Holger Hermanns, Kelley Strampp, and Bernd Becker. "Sigref- A Symbolic Bisimulation Tool Box." In: *ATVA*. Vol. 4218. LNCS. Springer, 2006, pp. 477–492.

[Wim+07] Ralf Wimmer, Holger Hermanns, Marc Herbstritt, and Bernd Becker. "Towards Symbolic Stochastic Aggregation." Tech. rep. SFB/TR 14 AVACS, 2007.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." In: *Commun. ACM* 52.4 (2009), pp. 65–76.

[YO97] Bwolen Yang and David R. O'Hallaron. "Parallel Breadth-First BDD Construction." In: *PPOPP*. 1997, pp. 145–156.

Summary

Decision diagrams are fundamental in computer science. They are extensively used in various fields, such as symbolic model checking, logic synthesis, fault tree analysis, test generation, data mining, Bayesian network analysis and game theory. Binary decision diagrams, which are the most common type of decision diagrams, are often used to represent Boolean functions, which are at the core of computer science.

A particular application that extensively uses decision diagrams is symbolic model checking. Model checking studies the properties of models, for example models of software, hardware, communication protocols, automated systems, and properties such as whether a program is correctly implemented, what the failure rate of a system is, how long the average person has to wait for a traffic light, etc. Models typically consist of states and transitions: the system is in a certain state (such as: all lights are red, or: some lights are green) and can transition from one state to another state.

A major challenge in model checking is that the computation time and the amount of memory required to compute interesting properties typically increases exponentially when the size of the models increases, when models are combined, or when we want to check more complex properties. One method to tackle this fundamental challenge is to consider sets of states instead of individual states, and to use Boolean functions represented by binary decision diagrams to encode these sets of states. Hence, symbolic model checking often consists mostly of operations on binary decision diagrams.

A major goal in computing is to perform ever larger calculations and to improve their performance and efficiency. The processing power of computers increases according to Moore's law, but as physical constraints limit the opportunities for higher clock speeds, the increases in processing power of modern chips mostly come from parallel processing. Efficient parallel algorithms are thus required to make effective use of the processing power of modern chips. Hence, research and development of scalable parallel algorithms is a key aspect to computing in the 21st century.

The main contribution of this thesis is the reusable multi-core decision diagram library Sylvan. Sylvan implements parallelized operations on various types of decision diagrams, in particular binary decision diagrams, multi-terminal binary decision diagrams and list decision diagrams. One of its strengths is that it can replace existing non-parallel decision diagram libraries to bring the processing power of multi-core machines to non-parallel applications.

In this thesis, we study the design of Sylvan and its two main ingredients, namely the parallel hash tables that store the decision diagram nodes and the operation cache, and the parallel framework Lace that we developed, which executes tasks in parallel using a work-stealing algorithm for load balancing and a novel double-ended queue to store the tasks of each thread. Furthermore, we study how Sylvan performs in two applications: state space exploration and bisimulation minimisation.

We look at state space exploration on a large benchmark set, using the existing model checking toolset LTSmin with Sylvan as a backend. We first only use the parallel operations offered by Sylvan in an otherwise sequential program to obtain a parallel speedup. We then also parallelize the state space exploration algorithm, exploiting a partitioning of the transition relation in LTSmin, and obtain an improved parallel speedup.

Bisimulation minimisation computes the smallest equivalent model of some input model according to some notion of equivalence, such as strong or branching bisimulation. For this application, we develop two new decision diagram algorithms for bisimulation minimisation and show that these significantly improve the efficiency of the minimisation algorithm, on top of the parallelism obtained by Sylvan.

In the experimental results presented in this thesis, we obtain a good parallel speedup: up to 29x with 48 threads when purely relying on Sylvan for parallel execution; up to 38x with 48 threads when the program that uses Sylvan is also parallelized. In a study that compares Sylvan with popular decision diagram implementations, we see that Sylvan is competitive with the other implementations when running single-core (3rd place of 6 implementations), and faster than the other implementations when more threads are used.

Samenvatting

Beslisbomen zijn een fundamenteel concept in de informatica. Beslisbomen worden gebruikt in verschillende onderzoeksgebieden, zoals het doorrekenen van (eigenschappen van) systemen, het analyseren van foutbomen, automatische testgeneratie, en speltheorie. Binaire beslisbomen zijn een van de meestgebruikte beslisbomen en worden onder meer gebruikt om Boolse functies te representeren, die aan de basis van de informatica liggen. Met name voor het doorrekenen van systemen, zoals software, hardware, communicatieprotocollen en andere systemen, zijn binaire beslisbomen één van de voornaamste technieken om de rekentijd en de hoeveelheid geheugen die nodig zijn bij het doorrekenen van systemen beheersbaar te houden.

De rekenkracht van computers neemt nog steeds toe. Vanwege natuurkundige belemmeringen zit deze toenemende rekenkracht al jaren niet meer in snellere processoren, maar in het gebruik van een steeds groter aantal rekenkernen en processoren. Om van deze toenemende rekenkracht gebruik te maken is het daarom essentieel om parallele algoritmen te ontwikkelen, die efficient gebruik maken van de rekenkracht van moderne computers.

De voornaamste bijdrage en onderwerp van dit proefschrift is de programmabibliotheek Sylvan, die operaties op beslisbomen implementeert voor computers met meerdere rekenkernen. Een van de voordelen van Sylvan is dat de operaties op beslisbomen parallel uitgevoerd worden ook als het bovenliggende algoritme niet parallel is. Dat betekent dat software die gebruik maakt van beslisbomen, maar zelf niet ontwikkeld is om door meerdere rekenkernen uitgevoerd te worden, automatisch parallel uitgevoerd wordt wanneer Sylvan gebruikt wordt voor de implementatie van beslisbomen.

Dit proefschrift behandelt het ontwerp van Sylvan en de twee belangrijkste componenten, allereerst de tabellen die de knopen van de beslisbomen bevatten en de tussenresultaten van de operaties op beslisbomen, en verder het onderdeel Lace, dat voor dit proefschrift is ontwikkeld om taken parallel uit te voeren, waarbij het verdelen van de werklast door middel van “werk stelen” gebeurt, en waarbij de uit te voeren taken worden opgeslagen in een zoge-

naamde twee-eindige wachtrij, die ook in dit proefschrift wordt beschreven. We hebben voor twee toepassingen de snelheid en efficientie van Sylvan bestudeerd: toestandsruimteverkenning bij het doorrekenen van systemen, en bisimulatieminimisatie.

Bij toestandsruimteverkenning (het bepalen van alle toestanden waarin een systeem zich kan bevinden) maken we gebruik van een verzameling referentiemodellen (de BEEM-verzameling), met behulp van LTSmin, een verzameling programma's voor het doorrekenen van systemen. Eerst kijken we naar het behaalde resultaat indien we enkel Sylvan gebruiken voor binaire beslisbomen en LTSmin verder ongemoeid laten; vervolgens gebruiken we een partitie van de toestandovergangsrelatie in LTSmin om ook de toestandsruimteverkenning zelf, het bovenliggende algoritme dus, parallel uit te voeren. Hierbij zien wij dat de parallele versnelling verbetert.

Bisimulatieminimisatie berekent het kleinste equivalente model van een gegeven model, gegeven een bepaalde notie van equivalentie, zoals sterke of vertakkende bisimulatie. Hiervoor ontwikkelen wij tevens twee nieuwe operaties op beslisbomen, speciaal voor bisimulatieminimisatie. Deze leveren een sterke verbetering op voor de efficientie van het algoritme, bovenop de behaalde parallele versnelling dankzij Sylvan.

De experimenten in dit proefschrift leveren een goede parallele versnelling op. Enkel het gebruik van Sylvan voor parallele versnelling, waarbij de rest van het programma dus niet parallel uitgevoerd wordt, levert tot 29x versnelling op met 48 rekenkernen. Wanneer we ook het bovenliggende programma parallel uitvoeren zien we tot 38x versnelling, wederom met 48 rekenkernen. In een studie samen met onder meer de vakgroep ISCAS (China) zien we dat Sylvan goed meekomt met de concurrentie wanneer slechts één rekenkern beschikbaar is (3e plek van de 6). Sylvan is sneller dan de concurrentie wanneer meer rekenkernen beschikbaar zijn.

Titles in the IPA Dissertation Series since 2013

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*.

Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

This thesis studies the parallelization of decision diagrams, a fundamental data-structure with applications in many fields, in particular symbolic model checking. Research into parallel processing is essential, as multi-core and many-core computers are ubiquitous. Graph algorithms such as decision diagram operations are known to be difficult to parallelize, as well as difficult to reason about. This is one of the reasons why parallelizing symbolic model checking is difficult.

The main result of this research is the multi-core decision diagram package Sylvan. We investigate scalable hash tables, load balancing via work stealing, using Sylvan for symbolic state space exploration and for symbolic bisimulation minimisation. The experimental results show high parallel speedup of up to 38x for benchmarks on a 48-core computer. Experiments that compare Sylvan to non-parallel decision diagram packages show that Sylvan is competitive when run with a single core, and faster when run with multiple cores.