The Parallelization of Binary Decision Diagram operations for model checking

Tom van Dijk

24 April 2012

Master's thesis

Department of Computer Science

UNIVERSITY OF TWENTE.

Graduation committee:

Prof.dr. J.C. van de Pol
Dr. M. Huisman
A.W. Laarman, MSc.

Contents

1 Introduction **1**

2 Preliminaries of Binary Decision Diagrams **3**
- 2.1 Model checking . . . 3
- 2.2 Storing states and transitions in memory . . . 5
- 2.3 Binary decision diagrams . . . 5

3 Implementation Techniques of BDDs **9**
- 3.1 BDDs in memory . . . 9
- 3.2 Implementation of BDD algorithms . . . 10
- 3.3 Using `ITE` for synthesis operations . . . 11
- 3.4 Model checking using `RelProd` . . . 13
- 3.5 Reducing redundancy using `RelProdS` . . . 15
- 3.6 Reversed `RelProdS` . . . 17
- 3.7 Conclusions . . . 20

4 Preliminaries of parallelism **21**
- 4.1 Parallelism . . . 21
- 4.2 Architectures . . . 21
- 4.3 Approaches to parallelization . . . 22
- 4.4 Theoretical models of multicore parallelism . . . 22
  - 4.4.1 Amdahl's Law . . . 23
  - 4.4.2 Extending Amdahl's Law . . . 23
  - 4.4.3 Communication bottlenecks . . . 24
- 4.5 Conclusions . . . 25

5 Implementation of parallelization **27**
- 5.1 Task distribution and result sharing . . . 27
- 5.2 Implementing task distribution using Wool . . . 28
- 5.3 Implementing result sharing . . . 29
- 5.4 Lockless data structures . . . 29
  - 5.4.1 Lockless memoization cache . . . 29
  - 5.4.2 Lockless hashtable with reference counting . . . 30
- 5.5 Conclusions . . . 34

6 Experiments **37**
- 6.1 Experimental setup . . . 37
- 6.2 Results using Wool . . . 39
- 6.3 Results using result sharing . . . 47
- 6.4 Comparing using result sharing to using Wool . . . 55
- 6.5 Changing memoization cache strategies . . . 57
- 6.6 Comparison to using BuDDy . . . 57
- 6.7 Conclusions . . . 62

7 Related work **63**

8 Conclusions and Future Work **65**
- 8.1 Conclusions . . . 65
- 8.2 Future Work . . . 66

List of Figures

2.1 BDD node representing $(x \wedge F_{x=1}) \vee (\overline{x} \wedge F_{x=0})$ . . . 6
2.2 BDDs of simple Boolean functions . . . 7
2.3 Reusing subgraphs using complement edges . . . 8

4.1 Typical task dependency graph . . . 22
4.2 Typical speedup graph . . . 23
4.3 Roofline model . . . 24

6.1 Wool results (1-48 workers, all models) . . . 40
6.2 Best speedup for Wool compared to work per model . . . 41
6.3 Wool relative work with increased workers . . . 43
6.4 Result of parallelizing Fibonacci using Wool . . . 44
6.5 Results of using interleaved memory allocation (36-48 workers, all models) . . . 45
6.6 Expected speedup graph with `compare_and_swap` . . . 46
6.7 Results of using result sharing . . . 48
6.8 Relative work with result sharing . . . 49
6.9 Results compensated for redundant work . . . 50
6.10 Speedup versus relative work using result sharing (only best result) . . . 51
6.11 Relative work versus work . . . 52
6.12 Results of using the kill flag (36-48 workers, all models) . . . 54
6.13 Results of using interleaved memory allocation (36-48 workers, all models) . . . 54
6.14 Results of setting CPU affinity (only 48 workers, all models) . . . 55
6.15 Best speedup for result sharing compared to work per model . . . 56
6.16 Result sharing vs Wool (best) . . . 57
6.17 Result sharing vs Wool (1) . . . 57
6.18 Result sharing vs Wool (36) . . . 57
6.19 Result sharing vs Wool (48) . . . 57
6.20 Wool with N=2 (1-4 workers) . . . 58
6.21 Result sharing with N=2 (1-4 workers) . . . 58
6.22 Wool with N=3 (1-4 workers) . . . 58
6.23 Result sharing with N=3 (1-4 workers) . . . 58
6.24 Wool with N=4 (1-4 workers) . . . 58
6.25 Result sharing with N=4 (1-4 workers) . . . 58
6.26 Wool with N=2 (36-48 workers) . . . 59
6.27 Result sharing with N=2 (36-48 workers) . . . 59
6.28 Wool with N=3 (36-48 workers) . . . 59
6.29 Result sharing with N=3 (36-48 workers) . . . 59
6.30 Wool with N=4 (36-48 workers) . . . 59
6.31 Result sharing with N=4 (36-48 workers) . . . 59
6.32 Wool speedup with N=4 (36-48 workers) . . . 60
6.33 Wool extra work (N=4) . . . 60
6.34 Result sharing extra work (N=4) . . . 60

7.1 Speedup graph of 13-bit Multiplier Construction using Cilk . . . 64

List of Tables

3.1 Boolean operators and their ITE variants. 11

6.1 Performed experiments on all selected models using 1-48 workers 37
6.2 Model data 42
6.3 CPU profile of bakery.4 with 48 workers 44
6.4 CPU profile of bakery.8 with 48 workers 45
6.5 Results of using a kill flag for a single operation 53
6.6 Results of using a kill flag for model collision.5 53
6.7 Extra work for $N = 4$ 61
6.8 Results using BuDDy, Wool, result sharing 61

1

Introduction

In this modern era humanity surrounds itself by technology impossible to be fully understood even by its own designers. This may lead to serious problems, when it becomes impossible to guarantee that a power plant will function properly, that the traffic lights prevent accidents rather than causing them, that aircrafts stay in the air in all foreseeable conditions and that the television actually responds to the button pressed on the remote control.

Paradoxically, technology partially alleviates this problem. By creating an abstraction of the systems that surround us we can use computers to verify that complex systems function properly according to certain properties. This is called *model checking*. In model checking, systems such as power plants, aircrafts, traffic lights and televisions, are modeled as a set of possible *states* the system can be in and a set of *transitions* between these states. In addition there are one or more *initial states* that describe the states the system could be in initially. States and transitions form a *transition system* that describes system behavior. Formal logics like Computational Tree Logic (CTL) and Linear Temporal Logic (LTL) can then be used to formally specify properties as formulae, e.g., the property that a system is free of deadlocks and that no forbidden states can be reached.

At the core of model checking sits the *reachability* algorithm, which calculates all possible states a system can be in based on the initial states and the transitions. Adaptations of this algorithm can calculate all states with certain properties that are reachable from states with certain properties, in order to determine whether a system obeys the CTL or LTL formulae. One of the problems with model checking is the size of the transition system. Even with small systems, the computational power and memory required to store all explored states is enormous. One way to deal with this problem is to not store every state individually, but to represent all states using Boolean functions. This is called *symbolic model checking* [14].

Boolean functions are generally stored in memory using Binary Decision Diagrams (BDDs) [5, 12, 13] that are usually stored in memory using a hashtable. In order to manipulate Boolean functions stored using BDDs there are several BDD operations. To calculate the reachable states, only four operations are necessary: calculating $\wedge$, calculating $\exists$, calculating a substititution and calculating $\vee$. In common BDD implementations there is a special algorithm to calculate the *relational product* which combines $\wedge$ and $\exists$.

In this report we present a new algorithm that combines this relational product with the substitution. This algorithm is more efficient than calculating the two operations separately.

Since model checking already has huge computational and memory requirements, techniques that increase the performance of model checking tools are constantly being developed. Recent developments in hardware and software development focus on using multi-core and multi-processor architectures. In order to use the computational power of all cores we need to *parallelize* our software, i.e., divide our algorithms into smaller parts that can be executed in parallel by multiple *workers* (usually one worker per processor core), in such a way that there is maximum *speedup*. To maximize speedup we need to minimize overhead caused by memory transactions and communication between workers. This means we need to develop data structures and al-

gorithms to manipulate these data structures that have good *scalability*, i.e., they perform well even when heavily used by many workers.

In this report we discuss two data structures that we developed with scalability in mind. These data structures are based on the *lockless* paradigm, which avoids mutual exclusion and depends on atomic operations instead. We also present two approaches to parallelize BDD algorithms. One approach uses the framework for task-based parallelism Wool [19]. The other approach uses a framework for result sharing we developed. In result sharing, all workers perform the same calculation, but share results of subcalculations. This approach is similar to the Swarm Verification of Holzmann et al. [26]

We have performed experiments with several models from the BEEM database [37] using the `LTSmin` toolset [9] extended to support our experimental BDD package Sylvan. We compared the results to the performance of the same toolset using the BDD package BuDDy as the backend for symbolic model checking. The results show that our approach gives a significant speedup compared to BuDDy and that even better speedups should be possible in the future.

This report is structured as follows. Chapter 2 introduces binary decision diagrams. The implementation of BDDs and the BDD algorithms `ITE`, `RelProd`, `RelProdS` and `RRelProdS` are discussed and proven to be correct in Chapter 3. Chapter 4 introduces parallelism and presents theoretical approaches to parallelism, including our extension of Amdahl's Law as a qualitative performance model that can be used to understand limited performance. The two approaches to parallalization that we studied as well as our design of two data structures, a lockless lossy memoization cache and a lockless hashtable that supports lazy garbage collection by reference counting, are presented in Chapter 5. The experiments and results are presented in Chapter 6. Chapter 7 discusses related work and Chapter 8 concludes the report and contains ideas for future work.

2

Preliminaries of Binary Decision Diagrams

The current chapter explains the relation between model checking and binary decision diagrams (BDDs). We also discuss some properties of BDDs that are relevant for our research.

2.1 Model checking

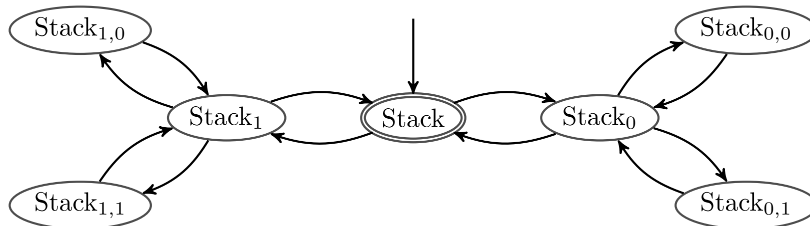
Model checking is a method to verify properties of systems, protocols, programs, et cetera, by manually specifying or automatically generating an abstract model of the system. This abstract model can then be used to verify that certain properties hold in all reachable states of the system or over all paths in the state graph. See also below.

First we define what a model is. A model is a triple $(X, S_{\text{initial}}, R)$ where X is a set of variable names, $S_{\text{initial}} \subseteq S$ is the set of initial states and R is a transition relation defining all possible transitions in the system.

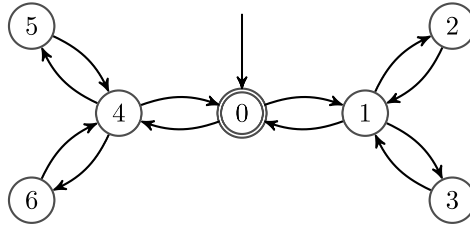
We assume some set $X = \{x_1, \ldots, x_n\}$ of *variable names*. A *state* $s\colon X \to \texttt{Bool}$ is a valuation of all variables in X. For instance, $s(x_1) = \texttt{true}$ and $s(x_2) = \texttt{false}$. We denote such a state by $x_1\overline{x_2}$. Let S be the set of all states, i.e., all possible valuations of the variables in X. A subset $V \subseteq S$ can be denoted by a Boolean function $F\colon S \to \texttt{Bool}$, indicating which states are in the set. For example, we can define $F = \{s \mid \neg s(x_1)\}$, which is the set of all states that assign `false` to the variable x_1. For ease of notation, we denote such a function by $F(s) = \overline{x_1}$.

A *transition relation* is a relation $R \subseteq S \times S$. We can again denote it by a Boolean function $T\colon S \times S \to \texttt{Bool}$. For instance, $T(s, s') = \{(s, s') \mid s(x_1) \wedge \neg s(x_2) \wedge s'(x_2)\}$. This relation defines two transitions: one between the state $x_1\overline{x_2}$ and x_1x_2, and one between $x_1\overline{x_2}$ and $\overline{x_1}x_2$. For ease of notation, we will again abbreviate $s(x_i)$ by x_i and $\neg s(x_i)$ by $\overline{x_i}$. Also, we abbreviate $s'(x_i)$ by x'_i and $\neg s'(x_i)$ by $\overline{x'_i}$. So, for the example above we have $T(s, s') = x_0\overline{x_1}x'_1$.

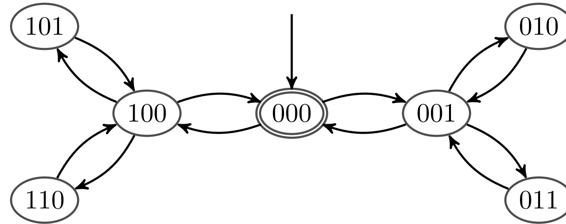
As an example of a model we consider a stack that can contain two Booleans. There is one initial state, which is the state where the stack is empty.



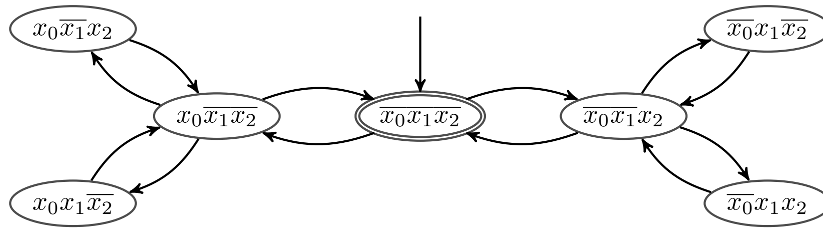
This model of the stack has 7 states. There are various ways to create an abstract model using Boolean variables. As an example, we simply assign a number to each state:



Instead of decimal numbers, we could use binary numbers:



Now we can simply use Boolean variables x_0, x_1 and x_2:



We have encoded the states using three Boolean variables. The model consists of one initial state (state $s = \overline{x_0x_1x_2}$) and it has 12 transitions.

The set of *reachable states* consists of all states s that are in S_{initial} or for which there is a sequence of states $s_0, ..., s_n$ such that

1) $s_0 \in S_{\text{initial}}$
2) $(s_{i-1}, s_i) \in R$, for $i \in \{1, \ldots, n\}$
3) $s_n = s$

Based on the initial states and the transitions, all reachable states can be calculated using a *reachability* algorithm (see Section 3.4). This is an algorithm that iteratively calculates all states that are reachable from the current set of reachable states, starting with the set of initial states, until no new reachable states are found. In the example model, states $x_0\overline{x_1x_2}$ and $\overline{x_0x_1}x_2$ will be found in the first iteration of the algorithm and the last four states will be found in the second iteration of the algorithm. In the third iteration, no new states are found and the algorithm terminates.

In our simple example we already knew the reachable states in advance. In many models, this is not the case and often only the initial states and the transition relation are known. For example, many models are compositions of smaller models with known transition relations.

Testing properties Model checking can be used to calculate all reachable states of algorithms and to verify that certain properties hold. Examples are the property that every state has one or more transitions to other states, and the property that certain states are visited on every infinite path in the state graph. Properties are often expressed in temporal logics like CTL [28] and LTL [46].

For example consider Dekker's algorithm. This is a mutual exclusion algorithm for two processes:

Process 1:

```
1  flag[0] := true
2  while flag[1] = true
3     if turn != 0
4        flag[0] := false
5        while turn != 0 { }
6        flag[0] := true
7  // critical section
8  turn := 1
9  flag[0] := false
```

Process 2:

```
1  flag[1] := true
2  while flag[0] = true
3     if turn != 1
4        flag[1] := false
5        while turn != 1 { }
6        flag[1] := true
7  // critical section
8  turn := 0
9  flag[1] := false
```

The goal of mutual exclusion is that it is impossible for processes to be in the critical section at the same time. Therefore there must not be a reachable state where both processes are at line 9.

This algorithm can be modeled using 5 variables: the three Boolean variables f_0 (flag[0]), f_1 (flag[1]) and t (turn), plus program counters p_1 and p_2. Since the program counter can have 9 different values, 4 Boolean variables are required to represent the program counter. This results in a total of 11 Boolean variables.

The transitions follow from the algorithm. For example, line 1 of process 1 will be translated to the transitions with $p_1 = 1$, $p'_1 = 2$, $f'_0 = 1$, $p'_2 = p_2$, $f'_1 = f_1$ and $t' = t$, i.e., all transitions from states where $p_1 = 1$ to states where $p_1 = 2$ and $f_0 = 1$, and all other variables are unchanged. There are 44 such transitions, because there are 11 values for p_2, 2 values for f_1 and 2 values for t. Line 2 of process 1 will be translated to two sets of transitions, one for the case where $f_1 = 1$ (go to line 3) and one for the case where $f_0 = 0$ (go to line 7).

By calculating the transitions of process 1 and process 2, we get transition relations R_1 and R_2. The set of transitions of the entire model is $R = R_1 \cup R_2$ and the set of initial states consists of all states where $p_1 = 1$ and $p_2 = 1$.

We can now calculate all reachable states and verify whether or not there is a reachable state where both processes are in the critical section, i.e., $p_1 = 7$ and $p_2 = 7$, in which case the algorithm can lead to a forbidden state. We could also verify whether there are any deadlocks and whether on infinite paths both processes can enter their critical section infinitely often.

2.2 Storing states and transitions in memory

In general, there are two methods to store the set of visited states.

1) *Explicit model checking*, in which every state is stored individually in a hash table or similar data structure.

2) *Symbolic model checking* [14], in which the set of visited states is represented by a Boolean function.

In this thesis we are only interested in symbolic model checking. Symbolic model checking has several advantages. First of all, small Boolean functions can represent a large number of states. Because of this property some models require much less memory with symbolic model checking. Also, as we will see in Section 2.3, testing whether two sets are equal (for example in the reachability algorithm, to see if there are any new states) is trivial in symbolic model checking since BDDs are canonical representations of Boolean functions. A disadvantages is that manipulating BDDs, for example adding a single state, takes more time than in explicit model checking.

2.3 Binary decision diagrams

Binary decision diagrams were introduced by Akers [5] and developed by Bryant [12, 13]. Because BDDs represent Boolean functions, they can be used to store sets of states in symbolic model checking. A BDD is a directed acyclic graph expressing the Shannon decomposition of a Boolean function.

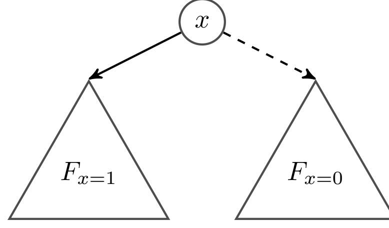


Figure 2.1: BDD node representing $(x \wedge F_{x=1}) \vee (\overline{x} \wedge F_{x=0})$

Definition of restriction. Let

$$s[x \leftarrow v] \stackrel{\text{def}}{=} \lambda y. \text{ if } (x = y) \text{ then } v \text{ else } s(y)$$

be the definition of substitution, then the definition of *restriction* is

$$F_{x=v}(s) \stackrel{\text{def}}{=} F(s[x \leftarrow v]).$$

An alternative definition is: Let F be a Boolean function on $X = \{x_1, \ldots, x_n\}$. The restrictions (also called *projections* or *cofactors*) $F_{x_i=1}$ and $F_{x_i=0}$ are Boolean functions defined as follows:

$$F_{x_i=1}(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) \stackrel{\text{def}}{=} F(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$
$$F_{x_i=0}(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) \stackrel{\text{def}}{=} F(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

Theorem. Let $F\colon S \to \texttt{Bool}$ be a Boolean function on S. Then

$$\big(F(s) \wedge x = v\big) \leftrightarrow \big(F_{x=v}(s) \wedge x = v\big)$$

and from this follows:

$$F(s) \equiv (xF_{x=1}(s)) \vee (\overline{x}F_{x=0}(s))$$

This identity is called the *Shannon expansion* or *Shannon decomposition* of F with respect to x [41]. For example, given a function $F = x_1(x_2 \vee \overline{x_3})$ we have $F_{x_2=1} = x_1$ and $F_{x_2=0} = x_1\overline{x_3}$. Therefore applying the Shannon decomposition gives $F \equiv (x_2x_1) \vee (\overline{x_2}x_1\overline{x_3})$.

A BDD over a set X has internal nodes labeled by a Boolean variable in X and leaves labeled 1 and 0. Every internal node with label x has two outgoing edges, one labeled 1 to the subgraph representing $F_{x=1}$ and one labeled 0 to the subgraph representing $F_{x=0}$. See also Figure 2.1.

We use the following convention to draw BDDs. Internal nodes are drawn as circles, subgraphs are drawn as triangles and leaves are drawn as boxes with a 1 or a 0. Edges of a node with variable x to $F_{x=1}$ are called 1-edges and are drawn solid. Edges to $F_{x=0}$ are called 0-edges and are drawn dashed. By repeatedly applying the Shannon decomposition, any Boolean function can be represented by a BDD. Examples of simple BDDs are given in Figure 2.2.

An *ordered BDD* is a BDD where all variables in all paths from the root to a leaf are encountered according to a total ordering $<$.

On every path from the root of an ordered BDD to a leaf, every variable is encountered at most once. Given a state s, an ordered BDD can be evaluated by following the 1-edges or the 0-edges, depending on the valuation of the variables in X. Every path from the root to leaf 1 represents a subset of F, where every variable on the path is either `true` or `false` depending on which edge was followed. For every variable x_i that is not on a specific path from the root to leaf 1, there are states with $x_i = \texttt{true}$ and states with $x_i = \texttt{false}$ in the set.

In order for an ordered BDD to be called *reduced*, no nodes must exist with two identical child nodes (these nodes are redundant) and no duplicate subgraphs must exist. All examples in Figure 2.2 are ordered and reduced. Any BDD can be reduced by applying the following two rules:

1) Eliminate redundant nodes.
2) Eliminate duplicate subgraphs by sharing subgraphs.

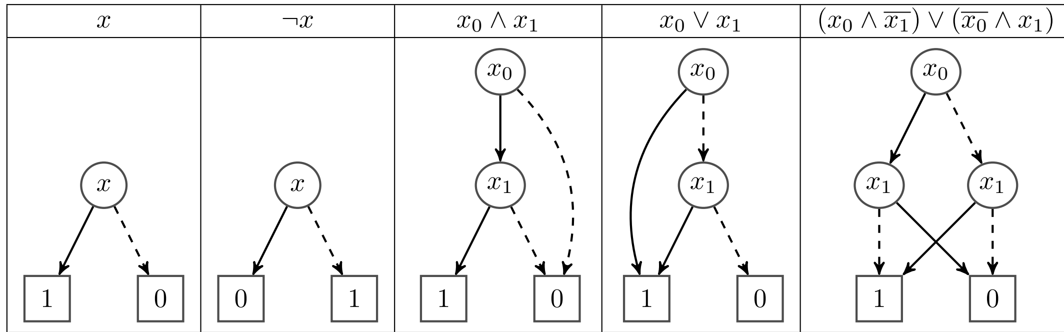
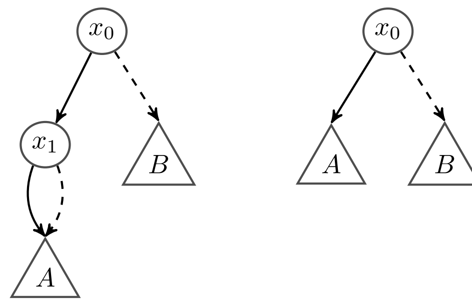
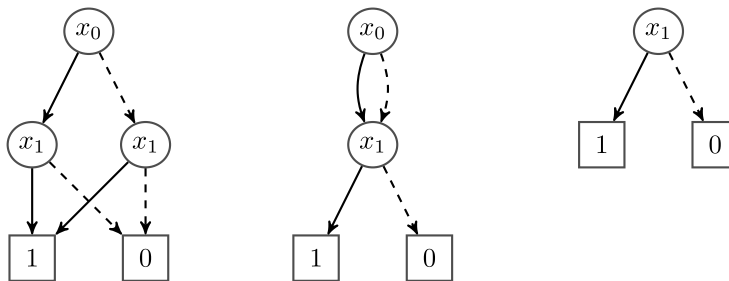


Figure 2.2: BDDs of simple Boolean functions

Note how the following two BDDs represent the same Boolean function. In a reduced BDD there are no redundant nodes like node x_1 in the left BDD.



Note how the following three BDDs also represent the same Boolean function. In the left BDD there is a duplicate subgraph. In the middle BDD this duplication is eliminated, but now there is a redundant node. The right BDD is a reduced BDD.



Reducing an ordered BDD by applying these two rules eliminates all possibilities to represent the same Boolean function in multiple ways. Therefore, one of the properties of a reduced ordered BDD is that it gives a *canonical* representation of Boolean functions. Because reduced ordered BDDs are canonical, testing whether two sets represented by BDDs are equal is trivial.

In this thesis we assume every BDD is reduced and ordered.

Complement edges

To decrease memory usage and calculation time, subgraphs can be reused to represent related Boolean functions by adding attributes to edges in a BDD. Several methods are mentioned in the literature [35, 11, 32, 33].

Complement edges (also called negated edges) indicate that the leaves 1 and 0 will be switched, negating the Boolean function represented by the node. Complement edges were introduced by Minato [35] and Brace, Rudell and Bryant [11].

We use a filled circle to denote a complement edge. For example, the following BDDs all represent the Boolean function $F = x_0$. In the two rightmost BDDs a subgraph is shared.

We will use the symbol $\neg$ to denote a complement edge in Boolean functions. For example, the second BDD represents the function $\neg((x_0 \wedge \neg 1) \vee (\overline{x_0} \wedge \neg 0))$. When evaluating $\neg F$, where

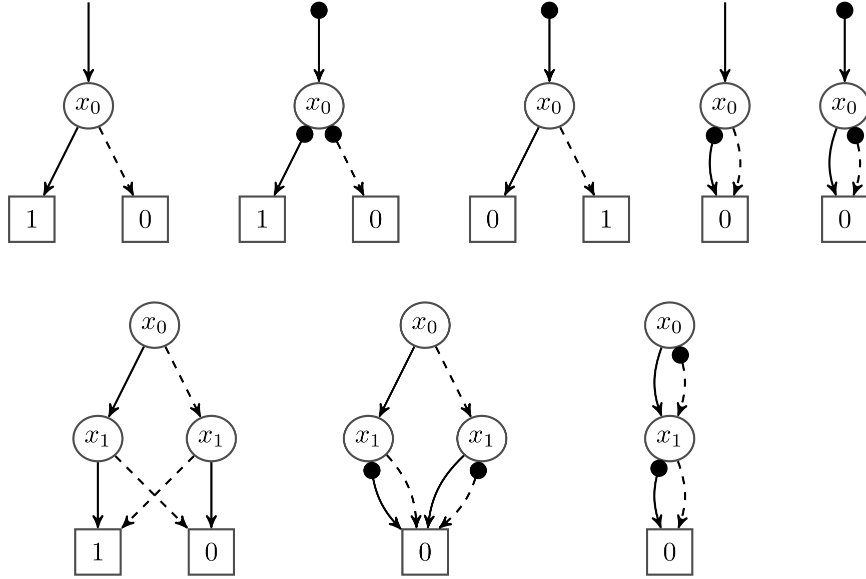


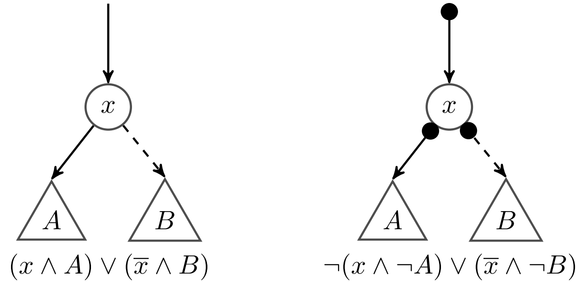
Figure 2.3: Reusing subgraphs using complement edges

$F = (x \wedge G) \vee (\overline{x} \wedge H)$, the complement carries over, since cofactors distribute over negation:

$$(\neg F)_{x=1} = \neg(F_{x=1}) = \neg G$$
$$(\neg F)_{x=0} = \neg(F_{x=0}) = \neg H$$

A more complex example that shows how complement edges allow reusing subgraphs is given in Figure 2.3. The left BDD is the BDD without complement edges. In the middle BDD, complement edges are used to reuse the subgraph 0. In the right BDD, the subgraph x_1 is also reused.

Using edge attributes breaks the property that BDDs uniquely represent Boolean functions. The following BDDs represent the same Boolean function:



To maintain canonicity we need a rule that selects exactly one of the alternatives, like the rule that forbids redundant nodes and the rule that forbids duplicate subgraphs. Minato [35] proposes the following constraints:

1) Only 0 is used as the value of a leaf.
   Alternative rule 1: only 1 is used as the value of a leaf.
   Alternative rule 2: do not allow complement edges on leaves.

2) No complement edges are allowed on 0-edges.
   Alternative rule: no complement edges are allowed on 1-edges.

An advantage of complement edges is that negation is trivial to implement.

We will use this property in section 3.3 to improve various BDD algorithms.

3

Implementation Techniques of BDDs

The current chapter discusses the implementation of BDDs. First we discuss how BDDs are usually stored in memory. Then we discuss the most important BDD operations for model checking. We present the existing algorithm `ITE` that is used to calculate the application of all binary Boolean operators. We also present the existing algorithm `RelProd` that is used to calculate the relational product of the Boolean function representing the set of states and the Boolean function representing the transition relation.

After discussing existing material we present our new specialized algorithm for the calculation of the relational product with substitution, `RelProdS`, and its dual, `ReversedRelProdS`. We suggest these algorithms are more efficient than separately calculating `RelProd` and applying substitution on the result.

3.1 BDDs in memory

BDD nodes are stored in memory using memory arrays. Every BDD node represents a BDD. A reference to a BDD is the index in that memory array [27]. We assign a unique label to every variable $x_1, \ldots, x_n$ in X. Every BDD node consists of this variable label, the reference to the 1-edge BDD, the reference to the 0-edge BDD and one bit to indicate a complement on the 1-edge.

In addition to the memory array holding the BDD nodes, a *Unique Table* is necessary to ensure that there are no duplicate nodes. This Unique Table is usually implemented as a hash table. In some BDD implementations, the array of BDD nodes is merged with the Unique Table, such that the hash table stores the nodes, instead of references to the nodes. The advantage is that the implementation is simpler. The disadvantage is that there is less memory locality, since nodes in a hash table are not stored together. In this thesis we focus on using a single hash table to store all BDDs, mainly because the implementation is much simpler.

BDD implementations also require *Computed Tables*, which are memoization caches for the operations. See Section 3.2 for more details.

Garbage collection is essential for BDDs. Every modification to a subgraph in a BDD also implies the modification of all ancestors in that BDD[1]. Unused BDD nodes should be deleted to free space for new BDD nodes. However, Somenzi [43] mentions that unused BDD nodes are often reused and that garbage collection should only be performed when there are enough *dead nodes* to justify the cost of garbage collection and recreating nodes that were deleted during garbage collection. The data structures used to store BDDs should therefore support garbage collection, for example using reference counting or mark-and-sweep approaches.

In BDD implementations, the method `MK` is used to create or reuse BDD nodes. This method uses the rules defined in Section 2.3 to guarantee canonicity. The implementation of `MK` is given

[1]While it could be possible to modify nodes rather than creating new nodes in the BDD table, this will also affect other BDDs that use the modified BDD. Also, modifying nodes usually requires an update in the Unique Table, so creating a new BDD node is actually less expensive than modifying an existing node.

in Listing 3.1.

```
BDD MK(int x, BDD F_{x=1}, BDD F_{x=0})
   if F_{x=0} = F_{x=1}
      return F_{x=0}
   if complemented(F_{x=0})
      return compl(hashtable.getOrCreate(x, compl(F_{x=1}), compl(F_{x=0})))
   return hashtable.getOrCreate(x, F_{x=1}, F_{x=0})
```

Listing 3.1: Implementation of MK

Theorem 3.1. *If x is less than all variables in $F_{x=1}$ and $F_{x=0}$ according to the ordering $<$, and $F_{x=1}$ and $F_{x=0}$ are reduced ordered BDDs according to $<$, then* MK *returns a reduced ordered BDD according to $<$ representing $(x \wedge F_{x=1}) \vee (\overline{x} \wedge F_{x=0})$ according to the rules in Section 2.3.*

Proof. In the first case, $F_{x=0}$ is returned which is a reduced ordered BDD according to $<$ and since $F_{x=1} = F_{x=0}$ the result obeys the specification. In the second and third case, using a hashtable guarantees that no duplicate subgraphs are created. In the second case, creating duplicates due to complementation is prevented. Since x is less than $F_{x=0}$ and $F_{x=1}$, the result is a reduced ordered BDD according to $<$. □

3.2 Implementation of BDD algorithms

BDD packages implement a number of operations on BDDs. Various basic operations mentioned by Drechsler and Sieling [18] are:

- *evaluation*: calculating cofactors, following edges
- *equivalence testing*: check whether two functions are equivalent
- *satisfiability*: enumerating and counting one or all satisfying valuations
- *synthesis*: creating new BDDs, e.g. by applying operators to functions
- *substitution*: replacing variables by other variables or functions
- *quantification*: calculating $\forall$ and $\exists$

Most BDD operations are recursively defined based on the Shannon decomposition. Such operations consist of selecting a variable x, recursively calculating the results of the subproblems where all parameters are restricted to $x = 0$ or $x = 1$, and calculating the end result, usually by creating a BDD node representing $(x \wedge F_{x=1}) \vee (\overline{x} \wedge F_{x=0})$.

BDD operations are typically implemented using *dynamic programming*, which is a method to solve problems by breaking them down into smaller subproblems [8]. There is a distinction between *top-down dynamic programming* and *bottom-up dynamic programming*. In top-down dynamic programming, the computation starts at the root problem and subproblems are recursively calculated. To avoid calculating subproblems multiple times, the results are cached using a memoization cache. In bottom-up dynamic programming, the smallest subproblems are first solved and their solutions are used to solve bigger subproblems until the root problem has been solved.

Bottom-up dynamic programming does not work well for BDD operations, because it is not known in advance which subproblems need to be solved. The number of possible subproblems is many times larger than the number of required subproblems to solve the root problem. Therefore top-down dynamic programming is used to implement BDD operations.

In the following sections we describe four BDD algorithms. We first specify each algorithm, then prove the correctness of the specification. We give an implementation of the algorithm. We define *rewrite rules* to normalize equivalent parameters (e.g. the result of $A \wedge B$ is the same as the result of $B \wedge A$) for every operation.

We assume the following equivalences in the correctness proofs:

$$F \equiv (x \wedge F_{x=1}) \vee (\overline{x} \wedge F_{x=0}) \qquad \text{Shannon decomposition}$$

$$(F \vee G)_{x=v} \equiv F_{x=v} \vee G_{x=v} \qquad \text{distribution of restriction over } \vee$$

$$(F \wedge G)_{x=v} \equiv F_{x=v} \wedge G_{x=v} \qquad \text{distribution of restriction over } \wedge$$

$$(\exists x F)_{y=v} \equiv \exists x (F_{y=v}) \qquad \text{commutivity of restriction and } \exists \text{ if } x \neq y$$

$\exists x F \equiv F_{x=1} \vee F_{x=0}$	definition of $\exists$
$\exists x(F \vee G) \equiv (\exists x F) \vee (\exists x G)$	distribution of $\exists$ over $\vee$
$(F \vee G)[S] \equiv F[S] \vee G[S]$	distribution of substitution over $\vee$
$(F \wedge G)[S] \equiv F[S] \wedge G[S]$	distribution of substitution over $\wedge$
$\overline{F}[S] \equiv \overline{F[S]}$	commutivity of substitution and negation

We also assume the notation $\exists X$ as shorthand for $\exists x_1 \dots \exists x_n$.

3.3 Using ITE for synthesis operations

One of the most basic BDD operations is applying a Boolean operator $F \ddagger G$, where $\ddagger$ is a binary Boolean operator (see Table 3.1). These operations can be calculated using the If-Then-Else (ITE) operation, which has the specification

$$\texttt{ITE}(A, B, C) \equiv (A \wedge B) \vee (\overline{A} \wedge C).$$

An alternative but equivalent specification is $\texttt{ITE}(A, B, C) \equiv (A \rightarrow B) \wedge (\overline{A} \rightarrow C)$. Common Boolean operators can be calculated using ITE as described in Table 3.1.

Table 3.1: Boolean operators and their ITE variants.

Boolean operator	ITE
$F \wedge G$	ITE(F, G, 0)
$F \vee G$	ITE(F, 1, G)
$F \oplus G$	ITE(F, $\overline{G}$, G)
$\neg(F \wedge G)$	ITE(F, $\overline{G}$, 1)
$\neg(F \vee G)$	ITE(F, 0, $\overline{G}$)
$F \rightarrow G$	ITE(F, G, 1)
$F \leftarrow G$	ITE(F, 1, $\overline{G}$)
$F \leftrightarrow G$	ITE(F, G, $\overline{G}$)
$\overline{F} \wedge G$	ITE(F, 0, G)
$F \wedge \overline{G}$	ITE(F, $\overline{G}$, 0)

Definition 3.2 (ITE operation)**.** Let $\texttt{ITE}_{x=v}$ be shorthand for $\texttt{ITE}(A_{x=v}, B_{x=v}, C_{x=v})$ with $v \in \{0, 1\}$ and let x be the top variable of A, B and C. Then ITE is defined as follows:

$$\texttt{ITE}(A, B, C) = \begin{cases} B & A = 1 \\ C & A = 0 \\ \texttt{MK}(x, \texttt{ITE}_{x=1}, \texttt{ITE}_{x=0}) & \text{otherwise} \end{cases}$$

Theorem 3.3. *$\texttt{ITE}(A, B, C)$ obeys the specification $(A \wedge B) \vee (\overline{A} \wedge C)$.*

Proof. By induction on the size of A, B and C. We assume the induction hypothesis $\texttt{ITE}_{x=v} \equiv (A_{x=v} \wedge B_{x=v}) \vee (\overline{A_{x=v}} \wedge C_{x=v}) \equiv ((A \wedge B) \vee (\overline{A} \wedge C))_{x=v}$ (by distribution of restriction). There are three cases:

1) If $A = 1$, then $B = (1 \wedge B) \vee (0 \wedge C) = (A \wedge B) \vee (\overline{A} \wedge C)$
2) If $A = 0$, then $C = (0 \wedge B) \vee (1 \wedge C) = (A \wedge B) \vee (\overline{A} \wedge C)$
3) Else, then

$$\begin{aligned} \texttt{MK}(x, \texttt{ITE}_{x=1}, \texttt{ITE}_{x=0}) &\stackrel{1}{=} (x \wedge \texttt{ITE}_{x=1}) \vee (\overline{x} \wedge \texttt{ITE}_{x=0}) \\ &\stackrel{2}{=} (x \wedge ((A \wedge B) \vee (\overline{A} \wedge C))_{x=1}) \vee (\overline{x} \wedge ((A \wedge B) \vee (\overline{A} \wedge C))_{x=0}) \\ &\stackrel{3}{=} (A \wedge B) \vee (\overline{A} \wedge C) \end{aligned}$$

Step 1: by definition. Step 2: by induction hypothesis. Step 3: by Shannon decomposition.

Furthermore, since x is the top variable of A, B and C, always at least one of the parameters to $\mathtt{ITE}_{x=v}$ is smaller than the original. □

Theorem 3.4. *The result of* $\mathtt{ITE}(A, B, C)$ *is a reduced ordered BDD according to* $<$ *if* A, B *and* C *are reduced ordered BDDs according to* $<$.

Proof. This is automatically true in case 1 and case 2, since B and C are reduced ordered BDDs according to $<$. In case 3, the result of MK is only a reduced ordered BDD according to $<$ if x is less than all variables in $\mathtt{ITE}_{x=1}$ and $\mathtt{ITE}_{x=0}$, and if $\mathtt{ITE}_{x=1}$ and $\mathtt{ITE}_{x=0}$ are ordered according to $<$. $A_{x=v}$, $B_{x=v}$ and $C_{x=v}$ can only contain variables greater than x since they are represented by ordered BDDs and x is the top variable of A, B and C. Since A, B and C are ordered BDDs, $A_{x=v}$, $B_{x=v}$ and $C_{x=v}$ are also ordered BDDs. Therefore x is less than all variables in $\mathtt{ITE}_{x=1}$ and $\mathtt{ITE}_{x=0}$ and the result of ITE is a reduced ordered BDD. □

Implementation The implementation of ITE is given in Listing 3.2.

```
BDD ITE(BDD A, BDD B, BDD C)
   // (1) Terminating cases
   if A = 1 then return B
   if A = 0 then return C

   // (2) Check cache
   if inCache(A, B, C) then return result

   // (3) Calculate top variable and cofactors
   x := topVariable(x_A, x_B, x_C)
   A_0 := cofactor0(A, x) B_0 := cofactor0(B, x) C_0 := cofactor0(C, x)
   A_1 := cofactor1(A, x) B_1 := cofactor1(B, x) C_1 := cofactor1(C, x)

   // (4) Recursive calls
   R_0 := ITE(A_0, B_0, C_0)
   R_1 := ITE(A_1, B_1, C_1)

   // (5) Calculate result
   result := MK(x, R_1, R_0)

   // (6) Store result in cache
   putInCache(A, B, C, result)

   // (7) Return result
   return result
```

Listing 3.2: ITE implementation

Use of a memoization cache reduces the number of ITE calls from exponential complexity to polynomial complexity in the number of nodes in A, B and C.

Efficient use of the computed table can be improved by executing the following normalization rules in the presented order to form *standard triples* [11]. These rewrite rules follow from logical equivalences and they select one of the alternative equivalent computations.

$$\mathtt{ITE}(A, A, C) \rightarrow \mathtt{ITE}(A, 1, C) \tag{3.1}$$
$$\mathtt{ITE}(A, \neg A, C) \rightarrow \mathtt{ITE}(A, 0, C) \tag{3.2}$$
$$\mathtt{ITE}(A, B, A) \rightarrow \mathtt{ITE}(A, B, 0) \tag{3.3}$$
$$\mathtt{ITE}(A, B, \neg A) \rightarrow \mathtt{ITE}(A, B, 1) \tag{3.4}$$
$$\mathtt{ITE}(A, B, B) \rightarrow B \tag{3.5}$$
$$\mathtt{ITE}(1, B, C) \rightarrow B \tag{3.6}$$
$$\mathtt{ITE}(0, B, C) \rightarrow C \tag{3.7}$$
$$\mathtt{ITE}(A, 1, 0) \rightarrow A \tag{3.8}$$
$$\mathtt{ITE}(A, 0, 1) \rightarrow \neg A \tag{3.9}$$
$$\mathtt{ITE}(A, 1, C) \rightarrow \mathtt{ITE}(C, 1, A) \quad \text{when } C{<}A \tag{3.10}$$
$$\mathtt{ITE}(A, 0, C) \rightarrow \mathtt{ITE}(\neg C, 0, \neg A) \quad \text{when } C{<}A \tag{3.11}$$

$$\texttt{ITE}(A, B, 1) \to \texttt{ITE}(\neg B, \neg A, 1) \quad \text{when } B{<}A \tag{3.12}$$
$$\texttt{ITE}(A, B, 0) \to \texttt{ITE}(B, A, 0) \quad \text{when } B{<}A \tag{3.13}$$
$$\texttt{ITE}(A, B, \neg B) \to \texttt{ITE}(B, A, \neg A) \quad \text{when } B{<}A \tag{3.14}$$
$$\texttt{ITE}(A, B, C) \to \texttt{ITE}(\neg A, C, B) \quad \text{when } A \text{ is complemented} \tag{3.15}$$
$$\texttt{ITE}(A, B, C) \to \neg\texttt{ITE}(A, \neg B, \neg C) \quad \text{when } B \text{ is complemented} \tag{3.16}$$

Rules 1 to 5 simplify the tuple (A,B,C) such that none of them are equivalent to internal BDD nodes (with possibly negated edges). The exception is the case where $B = \neg C$ which cannot be simplified. Rules 5 to 9 will handle all trivial combinations that do not require further calculation. Rules 10 to 14 are rewrite rules that use an ordering $<$ to exchange A with B or C whenever this is possible, such that A is first according to $<$. A possible ordering $<$ is an ordering on the memory address of the BDD, or on the index in the BDD array. Rules 15 and 16 remove the negation on A and B, such that only C may be a negated BDD and that possibly the result of the ITE operation will be negated. Note that rule 16 also negates the result of the ITE operation.

Rules 1 to 8 are trivial to implement. Rules 9 to 16 require the calculation of negated Boolean functions, but using complement edges this is also trivial. All rules can therefore be applied in constant time.

It can be trivially demonstrated that these substitution rules are idempotent.

3.4 Model checking using **RelProd**

The basic building block for model checking algorithms is the reachability algorithm, which calculates all reachable states given an initial set of states and a transition relation. In every iteration of this algorithm, a new set of reachable states is calculated based on the set of currently reachable states and the transition relation. This algorithm continues until no new states are found, i.e., the new set of visited states is equivalent to the previous set of visited states.

A popular symbolic algorithm for computing state transitions is based on calculating the *relational product* of the set of states and the transition relation [14], where both the set of states and the transition relation are represented by Boolean functions.

The relational product is defined as $\exists X \cdot (A \wedge B)$, where A and B are Boolean functions and X is a set of variables.

The reachability algorithm is typically implemented as in Listing 3.3.

```
BDD Reachability(BDD Initial, BDD T, Set X, Set X′)
   BDD Reachable := Initial, Previous := 0
   while Reachable != Previous
      BDD Next := (∃X · (Reachable ∧ T))[X′/X]
      Previous := Reachable
      Reachable := Reachable ∨ Next
   return Reachable
```

Listing 3.3: Basic reachability algorithm

Given a function $V(X)$ representing a set of states and a function $T(X, X')$ representing the transition relation, symbolic model checking tools like NuSMV [16] calculate the set of next states $V'(X)$ by calculating $V'(X')$ first and then substituting with the corresponding variables in X:

$$V'(X') = \exists X\big(V(X) \wedge T(X, X')\big)$$
$$V'(X) = V'(X')[X'/X]$$

The relational product can be used to calculate the successors because $V(X) \wedge T(X, X')$ essentially limits the transition relation to transitions from states in the set represented by $V(X)$. These states are then abstracted from the result using $\exists X$ so only the successors remain. However, the result uses variables in the set X' and all variables must be substituted to get $V'(X)$ instead of $V'(X')$. For example, if $V(X) = \overline{x_1}$ and $T(X, X') = \overline{x_1} x_2 x'_1 x'_2$, then $V'(X') = x'_1 x'_2$

and $V'(X) = x_1x_2$. That means the successors of the states with $s(x_1) = \texttt{false}$ are the states with both $s(x_1) = \texttt{true}$ and $s(x_2) = \texttt{true}$.

Definition 3.5 (`RelProd` operation)**.** Let $\texttt{RP}_{x=v}$ be shorthand for $\texttt{RelProd}(A_{x=v}, B_{x=v}, X)$ with $v \in \{0,1\}$ and let x be the top variable of A and B. Then `RelProd` is defined as follows:

$$\texttt{RelProd}(A,B,X) = \begin{cases} 1 & A = 1 \wedge B = 1 \\ 0 & A = 0 \vee B = 0 \\ \texttt{ITE}(\texttt{RP}_{x=0}, 1, \texttt{RP}_{x=1}) & x \in X \\ \texttt{MK}(x, \texttt{RP}_{x=1}, \texttt{RP}_{x=0}) & \text{otherwise} \end{cases}$$

Theorem 3.6. *$\texttt{RelProd}(A,B,X)$ obeys the specification $\exists X \cdot (A \wedge B)$.*

Proof. By induction on the size of A and B. We assume the induction hypothesis $\texttt{RP}_{x=v} \equiv \exists X \cdot (A_{x=v} \wedge B_{x=v})$. There are four cases:

1) If $A = 1 \wedge B = 1$, then $1 = \exists X \cdot 1 = \exists X \cdot (A \wedge B)$

2) If $A = 0 \vee B = 0$, then $0 = \exists X \cdot 0 = \exists X \cdot (A \wedge B)$

3) If $x \in X$, then

$$\begin{aligned} \texttt{ITE}(\texttt{RP}_{x=0}, 1, \texttt{RP}_{x=1}) &\overset{1}{=} (\texttt{RP}_{x=0} \wedge 1) \vee (\overline{\texttt{RP}_{x=0}} \wedge \texttt{RP}_{x=1}) \\ &\overset{2}{=} \texttt{RP}_{x=0} \vee \texttt{RP}_{x=1} \\ &\overset{3}{=} \Big(\exists X \cdot (A_{x=0} \wedge B_{x=0})\Big) \vee \Big(\exists X \cdot (A_{x=1} \wedge B_{x=1})\Big) \\ &\overset{4}{=} \Big(\exists X \cdot (A \wedge B)_{x=0}\Big) \vee \Big(\exists X \cdot (A \wedge B)_{x=1}\Big) \\ &\overset{5}{=} \exists X \cdot \big((A \wedge B)_{x=0} \vee (A \wedge B)_{x=1}\big) \\ &\overset{6}{=} \exists X \exists x \cdot (A \wedge B) \\ &\overset{7}{=} \exists X \cdot (A \wedge B) \end{aligned}$$

Step 1: by definition. Step 2: by logical equivalence. Step 3: by induction hypothesis. Step 4: by distribution of restriction over $\wedge$. Step 5: by distribution of $\exists$ over $\vee$. Step 6: by definition of $\exists$. Step 7: $x \in X$.

4) If $x \notin X$ (otherwise), then

$$\begin{aligned} \texttt{MK}(x, \texttt{RP}_{x=1}, \texttt{RP}_{x=0}) &\overset{1}{=} (x \wedge \texttt{RP}_{x=1}) \vee (\overline{x} \wedge \texttt{RP}_{x=0}) \\ &\overset{2}{=} \big(x \wedge \exists X \cdot (A_{x=1} \wedge B_{x=1})\big) \vee \big(\overline{x} \wedge \exists X \cdot (A_{x=0} \wedge B_{x=0})\big) \\ &\overset{3}{=} \big(x \wedge \exists X \cdot (A \wedge B)_{x=1}\big) \vee \big(\overline{x} \wedge \exists X \cdot (A \wedge B)_{x=0}\big) \\ &\overset{4}{=} \big(x \wedge (\exists X \cdot (A \wedge B))_{x=1}\big) \vee \big(\overline{x} \wedge (\exists X \cdot (A \wedge B))_{x=0}\big) \\ &\overset{5}{=} \exists X \cdot (A \wedge B) \end{aligned}$$

Step 1: by definition. Step 2: by induction hypothesis. Step 3: by distribution of restriction over $\wedge$. Step 4: by commutivity of restriction and $\exists$, since $x \notin X$. Step 5: by Shannon decomposition.

Furthermore, since x is the top variable of A and B, always at least one of the parameters of $\texttt{RP}_{x=v}$ is smaller than the original. $\square$

Theorem 3.7. *The result of $\texttt{RelProd}(A,B,X)$ is a reduced ordered BDD according to $<$ if A and B are reduced ordered BDDs according to $<$.*

Proof. This is automatically true in case 1 and case 2. In case 3, the result of `ITE` is a reduced ordered BDD according to $<$, which we proved earlier, since A and B are reduced ordered BDDs according to the same $<$. In case 4, the result of `MK` is only a reduced ordered BDD according to

$<$ if x is less than all variables in $\mathtt{RP}_{x=1}$ and $\mathtt{RP}_{x=0}$ and if $\mathtt{RP}_{x=1}$ and $\mathtt{RP}_{x=0}$ are ordered according to $<$. $A_{x=v}$ and $B_{x=v}$ can only contain variables greater than X since they are represented by ordered BDDs and x is the top variable of A and B. Since A and B are ordered BDDs, $A_{x=v}$ and $B_{x=v}$ are also ordered BDDs. Therefore, x is less than all variables in $\mathtt{RP}_{x=1}$ and $\mathtt{RP}_{x=0}$ and the result of `RelProd` is a reduced ordered BDD. □

Implementation The `RelProd` implementation is given in Listing 3.4.

```
BDD RelProd(BDD A, BDD B, X)
    // (1) Terminating cases
    if A = 1 ∧ B = 1 then return 1
    if A = 0 ∨ B = 0 then return 0

    // (2) Check cache
    if inCache(A, B, X) then return result

    // (3) Calculate top variable and cofactors
    x := topVariable(x_A, x_B)
    A_0 := cofactor0(A, x) B_0 := cofactor0(B, x)
    A_1 := cofactor1(A, x) B_1 := cofactor1(B, x)

    if x ∈ X
        // (4) Calculate subproblems and result when x ∈ X
        R_0 := RelProd(A_0, B_0, X)
        if R_0 = 1 then result := 1 // Because 1 ∨ R_1 = 1
        else
            R_1 := RelProd(A_1, B_1, X)
            result := ITE(R_0, 1, R_1) // Calculate R_0 ∨ R_1
    else
        // (5) Calculate subproblems and result when x ∉ X
        R_0 := RelProd(A_0, B_0, X)
        R_1 := RelProd(A_1, B_1, X)
        result := MK(x, R_1, R_0)

    // (6) Store result in cache
    putInCache(A, B, X, result)

    // (7) Return result
    return result
```

Listing 3.4: `RelProd` implementation

When $x \in X$ and $\mathtt{RP}_{x=0} = 1$ it is not necessary to calculate $\mathtt{RP}_{x=1}$. Alternatively, $\mathtt{RP}_{x=1}$ could be checked first and when it is 1, calculating $\mathtt{RP}_{x=0}$ could be skipped.

Similar to the normalization rules for `ITE` we can define the following normalization rules for `RelProd`:

$$\mathtt{RelProd}(1, 1, X) \to 1 \tag{3.1}$$
$$\mathtt{RelProd}(A, \neg A, X) \to 0 \tag{3.2}$$
$$\mathtt{RelProd}(A, 0, X) \to 0 \tag{3.3}$$
$$\mathtt{RelProd}(0, A, X) \to 0 \tag{3.4}$$
$$\mathtt{RelProd}(A, A, X) \to \mathtt{RelProd}(1, A, X) \tag{3.5}$$
$$\mathtt{RelProd}(A, 1, X) \to \mathtt{RelProd}(1, A, X) \tag{3.6}$$
$$\mathtt{RelProd}(A, B, X) \to \mathtt{RelProd}(B, A, X) \qquad \text{when } B<A \tag{3.7}$$

These normalization rules are trivial to implement. Especially rules 5 through 7 are important to improve the usefulness of the memoization cache.

3.5 Reducing redundancy using `RelProdS`

In this section we present a new algorithm that calculates the relational product with substitution.

As mentioned above, current model checkers, for example NuSMV, first calculate the relational product and then calculate the substitution of the result. The relational product operation is insufficient to generate the set of next states, because the result is defined on 'next state' variables X' instead of 'state' variables X. When calculating a substitution, new nodes are created for two reasons. First, modifying existing nodes is potentially harmful due to the sharing of subgraphs with other BDDs. Secondly, modifying existing nodes requires updating the Unique Table, so it is easier to create a new node, especially when nodes are stored inside the Unique Table rather than separately. The consequence is that two BDDs are created: one representing the set of new states defined on the X' variables and one representing the set of new states defined on the X variables.

We present a new algorithm that combines the relational product and substitution, eliminating the unnecessary creation of the BDD defined on the X' variables. Comparing our new algorithm to the old algorithm is future work.

Let A and B be BDDs representing Boolean functions, X be a set of variables and $[S]$ be a substitution. Then the specification of `RelProdS` is:

$$\texttt{RelProdS}(A, B, X, S) \equiv \Big(\exists X \cdot (A \wedge B)\Big)[S].$$

Definition 3.8 (`RelProdS` operation)**.** Let $\texttt{RPS}_{x=v}$ be shorthand for $\texttt{RelProdS}(A_{x=v}, B_{x=v}, X, S)$ with $v \in \{0, 1\}$ and let x be the top variable of A and B. Then `RelProdS` is defined as follows:

$$\texttt{RelProdS}(A, B, X, S) = \begin{cases} 1 & A = 1 \wedge B = 1 \\ 0 & A = 0 \vee B = 0 \\ \texttt{ITE}(\texttt{RPS}_{x=0}, 1, \texttt{RPS}_{x=1}) & x \in X \\ \texttt{ITE}(S(x), \texttt{RPS}_{x=1}, \texttt{RPS}_{x=0}) & \text{otherwise} \end{cases}$$

Theorem 3.9. *$\texttt{RelProdS}(A, B, X, S)$ obeys the specification $\big(\exists X \cdot (A \wedge B)\big)[S]$.*

Proof. By induction on the size of A and B. We assume the induction hypothesis $\texttt{RPS}_{x=v} \equiv \big(\exists X \cdot (A_{x=v} \wedge B_{x=v})\big)[S] \equiv \big(\exists X \cdot (A \wedge B)_{x=v}\big)[S]$ (by distribution of restriction). There are four cases:

1) If $A = 1 \wedge B = 1$, then $1 = (\exists X \cdot 1)[S] = \big(\exists X \cdot (A \wedge B)\big)[S]$.

2) If $A = 0 \vee B = 0$, then $0 = (\exists X \cdot 0)[S] = \big(\exists X \cdot (A \wedge B)\big)[S]$.

3) If $x \in X$ then:

$$\begin{aligned} \texttt{ITE}(\texttt{RPS}_{x=0}, 1, \texttt{RPS}_{x=1}) &\overset{1}{=} (\texttt{RPS}_{x=0} \wedge 1) \vee (\overline{\texttt{RPS}_{x=0}} \wedge \texttt{RPS}_{x=1}) \\ &\overset{2}{=} \texttt{RPS}_{x=0} \vee \texttt{RPS}_{x=1} \\ &\overset{3}{=} \Big(\exists X \cdot (A_{x=0} \wedge B_{x=0})\Big)[S] \vee \Big(\exists X \cdot (A_{x=1} \wedge B_{x=1})\Big)[S] \\ &\overset{4}{=} \Big(\exists X \cdot (A \wedge B)_{x=0}\Big)[S] \vee \Big(\exists X \cdot (A \wedge B)_{x=1}\Big)[S] \\ &\overset{5}{=} \Big(\big(\exists X \cdot (A \wedge B)_{x=0}\big) \vee \big(\exists X \cdot (A \wedge B)_{x=1}\big)\Big)[S] \\ &\overset{6}{=} \Big(\exists X \cdot \big((A \wedge B)_{x=0} \vee (A \wedge B)_{x=1}\big)\Big)[S] \\ &\overset{7}{=} \Big(\exists X \exists x \cdot (A \wedge B)\Big)[S] \\ &\overset{8}{=} \Big(\exists X \cdot (A \wedge B)\Big)[S] \end{aligned}$$

Step 1: by definition. Step 2: logical equivalence. Step 3: by induction hypothesis. Step 4: by distribution of restriction over $\wedge$. Step 5: by distribution of substitution over $\vee$. Step 6: by distribution of $\exists$ over $\vee$. Step 7: by definition of $\exists$. Step 8: $x \in X$.

4) If $x \notin X$ then:

$$\begin{aligned}
\texttt{ITE}(S(x), \texttt{RPS}_{x=1}, \texttt{RPS}_{x=0}) &\overset{1}{=} (S(x) \wedge \texttt{RPS}_{x=1}) \vee (\overline{S(x)} \wedge \texttt{RPS}_{x=0}) \\
&\overset{2}{=} (S(x) \wedge \texttt{RPS}_{x=1}) \vee (S(\overline{x}) \wedge \texttt{RPS}_{x=0}) \\
&\overset{3}{=} \Big(S(x) \wedge \big(\exists X \cdot (A_{x=1} \wedge B_{x=1})\big)[S]\Big) \vee \\
&\quad \Big(S(\overline{x}) \wedge \big(\exists X \cdot (A_{x=0} \wedge B_{x=0})\big)[S]\Big) \\
&\overset{4}{=} \Big(S(x) \wedge \big(\exists X \cdot (A \wedge B)_{x=1}\big)[S]\Big) \vee \\
&\quad \Big(S(\overline{x}) \wedge \big(\exists X \cdot (A \wedge B)_{x=0}\big)[S]\Big) \\
&\overset{5}{=} \Big(S(x) \wedge \big(\exists X \cdot (A \wedge B)\big)_{x=1}[S]\Big) \vee \\
&\quad \Big(S(\overline{x}) \wedge \big(\exists X \cdot (A \wedge B)\big)_{x=0}[S]\Big) \\
&\overset{6}{=} \Big(\big(x \wedge \big(\exists X \cdot (A \wedge B)\big)_{x=1}\big) \vee \big(\overline{x} \wedge \big(\exists X \cdot (A \wedge B)\big)_{x=0}\big)\Big)[S] \\
&\overset{7}{=} \big(\exists X \cdot (A \wedge B)\big)[S]
\end{aligned}$$

Step 1: by definition. Step 2: by commutivity of substitution and negation. Step 3: by induction hypothesis. Step 4: by distribution of restriction over $\wedge$. Step 5: by commutivity of restriction and $\exists$, since $x \notin X$. Step 6: by distribution of substitution over $\vee$. Step 7: by Shannon decomposition.

Furthermore, since x is the top variable of A and B, always at least one of the parameters of $\texttt{RPS}_{x=v}$ is smaller than the original. □

Theorem 3.10. *The result of* `RelProdS`(A, B, X, S) *is a reduced ordered BDD according to* $<$ *if* A *and* B *are reduced ordered BDDs according to* $<$.

Proof. In case 1 and case 2, this is automatically true. In case 3 and case 4, the result of `ITE` is a reduced ordered BDD, which we proved earlier, since A and B are reduced ordered BDDs according to $<$, therefore $A_{x=v}$ and $B_{x=v}$ are reduced ordered BDDs according to $<$. □

Variation The above algorithm can be modified for specific cases. If variables in X and corresponding (according to S) variables in X' are in the same order according to $<$, for example by pairing them, we can replace `ITE` in case 4 by $\texttt{MK}(S(x), \texttt{RPS}_{x=1}, \texttt{RPS}_{x=0})$. It can easily be proven that the result is still a reduced ordered BDD since $\texttt{RPS}_{x=1}$ and $\texttt{RPS}_{x=0}$ can only contain variables in X' that are greater than $S(x)$ according to $<$.

Implementation The `RelProdS` implementation is given in Listing 3.5.

As with `RelProd`, when $x \in X$ and $\texttt{RPS}_{x=0} = 1$ it is not necessary to calculate $\texttt{RPS}_{x=1}$. Alternatively, $\texttt{RPS}_{x=1}$ could be checked first and when it is 1, calculating $\texttt{RPS}_{x=0}$ could be skipped.

For `RelProdS` we can use the same normalization rules as for `RelProd`.

3.6 Reversed `RelProdS`

In model checking it is often necessary to calculate the set of previous states, that is, calculate the set of all states that have a transition to states in S. One of the applications is checking for states that do not have successors, also called *deadlocks*. We can calculate which states of a set S do not have successors, by calculating which states do have successors:

$$S_{\text{deadlock}} = S \setminus \texttt{predecessors}(\texttt{successors}(S)).$$

The successors of a set of states represented by the Boolean function F can be calculated using `RelProdS`$(F, T, X, [X/X'])$, where T is the Boolean function representing the transition relation. We can calculate the predecessor of a set of states using `RelProdS` and the inverse of

```
BDD RelProdS(BDD A, BDD B, Set X, Substitution S)
   // (1) Terminating cases
   if A = 1 ∧ B = 1 then return 1
   if A = 0 ∨ B = 0 then return 0

   // (2) Check cache
   if inCache(A, B, X, S) then return result

   // (3) Calculate top variable and cofactors
   x := topVariable(x_A, x_B)
   A_0 := cofactor0(A, x) B_0 := cofactor0(B, x)
   A_1 := cofactor1(A, x) B_1 := cofactor1(B, x)

   if x ∈ X
      // (4) Calculate subproblems and result when x ∈ X
      R_0 := RelProdS(A_0, B_0, X, S)
      if R_0 = 1 then result := 1 // Because 1 ∨ R_1 = 1
      else
         R_1 := RelProdS(A_1, B_1, X, S)
         result := ITE(R_0, 1, R_1) // Calculate R_0 ∨ R_1
   else
      // (5) Calculate subproblems and result when x ∉ X
      R_0 := RelProdS(A_0, B_0, X, S)
      R_1 := RelProdS(A_1, B_1, X, S)
      result := ITE(S(x), R_1, R_0)

   // (6) Store result in cache
   putInCache(A, B, X, S, result)

   // (7) Return result
   return result
```

Listing 3.5: `RelProdS` implementation

T, but then the inverse of T must be calculated, increasing memory usages. Alternatively, the predecessors can be calculated directly.

We present the dual of `RelProdS` which can be used to obtain this set of predecessors. Let X and X' be disjunct sets of variables and $[S]$ be the substitution $[X/X']$ and A be a Boolean function defined on X and B be a Boolean function defined on $X \cup X'$. Then `RRelProdS` is specified as follows:

$$\texttt{RRelProdS}(A, B, X', S) = \exists X' \cdot (A[S] \wedge B).$$

Definition 3.11 (`RelProdS` operation)**.** Let S^{-1} be the inverse of S, i.e., $[X'/X]$, x_A be the top variable of A, x_B be the top variable of B, x be the top variable of $S(x_A)$ and x_B, $\texttt{RRPS}_{x=v}$ be shorthand for $\texttt{RRelProdS}(A_{S^{-1}(x)=v}, B_{x=v}, X', S)$, and $\texttt{RRPS}^B_{x=v}$ for $\texttt{RRelProdS}(A, B_{x=v}, X', S)$. Then `RRelProdS` is defined as follows:

$$\texttt{RRelProdS}(A, B, X', S) = \begin{cases} 1 & A = 1 \wedge B = 1 \\ 0 & A = 0 \vee B = 0 \\ \texttt{ITE}(\texttt{RRPS}_{x=0}, 1, \texttt{RRPS}_{x=1}) & x \in X' \\ \texttt{ITE}(x, \texttt{RRPS}^B_{x=1}, \texttt{RRPS}^B_{x=0}) & \text{otherwise } (x \in X) \end{cases}$$

Theorem 3.12. *$\texttt{RRelProdS}(A, B, X', S)$ obeys the specification $\exists X' \cdot (A[S] \wedge B)$.*

Proof. By induction on the size of A and B. We assume the induction hypotheses $\texttt{RRPS}_{x=v} \equiv \exists X' \cdot (A_{S^{-1}(x)=v}[S] \wedge B_{x=v})$ and $\texttt{RRPS}^B_{x=v} \equiv \exists X' \cdot (A[S] \wedge B_{x=v})$. There are four cases:

1) If $A = 1 \wedge B = 1$ then $1 = \exists X' \cdot 1 = \exists X' \cdot \big(A[S] \wedge B\big)$.

2) If $A = 0 \vee B = 0$ then $0 = \exists X' \cdot 0 = \exists X' \cdot \big(A[S] \wedge B\big)$.

3) If $x \in X'$ then:

$$\begin{aligned}
\texttt{ITE}(\texttt{RRPS}_{x=0}, 1, \texttt{RRPS}_{x=1}) &\overset{1}{=} (\texttt{RRPS}_{x=0} \wedge 1) \vee (\overline{\texttt{RRPS}_{x=0}} \wedge \texttt{RRPS}_{x=1}) \\
&\overset{2}{=} \texttt{RRPS}_{x=0} \vee \texttt{RRPS}_{x=1} \\
&\overset{3}{=} \Big(\exists X' \cdot \big(A_{S^{-1}(x)=0}[S] \wedge B_{x=0}\big)\Big) \vee \\
&\quad \Big(\exists X' \cdot \big(A_{S^{-1}(x)=1}[S] \wedge B_{x=1}\big)\Big) \\
&\overset{4}{=} \Big(\exists X' \cdot \big(A[S]_{x=0} \wedge B_{x=0}\big)\Big) \vee \Big(\exists X' \cdot \big(A[S]_{x=1} \wedge B_{x=1}\big)\Big) \\
&\overset{5}{=} \Big(\exists X' \cdot \big(A[S] \wedge B\big)_{x=0}\Big) \vee \Big(\exists X' \cdot \big(A[S] \wedge B\big)_{x=1}\Big) \\
&\overset{6}{=} \exists X' \cdot \Big(\big(A[S] \wedge B\big)_{x=0} \vee \big(A[S] \wedge B\big)_{x=1}\Big) \\
&\overset{7}{=} \exists X' \exists x \cdot \big(A[S] \wedge B\big) \\
&\overset{8}{=} \exists X' \cdot (A[S] \wedge B)
\end{aligned}$$

Step 1: by definition. Step 2: by logical equivalence. Step 3: by induction hypothesis. Step 4: $S(S^{-1}(x)) = x$ since $x \in X'$ (If $x \in X$ then $S^{-1}(x) = x$ and $S(S^{-1}(x)) \neq x$). Step 5: by distribution of restriction over $\wedge$. Step 6: by distribution of $\exists$ over $\vee$. Step 7: by definition of $\exists$. Step 8: $x \in X'$.

4) If $x \in X$ then:

$$\begin{aligned}
\texttt{ITE}(x, \texttt{RRPS}^B_{x=1}, \texttt{RRPS}^B_{x=0}) &\overset{1}{=} (x \wedge \texttt{RRPS}^B_{x=1}) \vee (\overline{x} \wedge \texttt{RRPS}^B_{x=0}) \\
&\overset{2}{=} (x \wedge \exists X' \cdot (A[S] \wedge B_{x=1})) \vee (\overline{x} \wedge \exists X' \cdot (A[S] \wedge B_{x=0})) \\
&\overset{3}{=} (x \wedge \exists X' \cdot (A[S]_{x=1} \wedge B_{x=1})) \vee (\overline{x} \wedge \exists X' \cdot (A[S]_{x=0} \wedge B_{x=0})) \\
&\overset{4}{=} (x \wedge \exists X' \cdot (A[S] \wedge B)_{x=1}) \vee (\overline{x} \wedge \exists X' \cdot (A[S] \wedge B)_{x=0}) \\
&\overset{5}{=} (x \wedge (\exists X' \cdot (A[S] \wedge B))_{x=1}) \vee (\overline{x} \wedge (\exists X' \cdot (A[S] \wedge B))_{x=0}) \\
&\overset{6}{=} \exists X' \cdot (A[S] \wedge B)
\end{aligned}$$

Step 1: by definition. Step 2: by induction hypothesis. Step 3: $x \in X$, therefore x is not a variable in $A[S]$, therefore $A[S]_{x=v} = A[S]$. Step 4: by distribution of restriction over $\wedge$. Step 5: by commutivity of restriction and $\exists$, since $x \notin X'$. Step 6: by Shannon decomposition.

Furthermore, since x is the top variable of A and B, always at least one of the parameters of $\texttt{RPS}_{x=v}$ is smaller than the original. □

Theorem 3.13. *The result of* $\texttt{RRelProdS}(A, B, X', S)$ *is a reduced ordered BDD according to* $<$ *if* A *and* B *are reduced ordered BDDs according to* $<$.

Proof. In case 1 and case 2, this is automatically true. In case 3 and case 4, the result of `ITE` is a reduced ordered BDD, which we proved earlier, since A and B are reduced ordered BDDs according to $<$, therefore $A_{x=v}$ and $B_{x=v}$ are reduced ordered BDDs according to $<$. □

Implementation : The `RRelProdS` implementation is given in Listing 3.6.

As with `RelProd` and `RelProdS`, when $x \in X$ and $\texttt{RRPS}_{x=0} = 1$ it is not necessary to calculate $\texttt{RRPS}_{x=1}$. Alternatively, $\texttt{RRPS}_{x=1}$ could be checked first and when it is 1, calculating $\texttt{RRPS}_{x=0}$ can be skipped.

For `RRelProdS` we can use the same standard triples as for `RelProd` and `RelProdS`.

```
BDD RRelProdS(BDD A, BDD B, Set X', Substitution S)
   // (1) Terminating cases
   if A = 1 ∧ B = 1 then return 1
   if A = 0 ∨ B = 0 then return 0

   // (2) Check cache
   if inCache(A,B,X',S) then return result

   // (3) Calculate top variable and cofactors
   x := topVariable(S(x_A), x_B)
   A_0 := cofactor0(A, S^-1(x)) B_0 := cofactor0(B, x)
   A_1 := cofactor1(A, S^-1(x)) B_1 := cofactor1(B, x)

   if x ∈ X'
      // (4) Calculate subproblems and result when x ∈ X'
      R_0 := RRelProdS(A_0, B_0, X', S)
      if R_0 = 1 then result := 1
      else
         R_1 := RRelProdS(A_1, B_1, X', S)
         result := ITE(R_0, 1, R_1)
   else
      // (5) Calculate subproblems and result when x ∈ X
      R_0 := RRelProdS(A, B_0, X', S)
      R_1 := RRelProdS(A, B_1, X', S)
      result := MK(x, R_0, R_1)

   // (6) Store result in cache
   putInCache(A,B,X',S,result)

   // (7) Resturn result
   return result
```

Listing 3.6: `RRelProdS` implementation

3.7 Conclusions

In this chapter, we discussed how BDDs are implemented. We also presented a new algorithm, `RelProdS`, that removes the creation of unnecessary BDD nodes in the reachability algorithm. We also proved that the algorithms `ITE`, `RelProd`, `RelProdS` and `RRelProdS` are correct and result in a ordered, reduced BDDs. We implement the `RelProdS` and `ITE` algorithms for our experiments. Experiments that show that using `RelProdS` are more efficient are future work.

4

Preliminaries of parallelism

The current chapter discusses the need to parallelize algorithms in order to keep improving performance. We present a number of existing theoretical approaches to model multicore performance. We also present our extension to Amdahl's Law to understand more factors determining parallel performance.

4.1 Parallelism

In model checking, there is always a need to be able to process larger models in less time. To do that, more calculations have to be performed faster. Until the last decade, the usual approach for better performance was to increase CPU frequencies. Algorithms were optimized for a single processor and processors implemented various hardware optimizations, such as out-of-order execution and pipelining.

In 2005, Sutter published the famous article *The free lunch is over* [45] stating that processor speeds are reaching a physical limit and that software developers will have to develop programs that are concurrent rather than sequential.

4.2 Architectures

There are many different architectures that allow parallelism. For example, there are multicore architectures with a hierarchy of memories and many-core systems with hundred of cores and different types of local memory. There is a distinction between homogeneous and heteregeneous architectures. In homogeneous architectures the processor cores are similar, while in heterogeneous architectures the processor cores are specialized. Some architectures have a caching hierarchy with a shared bus, others have blocks of memory for every processor core and communication networks between the processor cores.

In our research, we will focus on multicore architectures, with multiple identical processor cores and a hierarchy of memories: a private L_1 cache for every core, a shared L_2 cache and main memory. In some multicore systems, there are multiple multiprocessors, where each multiprocessor has identical processor cores. In some multicore systems, the L_2 cache is shared by some cores and there is a L_3 cache shared by all cores.

There is a disctinction between Symmetric Multi-Processing (SMP) and Non-Uniform Memory Access (NUMA) architectures. In SMP architectures all memory access is performed using the same shared memory bus. In NUMA architectures it takes longer to access certain regions of memory than others, since some regions of memory are on different buses than others. NUMA alleviates the bottleneck of one shared memory bus by limiting the number of CPUs on any single memory bus [1]. Our focus will be on NUMA architectures, since we perform our experiments on a machine with 48 cores.

4.3 Approaches to parallelization

We will use the following terminology: An *algorithm* consists of a number of *operations*, which can be decomposed into small *tasks* or *suboperations*. Tasks are organized according to their dependencies: most tasks require other tasks to be finished in order to progress. This can be visualised in a *task dependency graph*. See also Figure 4.1.

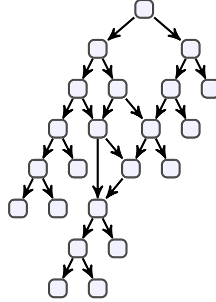


Figure 4.1: Typical task dependency graph

An algorithm is parallelized by dividing the algorithm into smaller parts that can be executed independently by multiple *workers*. The number of available workers depends on the implementation and on the hardware. In general, the number of workers is equal to the number of available hardware threads, which is often equal to the number of available processor cores. The *speedup* is a measure for the performance gain of parallelizing an algorithm. If an algorithm with 20 workers is executed 5 times as fast as with 1 worker, we say the speedup for 20 workers is 5. The ideal speedup in that case would be 20. In this example, the efficiency is 25%.

We distinguish a low-level approach to parallelization and a high-level approach. The low-level approach is where the parallelization occurs at task-level. The high-level approach is where the parallelization occurs at operation-level.

For example, the reachability algorithm consists of a number of Set operations, implemented using BDDs. The implementation of these operations can be decomposed into small tasks. Using the high-level approach we can execute operations in parallel, if some operations are independent. In the normal reachability algorithm all operations depend on the previous operation, but in the reachability algorithm of `LTSmin` (see Section 6.1) all `RelProdS` operations of one level could be executed in parallel. Using the low-level approach we could use a load balancing algorithm to distribute tasks over workers. This is the approach we use in our research, partially since the high-level approach has limited usefulness for model checking.

An advantage of high-level parallelization is that sequential optimizations in the implementation of operations can often still be exploited. A disdvantage of high-level parallelization is that it may be harder to increase the number of workers, because the number of operations that can be executed simultaneously may be very limited. An advantage of low-level parallelization is that the high-level algorithm need not be aware of the parallelization. Assuming that data structures are handled on the lower level, high-level algorithms do not need to be modified at all. A disadvantage of low-level parallelization is that they may be more fine-grained: the low-level algorithm may scale worse because the size of each job may be very small, increasing relative overhead due to parallelism. In addition, thread safety may introduce extra overhead, for example lock contention when using mutual exclusion, or false sharing of cache lines and atomic restarts when using lockless communication. See also Section 4.4.3.

4.4 Theoretical models of multicore parallelism

When parallelizing an algorithm, the performance speedup is often less than ideal. See also Figure 4.2. Ideally the speedup is identical to the number of workers. Amdahl's Law (see below) already puts an upperbound to the possible speedup. In practice, there is a point where adding workers decreases the speedup, due to communication costs and memory bandwidth limitations increasing faster than the extra performance. In order to understand the factors

that influence the performance of concurrent programs on multicore machines, there are some theoretical approaches.

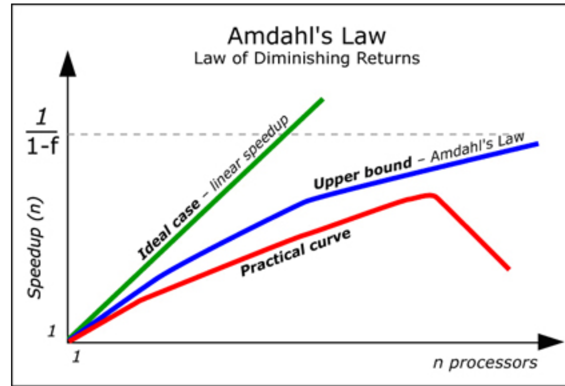


Figure 4.2: Typical speedup graph

4.4.1 Amdahl's Law

One famous approach is Amdahl's Law, which states that the maximum theoretical speedup by increasing the number of workers W is limited by the sequential part of a program [6].

Let α be the fraction of the program that cannot be executed in parallel, let T_S be the time required to execute the entire program sequentially and let T_W be the time required to execute the program using W workers.

Then the speedup $S(W)$ is defined as

$$S(W) = \frac{T_S}{T_W} = \frac{T_S}{\alpha T_S + (1-\alpha)T_S/W} = \frac{1}{\alpha + (1-\alpha)/W}$$

with a theoretical upperbound ($W \to \infty$) of $\frac{1}{\alpha}$. This means that even with an infinite amount of workers, theoretical speedups are limited.

4.4.2 Extending Amdahl's Law

Amdahl's Law is a simplification just to show that there is an upperbound to possible speedups. The model assumes that with an infinite number of workers calculating the parallel part will cost no time. It also assumes that there is no additional concurrency overhead. To be able to understand some variables determining parallel performance, we extend Amdahl's Law.

There is a lower bound to the time consumed by the parallel part. This is the *critical path*, the longest path from root to leaf in the task dependency graph, where the length of a path is the sum of the duration of every task on that path.

In addition, there is an overhead mainly due to communication costs (lock contention, false sharing, atomic restarts, work distribution). These costs often increase monotonically with the number of workers. For example, if an algorithm has to communicate with all workers, adding workers will increase the amount of communication quadratically.

Let β be the size of the critical path relative to T_S, limiting $(1-\alpha)/W$. Let $C(W)$ be the overhead with W workers relative to T_S. Then we get for T_W the equation

$$T_W = \alpha T_S + \max\big(\beta, (1-\alpha)/W\big)T_S + C(W)T_S.$$

For the speedup $S(W)$ we get

$$S(W) = \frac{1}{\alpha + \max\big(\beta, (1-\alpha)/W\big) + C(W)}.$$

Assuming $C(W)$ is monotonically increasing, we can calculate the number of workers to get the best speedup:

$$W_{\max} = \lfloor (1-\alpha)/\beta \rfloor.$$

Therefore, if $C(W)$ monotonically increases, there is a theoretical upperbound of

$$S_{\max} = \frac{1}{\alpha + \beta + C(\lfloor (1-\alpha)/\beta \rfloor)}.$$

This model suggests that there is a number of workers, beyond which no performance gains are possible, given certain values of α, β and $C(W)$.

To increase speedup and upperbound, α (sequential fraction) and β (critical path fraction) should be as low as possible. In some cases, it may be possible to decrease the absolute size of the sequential part at the cost of more work and still get a better speedup relative to the program. In all cases, even problems with very low values for α and β are limited by overhead $C(W)$. The speedup will already decrease when $C(W)$ increases more than $(1-\alpha)/W$ decreases.

Values of α and β also depend on the input data. For example, more data may result in a lower α, without affecting β. This implies that in these cases using a larger input data will improve the speedup of the algorithm. Gustafson showed that the speedup $S(W)$ could be increased by increasing the size of the problems [23]. Since more work could be done in the same amount of time when using more workers, α would be lower and the speedup would be much higher.

Grouping tasks together also influences these variables. This essentially increases the size of tasks and decreases the number of tasks. This results in a decreased $C(W)$, since less communication is necessary to manage tasks, while increasing the length of the critical path β, due to the larger duration of each task.

4.4.3 Communication bottlenecks

A number of papers have been published discussing bottlenecks in multicore architectures. The most relevant models are the Roofline model of Williams et al. [47] and the computational model of Bader et al. [7].

The Roofline model is based on the concept of *operational intensity*, which is defined as the number of operations per byte of DRAM (main memory) traffic. The execution of a program is limited by the processor performance and the memory bandwidth (see Figure 4.3). Depending on the operational intensity either the processor performance or the memory bandwidth is the limiting factor. The Roofline model is used to determine the highest possible performance of a system.

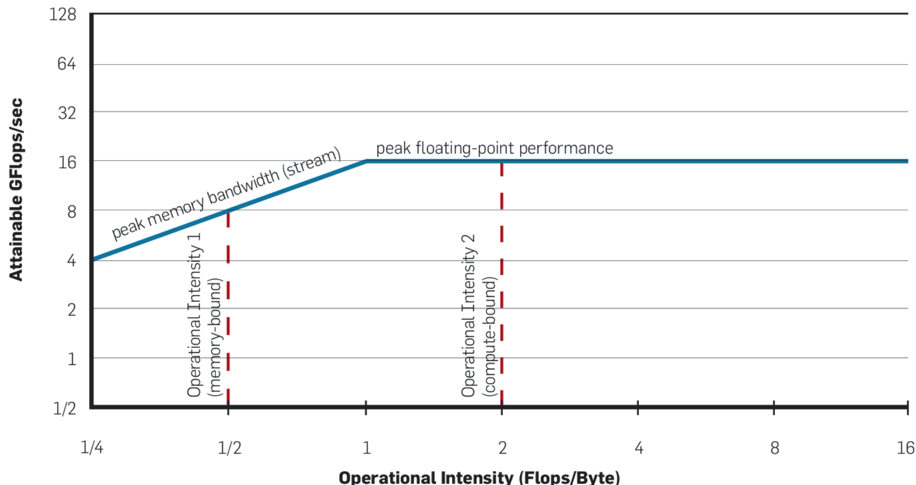


Figure 4.3: Roofline model

Bader et al. suggest that the complexity of a parallel algorithm depends not only on the complexity of the algorithm, but also on the bandwidth between the processor caches and main memory and on the time required for synchronization. Performance on multicore systems is mainly affected by the number of workers, caching, memory bandwidth and synchronization.

Traffic between processor caches and main memory occurs in blocks called *cache lines*. On most modern processors, cache lines are blocks of 64 bytes. Whenever data is requested from main memory to a processor cache, all cache lines that contain this data are transferred. When one of the processor cores writes to a memory address, all processor caches that contain the same cache line must be updated.

Efficient use of cache lines is important to reduce the impact of limited memory bandwidth. We note the following effects in particular:

1) *Locality* of data can improve performance. When data is accessed, the entire cache line is transferred. When different data on the same cache line is accessed, no additional memory transfers are neccessary since it will already be available in the processor cache. Reusing cache lines, such as the program stack, also decreases memory bandwidth usage, especially when these cache lines are not shared by other processor cores and caches.

2) *False sharing* occurs when multiple processor cores access different data that share the same cache line. When one worker writes data, all caches must be updated. If this could be avoided by storing data on different cache lines, the unnecessary transfer is called false sharing.

3) *Alignment* of data is important as well. If data is improperly aligned, i.e. a block of data is stored on more cache lines than is necessary, then there are also more data transfers than is necessary.

4) *Atomic restarts* occur when using atomic processor instructions like `compare_and_swap` to modify data. When atomic operations fail algorithm usually restart the operation, which is called an atomic restart.

5) *Lock contention* is an effect due to using mutual exclusion in communication between workers. When a worker attempt to get a lock held by another worker, it has to wait until the other worker releases the lock. This increases the communication overhead.

6) *Limited memory bandwidth* increases the average time required to perform memory transfers.

We model these effects as increases in $C(W)$. These effects all limit the possible speedup. False sharing and bad alignments also reduce the performance when only using one worker, but they also cause the bandwidth limit to be reached using less workers. Increasing the operational intensity or decreasing bandwidth requirements, by reducing false sharing, increasing locality and considering data alignment, speedups that are limited by memory bandwidth can be improved.

4.5 Conclusions

In conclusion, from a theoretical perspective, speedups are limited by the sequential part of the system, by the length of the critical path, by increasing communication costs and by limited memory bandwidth. Note that increasing communication costs from atomic restarts and lock contention and limited memory bandwidth are two different but related mechanisms that increase communication overhead. The above models are qualitative models providing insight in possible causes of limited speedup. In Chapter 6 we shall discuss some of these causes.

5

Implementation of parallelization

The current chapter discusses the implementation of parallelism for dynamic programming problems. We discuss an approach using Wool for task distribution and an approach based on result sharing. We also present two lockless data structures.

5.1 Task distribution and result sharing

The primary goal of parallelizing an algorithm is speedup. Ideally, all work is distributed evenly among workers and a speedup is obtained equal to the amount of workers. The problem of distributing work evenly is called *load balancing*. There are several ways to solve this problem.

The first approach is distributing tasks to workers. When a task is executed, subtasks are created. These subtasks can be assigned to workers, or added to one or more task queues. Workers either execute their assigned tasks or steal tasks from the queue(s). The result of the subtasks is stored in memory (for example, the task structure could contain a field for the result) and the task status is updated. Examples of load balancing strategies include work stealing algorithms such as Asynchronous Random Polling [4], the Scalable Locality-aware Adaptive Work-stealing Scheduler [22], and CAB, a cache-aware task-stealing algorithm designed for multi-socket multicore architectures [15].

Several frameworks have been developed to make creating task-based implementations easier. Some approaches, such as Cilk [10], OpenMP [17] and Gossamer [39] are compiler-based: they compile the program to a different form, usually by adding code, translating from an extended version of C to normal C. Others, such as SWARM [7] and Wool [19, 38] are library-based. These frameworks support creating tasks (`fork`, `spawn`) and waiting for their completion (`join`, `sync`) in order to use the results.

When parallelizing a dynamic programming problem using task distribution it is possible that identical tasks are performed multiple times (redundant work). See also Section 3.2. To reduce the amount of redundant work, a memoization cache is used. Workers share results of subtasks using this memoization cache.

The second approach to parallelizing an algorithm is not based on distributing subtasks, but purely on sharing results of subtasks. All workers calculate the same tasks and communicate results of subtasks using a memoization cache. By calculating the subtasks of a task in a random order, the chances that subtasks are calculated multiple times by different workers are reduced. For example, if a task has two subtasks, there is a 50% chance that two workers will select a different subtask to calculate first. When both tasks have been calculated, both workers share the result using the memoization cache. With a sufficiently large number of tasks and choices, the expected number of tasks per worker will approach $\frac{T}{N}$.

This approach is similar to the Swarm Verification of Holzmann et al. [26]

5.2 Implementing task distribution using Wool

To implement task distribution, we selected the Wool framework [19]. Wool is a work stealing framework especially developed for fine-grained task-based parallelism..

There are several reasons for this selection. The first reason is that according to [38] the framework offers superior scalability in fine-grained task-based parallelism. The second reason is that there is earlier work parallelizing the BDD package BuDDy using Cilk [24] and using Wool we should expect similar results. Finally, it is fairly straightforward to implement parallellism using the Wool framework. Wool is conceptually similar to Cilk, but offers slightly better performance [38].

Wool is a framework for fine-grained parallelism implemented in C. Subtasks are created using the `SPAWN` macro. Every `SPAWN` has a matching `SYNC` macro. A `CALL` macro is used when a `SPAWN` would immediately be followed by a `SYNC`.

As an example, consider the following recursive program and how it is parallelized in two different ways using Wool.

```
// Original function
fib(n)
   if n < 2 then return n
   return fib(n-1) + fib(n-2)

// Parallel task using only SPAWN and SYNC
TASK(fib1, n)
   if n < 2 then return n
   SPAWN(fib1, n-1)
   SPAWN(fib1, n-2)
   m = SYNC // matches n-2
   return m + SYNC // matches n-1

// Parallel task using SPAWN, SYNC and CALL
TASK(fib2, n)
   if n < 2 then return n
   SPAWN(fib2, n-1)
   m = CALL(fib2, n-2)
   return m + SYNC

// Function using Wool, depends on fib1 or fib2
fib(n)
   return CALL(fib2, n) // or fib1.
```

Wool is implemented using local task stacks. Every worker has a local task stack. The macro `SPAWN` adds the task to this stack, where it can be stolen by other workers. When `SYNC` is used, it will execute the task on top of the stack, unless it has been stolen. If this task has been stolen, the worker will execute other tasks until the task is complete, for example by stealing tasks from other workers. When `SYNC` is done, it returns the result value of the task that was on top of the thread-local stack. In the implementation, the `CALL` macro will actually not add tasks to the stack but immediately execute the task. This means that only actual parallelism is exposed to other workers via the task stacks.

For example, in the above example, while the first worker is calculating `fib2(n-2)`, other workers can steal `fib2(n-1)`.

An important limitation of Wool in the context of this research is that there is no support for task sharing. When a task is created, only the creator can wait for that task to complete. The framework does not expose methods to detect existing identical tasks and wait for their completion rather than starting a duplicate calculation. Using a memoization cache aleviates this problem, but cannot solve it completely. A memoization cache only prevents redundant calculations *after* the result has been calculated. If multiple workers start the same calculation, the result is not yet available in the cache. This prevents proper dynamic programming. In order to implement dynamic programming, the memoization cache could be used to store a *dummy result* and the framework must somehow offer a method to steal other work until some condition has been satisfied, e.g., the actual result is in the cache.

5.3 Implementing result sharing

We implemented result sharing using our own set of macros in C. Every worker has its own random number generator with different seeds on different cache lines, minimizing unnecessary memory transfers.

```
int fibr(int n)
   if n < 2 then return n
   if inResultCache(n) then return result
   if random() % 2
      n1 := fibr(n-1)
      n2 := fibr(n-2)
   else
      n2 := fibr(n-2)
      n1 := fibr(n-1)
   result := n1 + n2
   putResultCache(n, result)
   return result

int fib(int n)
   return fibr(n) || fibr(n) || ... || fibr(n)
```

All `fibr(n)` operations return the same result. The parallel operation `||` starts all workers and returns the result of any `fibr(n)` operation after all workers are done.

If the calculation is not trivial ($n < 2$), the memoization cache is consulted to see if the result has already been calculated. If it has not been calculated yet, based on the result of the random number generator, either `fibr(n-1)` or `fibr(n-2)` is calculated first. Finally, the end result is calculated and put in the memoization cache.

5.4 Lockless data structures

Calculating BDD operations requires two data structures: the hashtable containing all BDD nodes and the memoization cache to store results of operations, which is used both in the approach using Wool and in the approach using result sharing.

Recent research has been dedicated to developing data structures and algorithms that require no explicit locking, using atomic processor instructions like `compare_and_swap` and `fetch_and_add` for communication rather than mutexes, which can be slow. Locking can have severe consequences for the parallel performance of the algorithm. Herlihy and Shavit [25] distinguish *lock-free* algorithms, *wait-free* algorithms and *lockless* algorithms. Lock-free algorithms are algorithms without mutual exclusion that guarantee system-wide progress, i.e., always *some* worker can continue. Wait-free algorithms (also called *non-blocking* algorithms) are algorithms that guarantee per-worker progress, i.e. always *all* workers can continue. Lockless algorithms are algorithms that only statistically provide progress guarantees but avoid explicit locks by using similar techniques as used in lock-free solutions.

First we discuss our implementation of a lockless memoization cache and then we discuss our implementation of a lockless hashtable that supports garbage collection.

5.4.1 Lockless memoization cache

The lockless memoization cache is based on a hash table with two arrays, one array of hash values and one array with the corresponding data [29]. We use the same algorithm as in [29] but we only use the first cache line. The data consists of the key (a canonical representation of the parameters of each task) and the value (the result of the operation), both of a fixed size. We use the names `keylength` for the size of the key and `datalength` for the combined size of the key and the value.

When inserting a value in the cache, the hash array is consulted to see whether there already is a key with the same hash in the table. Only if the hashes match, the larger data array is read. If the target bucket is empty, then that bucket will be used to store the data. If the hashes do not match or if the keys do not match, the other buckets in the same processor cache line in the hash array as the initial bucket will be checked.

The most important requirement for the memoization cache is that there is a `put` and a `get` method and that using the `get` will never give a result that was not entered into the memoization cache using `put`.

The memoization cache is allowed to overwrite existing results, i.e., the cache is a *lossy* cache. The advantage is that lossy caches are easier to implement and faster. A disadvantage is the possibility of losing results when they are overwritten. The consequence of losing one result is that one subtask needs to be recalculated, which is not expensive for BDD operations, since the subtasks of that subtask are also stored in the memoization cache. We therefore expect that the cost of recalculating lost data is smaller than the cost of bookkeeping whether results must be kept or can be discarded.

In addition, since we use garbage collection with reference counting on the BDD nodes, we need to ensure that the reference counts are increased when results are put in the cache and decrease when results are overwritten. The `put` algorithm returns 0 when nothing was added to the memoization cache, 1 when the data was added and 2 when the data was added and existing results were overwritten. The overwritten results are exchanged with the data in the parameter, so the caller can update the reference counts afterwards.

In our implementation, hash values are 4 bytes and since processor cache lines are 64 bytes in most multicore architectures there are 16 buckets per processor cache line in the hash array.

For example, if a key hashes to the 14th bucket in a cache line and this bucket does not match, the algorithm will then check the 15th, the 16th and then the 1st bucket in the cache line, until all buckets have been checked. If one of them is empty, that bucket is used to store the data. If all buckets are filled and there are no matches, the initial bucket is overwritten. Unless another worker is by chance manipulating the same cache line, checking all buckets only requires a single memory transfer.

This method minimalizes the number of memory transfers necessary, while decreasing the chance that results are overwritten when there are *hash collissions*, i.e., when multiple keys have the same hash value.

In order to allow multiple workers to safely use the data structure, we use local short-lived locks. The highest bit of each hash in the hash array is used as a lock. It is set to 0 by default and set to 1 using the `compare_and_swap` instruction to set the lock. The owner of the lock then accesses the data array and releases the lock by setting the bit to 0 again.

The algorithm for `put` is given in Listing 5.1, using pseudocode with pointers and C-like bitwise operators to manipulate locks.

Note that reading the value of bucket will only involve memory transfers when the cache line is not yet in the processor cache, i.e., it will only involve memory transfers the first time and whenever the value has been changed elsewhere. Depending on the value of the bucket, a different action is taken. If the lock bit is set, the worker will wait until the lock is removed. If the lock bit is not set and the current value is `EMPTY`, the worker will try to acquire the lock using the `compare_and_swap` operation and write the data when it is succesful. If the lock bit is not set and the current value matches the hash of our data, the worker will try to acquire the lock and then compare the keys. Acquiring the lock is important since it may be possible that another worker modifies the cache while we are reading the data.

The `get` algorithm in Listing 5.2 is similar to the `put` algorithm.

This is the algorithm that we used for our experiments. It could be further improved in future work. For example, the algorithm currently waits for all locks, including locks on buckets it is not interested in. We can also relax certain requirements. The above algorithm assume that it is not allowed to have duplicate entries in the cache. This requirement can be relaxed. The algorithm can be simplified while maintaining the requirement that the result obtained using `get` is always a result entered into the cache using `put`. See Listing 5.3 and Listing 5.4.

5.4.2 Lockless hashtable with reference counting

As explained in Section 3.1, to store the BDD nodes we use a hash table and we need garbage collection.

We use *reference counting*, which is a common method to implement garbage collection. The number of references to a BDD node (or other object in the hash table) is recorded and when there are no more references, the node can be `freed`. Since we want to delay freeing the node as long as possible [43], we use a buffer implemented using a lockless memoization cache to store

```
put(data)
   hash := calculateHash(data, keylength) // calculate hash of key
   firstIndex := hash % tablesize // get index of first bucket
   for index ∈ cacheLineIndices(firstIndex)
      bucket := &hasharray[index]
      // if a worker has locked this bucket, wait for it
      await ! *bucket & LOCK
      // check if the bucket is empty
      if *bucket = EMPTY
         if compare_and_swap(bucket, EMPTY, hash|LOCK)
            write data to data array
            bucket := hash // unlock
            return 1 // added
      // check if the bucket matches hash
      while (*bucket&~LOCK) = hash
         if compare_and_swap(bucket, hash, hash|LOCK)
            if key matches key in data array
               bucket := hash // unlock
               return 0 // not added
            else
               bucket := hash // unlock
               break // escape while loop
      // if here, it is a different value and we can ignore it
   // No match and no empty spot...
   bucket := &hasharray[firstIndex]
   // acquire lock
   while ! compare_and_swap(bucket, (*bucket&~LOCK), hash|LOCK)
      await ! *bucket & LOCK
   exchange data with existing data in data array
   bucket := hash // unlock
   return 2 // overwritten
```

Listing 5.1: put algorithm

```
get(data)
   hash := calculateHash(data, keylength) // calculate hash of key
   firstIndex := hash % tableSize // get index of first bucket
   for index ∈ cacheLineIndices(firstIndex)
      bucket := &hasharray[index]
      // if a worker has locked this bucket, wait for it
      await ! *bucket & LOCK
      // check if the bucket is empty
      if *bucket = EMPTY return 0 // not in cache
      // check if the bucket matches hash
      while (*bucket & ~LOCK) = hash
         if compare_and_swap(bucket, hash, hash|LOCK)
            if key matches key in data array
               copy data to buffer
               bucket := hash // unlock
               return 1 // in cache
            else
               bucket := hash // unlock
               break // escape while loop
   // If there is no match, maybe it has been added in the first bucket
   bucket := &hasharray[firstIndex]
   while (*bucket & ~LOCK) = hash
      if compare_and_swap(bucket, hash, hash|LOCK)
         if key matches key in data array
            copy data to buffer
            bucket := hash; // unlock
            return 1 // in cache
         else
            bucket := hash // unlock
            return 0 // not in cache
      await ! *bucket & LOCK
   return 0 // not in cache
```

Listing 5.2: get algorithm

```
put(data)
   hash := calculateHash(data, keylength) // calculate hash of key
   firstIndex := hash % tablesize // get index of first bucket
   for index ∈ cacheLineIndices(firstIndex)
      bucket := &hasharray[index]
      // check if the bucket is empty
      if *bucket = EMPTY
         if compare_and_swap(bucket, EMPTY, hash|LOCK)
            write data to data array
            bucket := hash // unlock
            return 1 // added
      // check if the bucket matches hash
      if (*bucket&~LOCK) = hash
         if key matches key in data array
            return 0 // did not add
      // if here, it is a different value and we can ignore it
   // No match and no empty spot...
   bucket := &hasharray[firstIndex]
   // acquire lock
   if compare_and_swap(bucket, (*bucket&~LOCK), hash|LOCK)
      exchange data with existing data in data array
      bucket := hash // unlock
      return 2 // overwritten
   else
      return 0 // did not add
```

Listing 5.3: relaxed put algorithm

```
get(data)
   hash := calculateHash(data, keylength) // calculate hash of key
   firstIndex := hash % tableSize // get index of first bucket
   for index ∈ cacheLineIndices(firstIndex)
      bucket := &hasharray[index]
      // check if the bucket is empty
      if *bucket = EMPTY return 0 // not in cache
      // check if the bucket matches hash
      if (*bucket & ~LOCK) = hash
         if compare_and_swap(bucket, hash, hash|LOCK)
            if key matches key in data array
               copy data from data array
               bucket := hash // unlock
               return 1 // in cache
            else
               bucket := hash // unlock
   // If there is no match, never mind
   return 0 // not in cache
```

Listing 5.4: relaxed get algorithm

nodes that are *dead*, i.e., that are not referenced to anywhere. Nodes in this buffer are nodes that may in the future be garbage collected. Whenever an index in this buffer is overwritten, the corresponding BDD node is deleted from the hash table if the reference count is still 0. When the BDD is full, we empty the buffer and deleted every BDD in the buffer with the reference count set to 0.

As a hashtable, we use a data structure similar to [29] but modified to allow deleting nodes and to store the reference count. The hashtable consists of two arrays of equal size: the data array and the hash array. We use the lower bits of the hashes in the hash array to store the reference counter. This is expected to have low impact on hash collisions, since the low bits are often similar in hash collisions, while the high bits are different, due to the lower bits being used to determine the position in the hash table.

When an entry in the hash table is about to be deleted, we use a reserved value in the reference count field (`DELETING`) that indicates that the entry is being deleted. We also use a reserved value (`SATURATED`) to indicate that the reference count is *saturated*, which is a common technique to prevent integer overflows. When the reference count reaches the saturation value, it will no longer increase or decrease. When an entry has been deleted, the hash will be set to the reserved value `TOMBSTONE`.

Like the lockless cache, we use `compare_and_swap` and local short-lived locks to allow multiple workers to use the data structure safely.

Every bucket in the hash array is 4 bytes long. The highest bit is the `LOCK` bit. The low 16 bits are the reference count. The other 15 bits are the hash value.

The algorithm to increase the reference count is as follows:

```
incref(bucket)
   hash := *bucket
   rc := hash & RC_MASK // keep only the bits used for reference counting
   if rc = SATURATED then return SUCCESS // do not increase when saturated
   if rc = DELETING then return DELETING // contains DELETING value
   if compare_and_swap(bucket, hash, hash+1) then return SUCCESS
   return NOCAS // CAS failed, please retry
```

Listing 5.5: Algorithm to increase reference count

The algorithm to decrease the reference count is as follows:

```
decref(bucket)
   hash := *bucket
   rc := hash & RC_MASK // keep only the bits used for reference counting
   if rc = SATURATED then return SUCCESS // do not decrease when saturated
   assert rc != DELETING // impossible if you only decref after incref
   if compare_and_swap(bucket, hash, hash-1)
      if rc = 1 // original value was 1, so it is now 0
         // delegate NOWZERO event to GC handler (add to buffer)
         handleGCzero(bucket)
      return SUCCESS
   return NOCAS // CAS failed, please retry
```

Listing 5.6: Algorithm to decrease reference count

If `incref` or `decref` return `NOCAS`, they should be called again until either `SUCCESS` or `DELETING` (for `incref`) is the result. The return value `DELETING` means that the node should not be used anymore and a new node should be created.

We only need an algorithm `getOrCreate` to create unique BDD nodes. This algorithm is slightly more complicated than the algorithm for the lockless hash table by Laarman et al. [29] since we use tombstones for deleted values. The requirement of this algorithm is that it should either return a bucket that contains the input data or the reserved value `TABLE_FULL`. The returned bucket is either a newly created bucket or an existing bucket. In either case, the reference count must be increased by 1. The algorithm must guarantee that no duplicate nodes exist, i.e., there are no multiple buckets with the same data.

Similar to [29], there is a *probe sequence* of buckets that are checked in a predetermined order given the data to be entered into the hash table, which is calculated using the hash function. This probe sequence is based on using every processor cache line fully. The first bucket is calculated based on the result of the hash function and the cache line of this bucket is fully explored. Then

a new hash is calculated for a new bucket, and the cache line of that bucket is explored. This continues until a predetermined threshold that indicates the hash table is full.

In the original algorithm of Laarman [29], only the final bucket that will contain the data is locked while the data is copied to the data array. In our algorithm, it is possible to delete nodes, replacing them with tombstones, and reuse these nodes. It is therefore possible that another worker deletes an node and that a third worker uses this bucket to store the data the first worker also wants to add. To prevent this, we do not lock on the final bucket, but on the first bucket. Workers that want to add data with the same first bucket will wait until the first worker has completed the algorithm.

The first encountered tombstone is also locked to reserve it for possible use. When an empty bucket is found, this means that the data is not yet in the hashtable.

See Listing 5.7 for the result.

This is the algorithm that we used for our experiments. It could be improved further, for example by overwriting existing BDD nodes when their reference count is 0.

5.5 Conclusions

In this chapter, we discussed the two approaches that we will use in our experiments. We presented an overview of how Wool works and how result sharing works. We also presented our extensions of the lockless hashtable by Laarman et al. [29]: a lockless memoization cache and a lockless hashtable with garbage collection implemented using reference counting. These lockless data structures form the basis of our solution to parallelizing BDD operations.

```
getOrCreate(data)
   hash := calculateHash(data, keylength) // calculate hash of key
   // calculateHash never returns TOMBSTONE or EMPTY
   firstIndex := hash % tableSize // get index of first bucket
   firstBucket := &hasharray[firstIndex]
   hash &= 0xFFFF0000 // Set lower bits to 0 for reference count

   // lock firstBucket
   while ! compare_and_swap(firstBucket,*firstBucket&~LOCK,*firstBucket|LOCK)
      await ! *firstBucket & LOCK

   for index ∈ cacheLineIndices(firstIndex)
      bucket := &hasharray[index]
      if HASHPART(*bucket) = EMPTY
         // data not yet in table... write somewhere
         if we reserved a tomb bucket
            write data to tomb bucket data array
            *tombBucket := hash + 1 // unlock and set reference count to 1
            if tombBucket != firstBucket then *firstBucket &= ~LOCK // unlock
            return tombBucket
         if bucket = firstBucket // already locked
            write data to data array
            *bucket := hash + 1 // unlock and set reference count to 1
            return bucket
         // lock it
         if compare_and_swap(bucket, EMPTY, hash|LOCK)
            write data to data array
            *bucket := hash + 1 // unlock and set reference count to 1
            *firstBucket &= ~LOCK // unlock first bucket
            return bucket
         // CAS failed, wait until other worker is done
         *firstBucket &= ~LOCK // unlock first bucket, to prevent deadlocks
         await ! *bucket & LOCK // wait until lock is released
         // restart algorithm because we lost our lock on the first bucket
         return getOrCreate(data)
      if HASHPART(*bucket) = hash
         await incref(bucket) = SUCCESS
         if data matches data array
            if we reserved a tomb bucket then *tombBucket &= ~LOCK // unlock
            if tombBucket != firstBucket then *firstBucket &= ~LOCK // unlock
            return bucket_index
         await decref(bucket) != NOCAS
      if no reserved tomb bucket and HASHPART(*bucket) = TOMBSTONE
         if bucket = firstBucket // already locked
            tombBucket := firstBucket
         else if compare_and_swap(bucket, TOMBSTONE, TOMBSTONE|LOCK)
            tombBucket := bucket
         // if compare_and_swap failed, never mind...
   // If we are here, there was no empty bucket...
   if we reserved a tomb bucket
      write data to data array
      *tombBucket := hash + 1 // unlock and set reference count to 1
      if tombBucket != firstBucket then *firstBucket &= ~LOCK // unlock
      return tombBucket
   firstBucket &= ~LOCK // unlock
   return TABLE_FULL
```

Listing 5.7: `getOrCreate` algorithm

6

Experiments

The current chapter discusses our experiments. First we explain the design of our experiments. We then present and discuss the results of experimenting using the framework Wool. After this, we present and discuss the results of our experiments with the result sharing approach. Then we compare the results to the performance of using the BuDDy library for BDD operations. We conclude this chapter with conclusions.

6.1 Experimental setup

The goal of our experiments is to assess the efficiency of parallelization techniques that increase the performance of BDD operations frequently used in model checking tools.

We performed our experiments using a model checker (`dve2-reach` from the `LTSmin` toolset [9])), our own implementation of a BDD library (Sylvan) and a parallellization framework, either Wool or result sharing.

We measured the execution time of each run. We also gathered some other data, for example the number of calculations, how much work each worker did, etc. We determined that the influence of data gathering on the result of the experiments is negligible, so we did not perform the experiments and the data gathering separately. In addition, we used the statistical profiler `gperftools` [2] for CPU profiling, measuring the runtime behavior of our program.

We expected some random noise, due to environmental influences such as temperatures. After running several small experiments, we found the random variations in our measurements to be small enough not to influence our observations and conclusions.

We also compared our results with the performance of a sequential BDD package.

A list of performed experiments can be found in Table 6.1.

Table 6.1: Performed experiments on all selected models using 1-48 workers

experiment name	experiment description
`wool`	Basic parallelization using Wool
`wool.n`	Same as `wool`, but using the NUMA interleaving memory allocator
`rs`	Basic parallelization using result sharing
`rs.s`	Same as `rs` but using a kill flag
`rs.ns`	Same as `rs.s`, but using the NUMA interleaving memory allocator
`rs.nas`	Same as `rs.ns`, but setting the CPU affinity of each worker
`wool.2`	Same as `wool`, but using the cache strategy $N = 2$
`wool.3`	Same as `wool`, but using the cache strategy $N = 3$
`wool.4`	Same as `wool`, but using the cache strategy $N = 4$
`rs.2`	Same as `rs`, but using the cache strategy $N = 2$
`rs.3`	Same as `rs`, but using the cache strategy $N = 3$
`rs.4`	Same as `rs`, but using the cache strategy $N = 4$

Reachability in `LTSmin` We used the tool `dve2-reach` from the `LTSmin` toolset [9] to run the experiments. The `dve2-reach` tool explores all reachable states for models expressed in the DiVinE-2 language. The tool implements a number of reachability algorithms.

The basic reachability algorithm (Listing 3.3 in Section 3.4), modified to use the `RelProdS` algorithm introduced in Section 3.5, is given in Listing 6.1.

```
Set Reach(Set start, Set T)
   Set Visited := start, Previous := ∅
   while Visited != Previous
      Previous := Visited
      Set Next := RelProdS(Visited, T)
      Visited := Visited ∪ Next
   return Visited
```

Listing 6.1: Basic reachability algorithm

In the `LTSmin` toolset, this algorithm is slightly more complicated, because the transition relation is calculated on-the-fly. After every iteration in the reachability algorithm, the transition relation is updated with new transitions. In addition, the transition relation is split into multiple transition groups. The number of transition groups depends on the model. See Listing 6.2 for the implementation of reachability in `LTSmin`.

```
Set reach_bfs(Set start, Set[] group_next)
   Set visited := start, old_vis := ∅
   while visited != old_vis
      old_vis := visited
      for i ∈ {1,...,number of groups}
         expand_group_next(i, visited)
      for i ∈ {1,...,number of groups}
         visited := visited ∪ RelProdS(old_vis, group_next[i])
   return visited;
```

Listing 6.2: Reachability algorithm `reach_bfs` in `LTSmin`

In our experiment, we precompute the full transition relation first, so we can focus on the reachability algorithm instead of on efficiently updating of the transition relation. If we do not precompute the transition relation, we would also measure the performance of the `expand_group_next` function, in which we are not interested.

To prevent the BDD calculations from reusing cached results from the first loop, the memoization caches are emptied first. See Listing 6.3 for the final algorithm.

```
Set reach_bfs(Set start, Set[] group_next)
   Set visited := start, old_vis := ∅
   while visited != old_vis
      old_vis := visited
      for i ∈ {1,...,number of groups}
         expand_group_next(i, visited)
      for i ∈ {1,...,number of groups}
         visited := visited ∪ RelProdS(old_vis, group_next[i])
   Empty memoization caches, start timer.
   visited := start
   old_vis := ∅
   while visited != old_vis
      old_vis := visited
      for i ∈ {1,...,number of groups}
         visited := visited ∪ RelProdS(old_vis, group_next[i])
   Stop timer and report results.
   return visited
```

Listing 6.3: Modified version of `reach_bfs`

Note that the implementation of reachability is oblivious of the parallelization, since this is implemented in the actual BDD operations.

To run our experiments we call `LTSmin` with the parameter `-rgs`. This parameter enables static reordering of variables at the start of the reachability algorithm. We used this parameter for our experiments with BuDDy and with Sylvan.

Implementation of Sylvan Sylvan is our experimental BDD library. We implemented BDD operations as explained in Chapter 3 using the frameworks and data structures presented in Chapter 5.

We extended the `LTSmin` toolset to use Sylvan as a backend for symbolic model checking. Version 1.9 of `LTSmin` [3] and the development branch (next) of `LTSmin` contain this extension. We used the parameters `--sylvan-cachesize=29` and `--sylvan-datasize=29` to use the largest possible hash tables (256*1024*1024 entries).

The full command is: `dve2-reach -rgs --vset=sylvan --sylvan-threads=... --sylvan-cachesize=29 --sylvan-datasize=29 --sylvan-bits=...` *`model.dve2C`*

The parameter `sylvan-bits` is a model-specific parameters for the number of bits per state integer in `LTSmin`, similar to the `fdd-bits` parameter for the BuDDy backend in `LTSmin`.

The performance of Sylvan may not be as good as optimized libraries like BuDDy and CuDD, but this is unlikely to have a significant impact on the experimental results, since we are interested in the speedups due to parallelization.

Models Since we want to measure the performance of symbolic model checking, we executed the reachability algorithm on various different models. We performed our experiments using several models from the BEEM database [37] as input database. The BEEM database is a database for explicit model checking. We selected models from this database at random and removed some of the smaller variations from our selection. See Table 6.2 for some data on the models we selected. In this table, *depth* is the number of iterations in the reachability algorithm until all reachable states have been found. *Groups* is the number of transition groups in `LTSmin`. *States* is the total number of reachable states and *nodes* is the number of BDD nodes required to store the BDD representing the final set of reachable states in memory. `RelProdS` and `ITE` operations consist of a large number of small tasks. *Work* is the actual number of non-trivial `RelProdS` and `ITE` subcalculations performed during reachability using a single worker.

Note that the depth and the number of groups determine the number of calls from the reachability algorithm to `RelProdS` and `ITE`. We only included the number of reachable states and the number of BDD nodes required for the set of reachable states for completeness. The size and shape of the BDDs during the reachability operation is much more important for the number of operations to be executed.

Architecture We performed our experiments on a NUMA machine with 4 AMD Opteron™ 6168 processors, each with 12 cores, a total of 48 cores. There is 128GB total memory and each core has 64KB L_1 cache. Each processor has 512KB L_2 cache and 5118KB L_3 cache.

6.2 Results using Wool

The first experiment we performed was simply the reachability algorithm discussed above using the Wool framework. The speedups obtained in this experiment are shown in Figure 6.1. The speedup is calculated relative to 1 worker.

It is obvious from Figure 6.1 that there is a strong relation between the size of the input model in number of operations and the obtained speedup. See again Table 6.2 for the size of the models. See also Figure 6.2 for a plot of the best speedup of each model and the number of operations per model.

Ideally, we would want to see a linear speedup graph where the speedup is equal to the number of workers, i.e., when we have 48 workers we would like to see a speedup of 48. Lacking that, we would like a speedup graph where the speedup never decreases with increasing number of workers. The speedup we obtained was in the best case 20, for the `collision.5` model using 36 workers. In all cases, the speedup decreases after usually about 36 workers. While this is certainly a good result, we are interested in investigating why the speedup is limited. We would like to know why there is a peak around 36 workers.

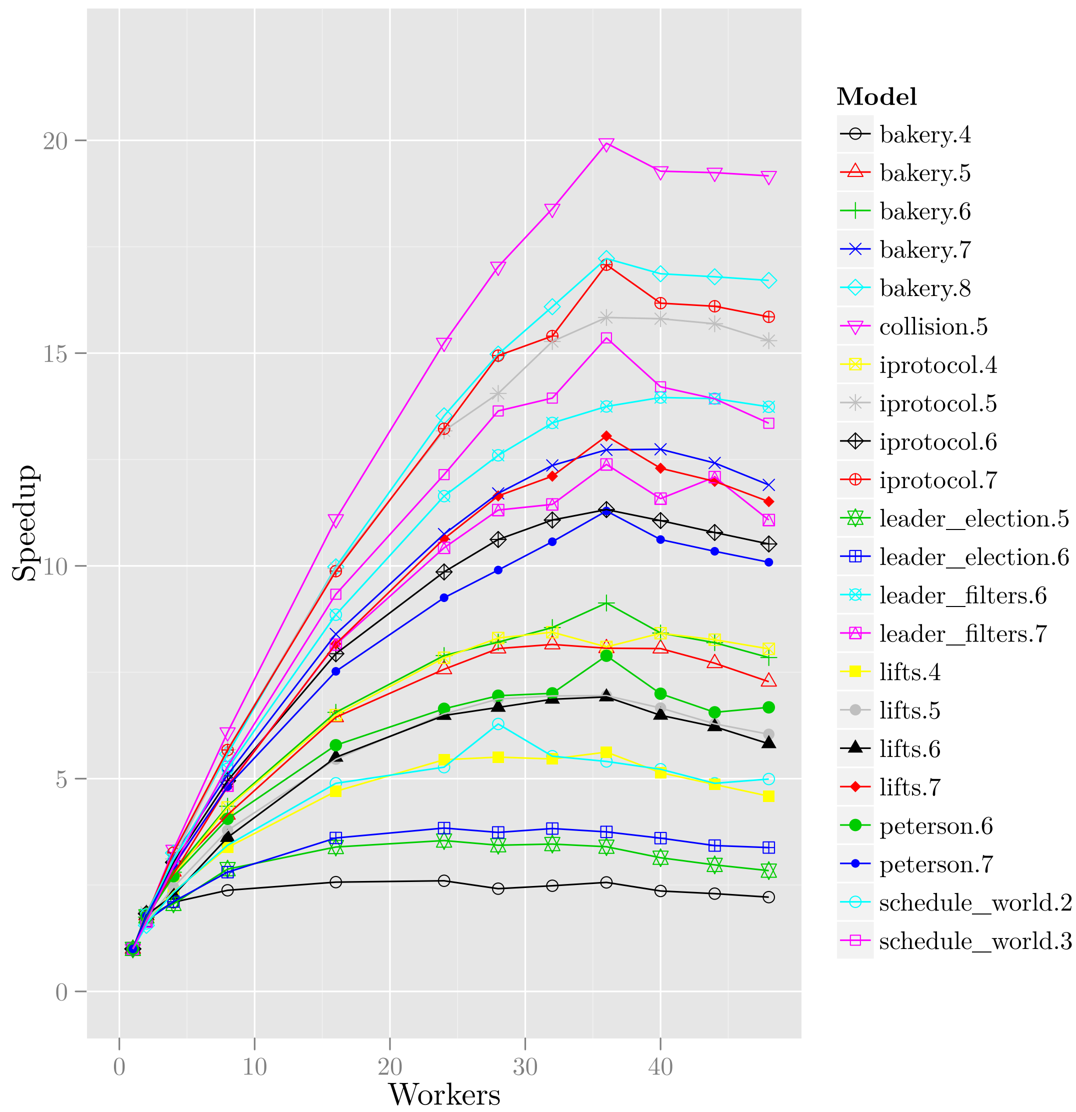


Figure 6.1: Wool results (1-48 workers, all models)

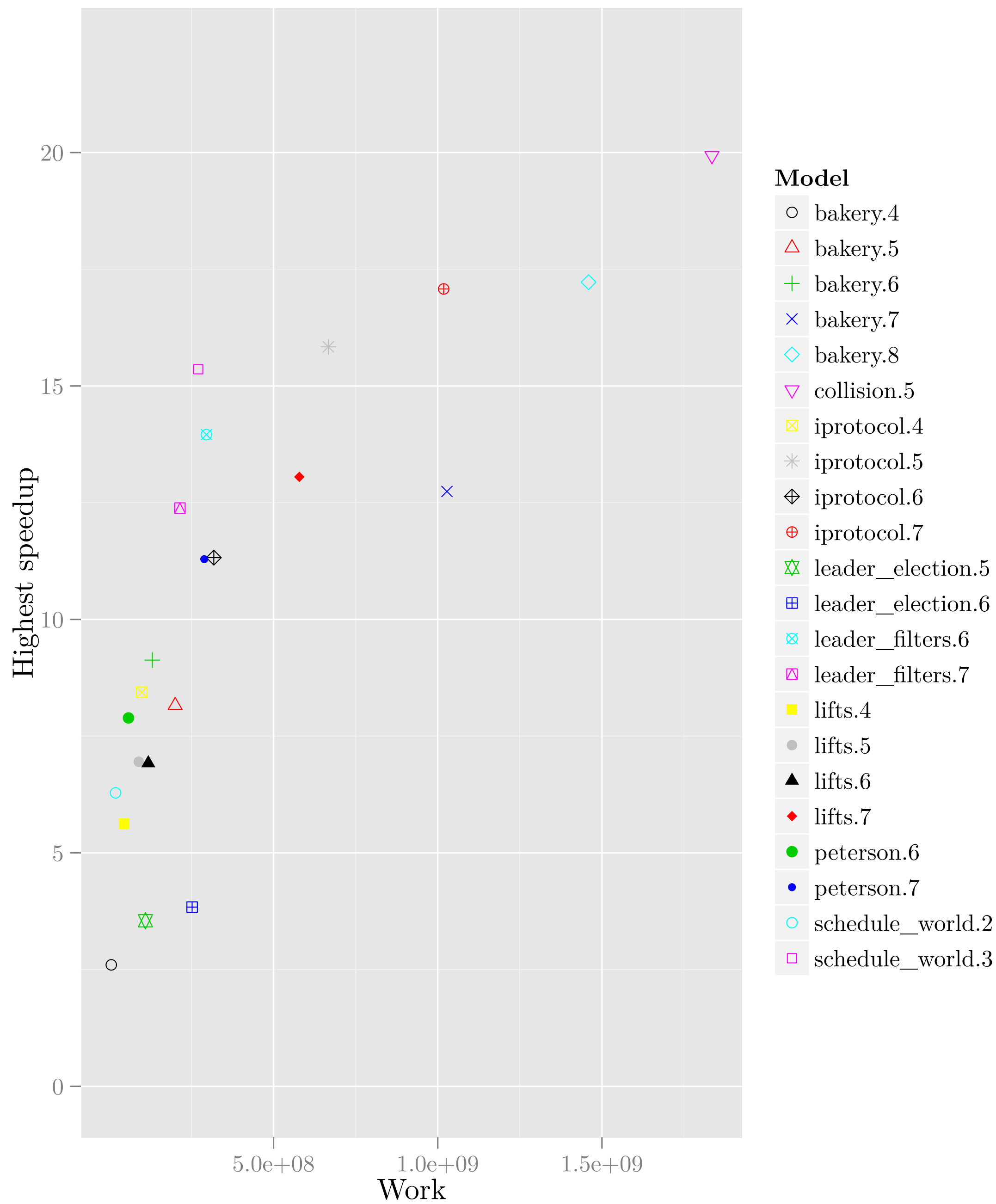


Figure 6.2: Best speedup for Wool compared to work per model

Table 6.2: Model data

Model	Depth	Groups	States	Nodes	Work
bakery.4	103	21	157,003	12,009	6,003,764
bakery.5	288	28	7,866,401	156,771	198,664,965
bakery.6	129	28	11,845,035	148,119	130,645,221
bakery.7	448	28	29,047,471	681,012	1,018,315,634
bakery.8	293	35	253,131,202	582,463	1,452,973,233
collision.5	179	29	431,965,993	29,537	1,747,001,660
iprotocol.4	166	26	3,290,916	113,774	97,854,082
iprotocol.5	232	26	31,071,582	545,578	665,267,682
iprotocol.6	452	26	41,387,484	190,427	314,269,205
iprotocol.7	279	26	59,794,192	676,092	1,000,748,636
leader_election.5	159	88	4,803,952	62,682	109,241,755
leader_election.6	220	99	35,777,100	106,124	250,620,273
leader_filters.6	84	20	220,913,716	466,178	286,035,694
leader_filters.7	70	24	26,302,351	326,990	203,210,754
lifts.4	115	69	112,792	118,212	44,359,194
lifts.5	115	69	191,567	173,495	88,308,709
lifts.6	214	91	333,649	392,470	115,905,084
lifts.7	218	91	5,126,781	569,314	573,845,199
peterson.6	79	20	174,495,861	38,169	57,614,947
peterson.7	175	25	142,471,098	72,275	286,336,555
schedule_world.2	16	26	1,570,340	18,779	18,376,072
schedule_world.3	21	34	166,649,331	28,500	265,204,724

There are a number of possible explanations. We investigated these explanations and describe how we examined them.

1) The amount of redundant work increases with the number of workers, due to multiple workers starting the same calculation before the result has been calculated, which is caused by subgraph sharing. Therefore, if input BDDs of the model checking suboperations feature heavy subgraph sharing, redundancy is more likely. The relative increase of work is shown in Figure 6.3. Work in Figure 6.3 is the sum of all non-trivial `ITE` and `RelProdS` operations. It is clear that the increase of work is very small (highest was 7%) and not sufficient to explain the limited speedups.

2) There may be a significant overhead from Wool to distribute work. We can partially test this hypothesis by comparing with the speedup graph of the Fibonacci algorithm. See Listing 6.4 for the algorithm. We use the Fibonacci algorithm because it is a very tiny example of a similar problem: compute two subproblems and return the sum. We did not use any memoization caches for the calculation.

   See Figure 6.4 for the result. Each point was measured at least 3 times and the average was taken. From Figure 6.4 it appears there is indeed a significant overhead in Wool which limits the possible speedup. This overhead may also be due to the architecture of the system we performed our experiments on. Note that the actual calculation of the Fibonacci algorithm takes almost no time compared to the overhead from Wool, so what we are measuring is how well Wool scales on our machine.

   When parallelizing model checking the scalability of Wool only partially determines the scalability of parallel model checking, since the tasks are not nearly empty as with the Fibonacci algorithm, but consist of consulting the memoization cache, creating a BDD node (or increasing the reference count of an existing BDD node) and updating the memoization cache.

   An interesting feature of Figure 6.4 is the sudden performance drop from 31 to 32 workers. We do not see a similar drop in our experiments. A second interesting feature is the slightly improved performance around 40 workers. We believe these two features may be related somehow to our particular architecture, although we do not understand exactly how. This is something that could be investigated in future work.

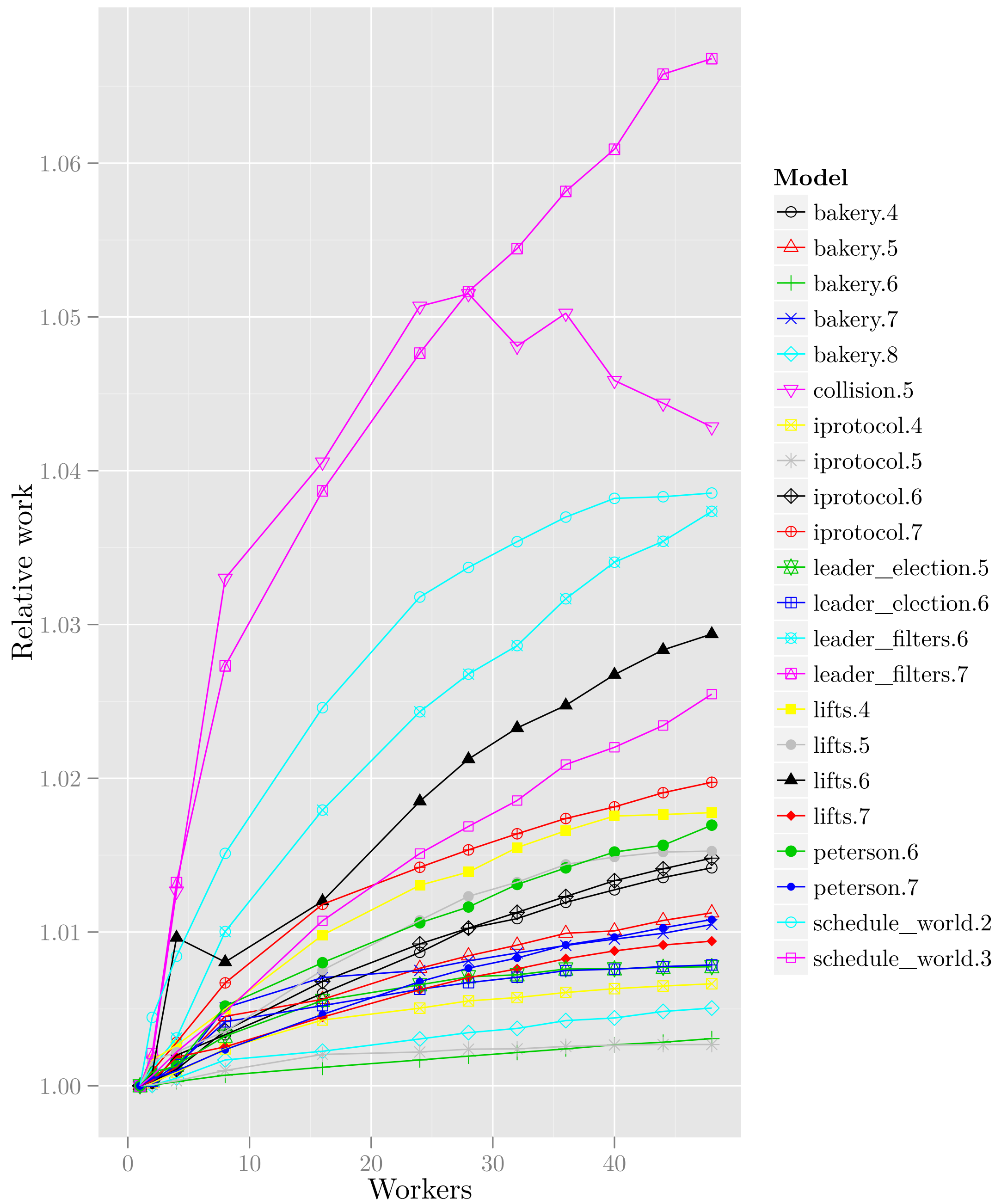


Figure 6.3: Wool relative work with increased workers

```
TASK(fib, n)
   if n < 2 then return n
   SPAWN(fib, n-1)
   m := CALL(fib, n-2)
   return m + SYNC

fib(n)
   return CALL(fib, n)
```

Listing 6.4: Fibonacci algorithm parallelized using Wool

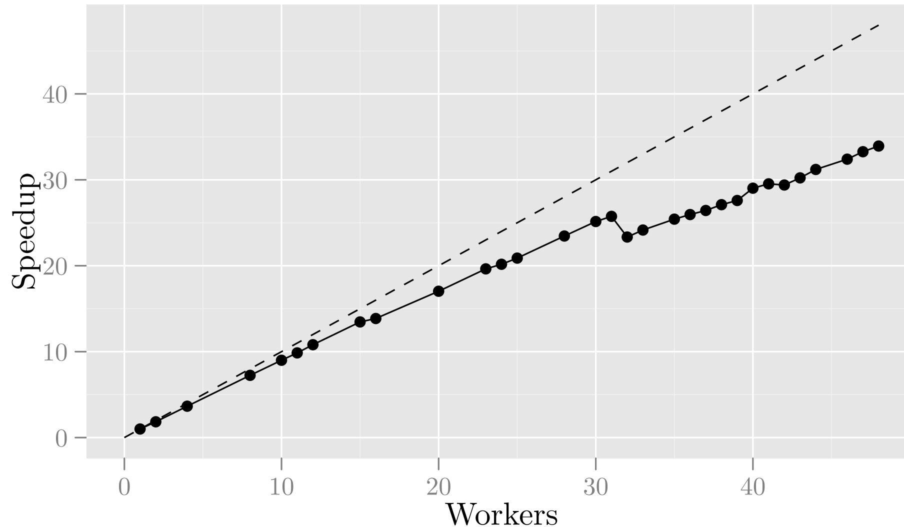


Figure 6.4: Result of parallelizing Fibonacci using Wool

3) Maybe the models are not large enough to obtain good scalability. This is related to the notion of the critical path from Chapter 4. A large critical path (β) implies that there is limited parallelism. If there is limited parallelism, we should be able to measure how often workers are waiting for work using statistical profiling. By measuring how often the program is in the `do_work` and `steal` functions of Wool, we may be able to judge whether this is an issue.

Table 6.3: CPU profile of `bakery.4` with 48 workers

function	percentage	additional information
`steal`	50.8%	waiting for work plus overhead
`do_work`	28.5%	waiting for work plus overhead
`try_ref`	4.6%	increasing reference count
`ticketlock_trylock`	3.2%	locking primitive in Wool (overhead)
`myrand`	2.3%	random function called for work stealing in Wool

Table 6.3 and Table 6.4 show the percentage of time spent in a certain function during the experiment. Table 6.3 shows the top 5 functions when executing the experiment on the `bakery.4` model using 48 workers and table 6.4 shows the top 14 functions for the larger `bakery.8` model with 48 workers. It is clear that in the smaller model at least 84.8% of the time is spent inside Wool functions. In the larger model this number is much smaller: only at least 22.5% of the time is spent inside Wool functions. Based on these numbers we believe it very likely that with `bakery.4` there is too little work to parallelize with 48 workers. This is less clear with the `bakery.8` model. It may be possible that not every operation in `bakery.8` is sufficiently large, since the reachability algorithm consists of a large number of smaller operations.

We should take into account that the reachability consists of a large number of operations and that we are really measuring the average scalability of these operations. Our data suggests that the `bakery.4` model largely consists of operations that do not scale well. With the `bakery.8` model, it is less clear how often the workers were waiting for work, but it is less than with the `bakery.4` model.

4) Since we performed our experiments on a NUMA architecture, it might be possible that memory allocation over different memory banks cause or worsen memory bandwidth problems. We ran an experiment where the hash tables were allocated interleaved using the NUMA library, function `numa_alloc_interleaved`. The result of this experiment (showing only with 32-48 workers) compared to using the default allocator (`posix_memalign`)

Table 6.4: CPU profile of `bakery.8` with 48 workers

function	percentage	additional information
steal	17.6%	waiting for work plus overhead
llcache_get_and_hold	16.2%	retrieving from memoization cache
try_ref	16.1%	increasing reference count
try_deref	10.6%	decreasing reference count
lock	6.5%	locking in LL hash table
relprods	4.4%	calculating RelProdS
llcache_put_and_hold	3.8%	putting in memoization cache
get_thread_id	3.5%	auxiliary function
ite	3.4%	calculating ITE
memxchg	2.7%	called when overwriting in memoization cache
llgcset_lookup_hash	2.1%	creating nodes in LL hash table
wool_sync	1.9%	implementation of SYNC macro
do_work	1.8%	waiting for work plus overhead
ticketlock_trylock	1.2%	locking primitive in Wool (overhead)

is presented in Figure 6.5. The plot compares the time in seconds of every experiment performed using interleaved allocated memory and using normal allocated memory. It is clear that there is little or no improvement.

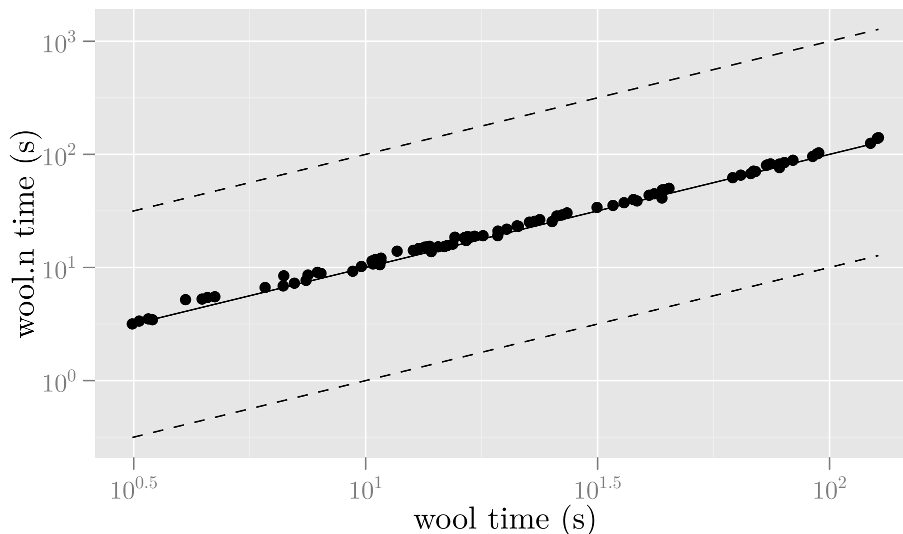


Figure 6.5: Results of using interleaved memory allocation (36-48 workers, all models)

5) Thread migration may cause performance degradation. The Linux kernel may sometimes move workers to different processor cores. In addition, migrating a worker also implies moving local data. To prevent this, the *processor affinity* can be set manually rather than automatic assignment by the Linux kernel. We checked this hypothesis for the experiments using result sharing. See below for the results. We found that there was no improvement when processor affinity was set manually. In fact, on some models setting the processor affinity would result in severely worse performance. We expect that thread migration is not an issue.

In conclusion, we found that redundant work, memory allocation on the NUMA architecture and possible thread migration are not significant in limiting the speedups of our algorithm and that Wool overhead and limited parallelism are both factors in limiting speedups.

After exploring the above explanations, we now formulate the hypothesis, based on the model in Chapter 4, that limited memory bandwidth and limited scalability of the data structures are

a major cause of limited speedups.

Creating or reusing a new BDD node involves first setting a lock on the first bucket, then either setting a lock on the target bucket, writing data, and releasing all locks, or increasing the reference count, comparing data and optionally decreasing the reference count when the data doesn't match. Of these, setting locks and manipulating the reference counts involves the atomic primitive `compare_and_swap`. Writing entries into the memoization cache involves atomically incrementing the reference counts of all parameters and of the result BDD, as well as setting and releasing a local lock on the target bucket. With the exception of releasing locks, these operations are all implemented using `compare_and_swap`.

BDD operations mainly consist of calculating new BDD nodes, testing whether or not these nodes already exist, and putting results of operations in the memoization cache. Therefore, BDD operations are dominated by `compare_and_swap` operations. Creating a new BDD node and adding a new entry to the memoization cache involves at least 6 `compare_and_swap` operations for `ITE` and 7 for `RelProdS`. When these `compare_and_swap` operations fail, there is an additional cost to reload the value in processor cache and locally restart the procedure.

In Figure 6.1 we can see that the highest speedup is obtained around 36 workers. From the discussion in Chapter 4 we expect that such peaks occur at points where the added communication overhead in relation to limited memory bandwidth is more expensive than the performance benefits of adding a worker. From Figure 6.4 we know that the added communication overhead in Wool even with 48 workers does not decrease performance, so we must attribute this behavior to communication overhead in the data structures (atomic restarts due to concurrent modifications of the same memory location) and to memory bandwidth limitations. Of these two, it is much more likely that memory bandwidth is limited, since concurrent modifications of the same memory location are unlikely to strongly influence our results since the hash tables are very large.

We ran an experiment in which N workers performed 500,000,000 CAS operations on random locations in a preallocated array of 100,000 32-bit integers. We extrapolated the measurements to expected speedup in Figure 6.6. This figure suggests there is limited speedup due to bandwidth limitations, but we do not see a peak like in Figure 6.1. This is probably due to the fact that our benchmark of `compare_and_swap` uses a fully randomized access pattern, which is not the case in actual BDD operations.

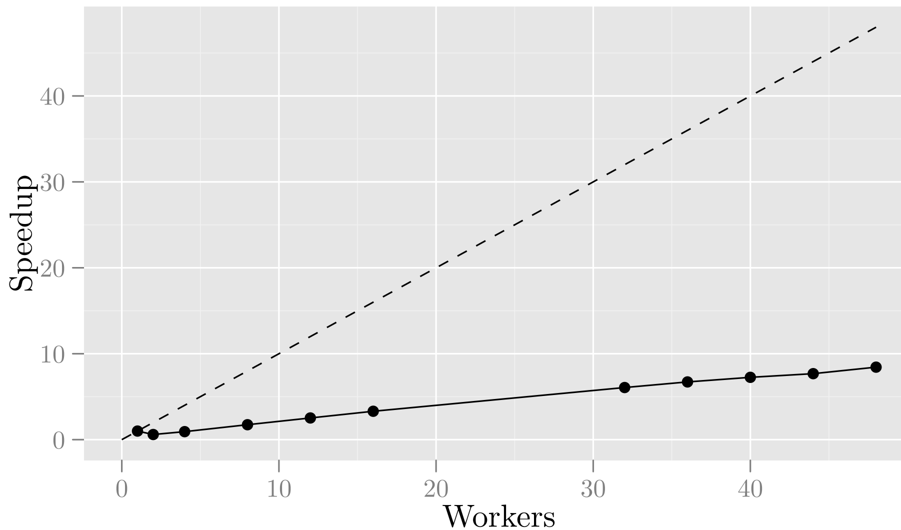


Figure 6.6: Expected speedup graph with `compare_and_swap`

Finally, after every operation, there is some sequential execution between operations. According to Amdahl's Law, this limits possible speedups. We expect the influence from this sequential code to be very small in our case, since this sequential part only consists of immediately calling the next parallel operation.

Conclusions The limited scalability of the reachability algorithm may be due to the combination of limited parallelism, limited bandwidth, and overhead from Wool. It appears that redundant work does not significantly decrease performance. We excluded several other explanations.

We think the shape of the speedup graph, with a peak around 36 workers, is the result of the memory bandwidth requirements of the BDD operations when there is sufficient work to parallelize, which consist mainly of memory transfers of which most are the atomic primitive `compare_and_swap`, plus the overhead from Wool.

The reachability algorithm consists of a number of smaller BDD operations. Some of these scale well and others scale worse. The height of the speedup graph, i.e., whether the highest speedup is 20 or perhaps 5, is determined by the average scalability of these smaller BDD operations.

In order to improve the scalability, we should look at the number of memory transfers required per BDD node, especially `compare_and_swap` operations. Less use of the memoization cache could be an option to reduce the number of memory transfers on average, at the cost of more redundant operations. See Section 6.5. Another option would be removing updates to the reference counts when using the memoization cache. Changing to an architecture with a larger memory bandwidth would also increase the scalability, for obvious reasons. Finally, if it is possible to merge small transition groups in `LTSmin`, for example using the parameter `-rga`, to increase the scalability of individual BDD operations without significantly increasing the total amount of work, the speedups may improve without compromising overall performance.

6.3 Results using result sharing

The second experiment we performed uses result sharing for parallelism rather than Wool. The speedups obtained in this experiment are shown in Figure 6.7. If we compare Figure 6.7 to Figure 6.1 we see similar results, except that most speedup lines are smoother in Figure 6.7. See Section 6.4 for a more detailed comparison of these results with the results using Wool.

Similarly to our experiments using Wool, we can explore various explanations for lower speedup.

1) The amount of redundant work increases with the number of workers, due to multiple workers starting the same calculation before any worker has calculated the result. The relative increase of work is shown in Figure 6.8. Comparing Figure 6.8 to Figure 6.3 it is obvious that there is much more redundanct work than with Wool. Where using Wool the worst case was 7% increased work, with result sharing we get at least 20% and at worst over 600% increased work. This may be explained from limited parallelism, which using Wool is expressed by excessive time spent in `steal` and `do_work`. With Wool there is only redundant work when there is subgraph sharing, but in the approach with result sharing every calculation is performed by every worker unless the result has been calculated already.

   If we compensate for the redundant work, by dividing the measured time by the amount of work relative to 1 worker, we would get a speedup graph as in Figure 6.9. As is shown by Figure 6.9, merely calculating more work is insufficient to explain limited speedup. Still, Figure 6.10 suggests a correlation between the amount of redundant work and less speedup. That does not mean that calculating redundant work causes less speedup beyond what we compensated for in Figure 6.9. Another explanation is that redundant work is not the only consequence of limited parallelism. For example, with limited parallelism it is more likely that multiple workers are competing to calculate the same suboperation, resulting in extra memory transfers between processor caches and main memory. Figure 6.11 suggests there may be a relation between the amount of work and the amount of redundant work, but the `leader_election` models are a clear exception. This is an area that may require additional future investigation.

2) There may be overhead from the framework that causes limited speedup. The framework for result sharing is much lighter than Wool, since it only involves some communication at the start of each root operation and waiting for all workers to be finished at the end of each root operation. We expect the overhead at the start of the operation to be insignificant.

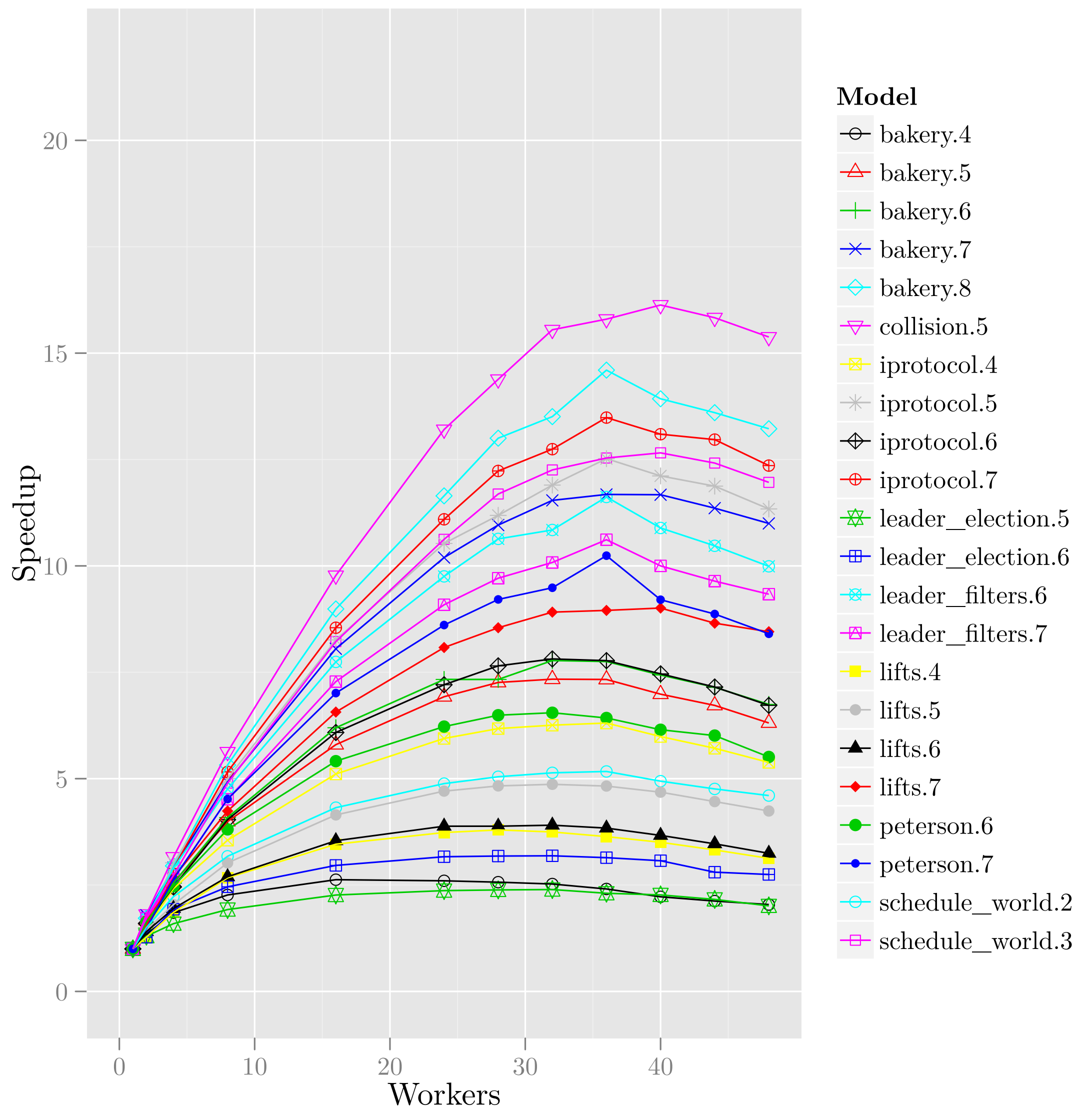


Figure 6.7: Results of using result sharing

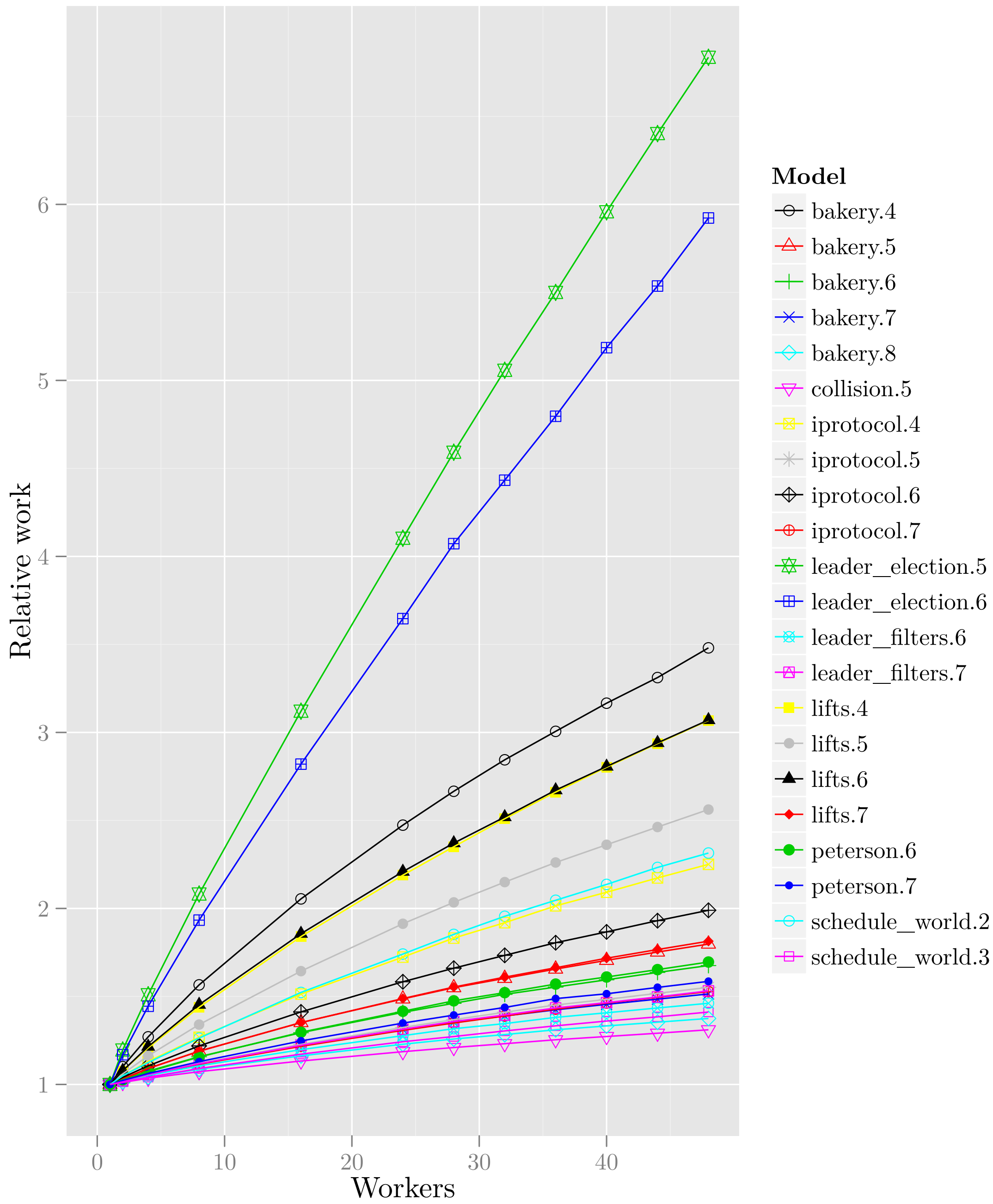


Figure 6.8: Relative work with result sharing

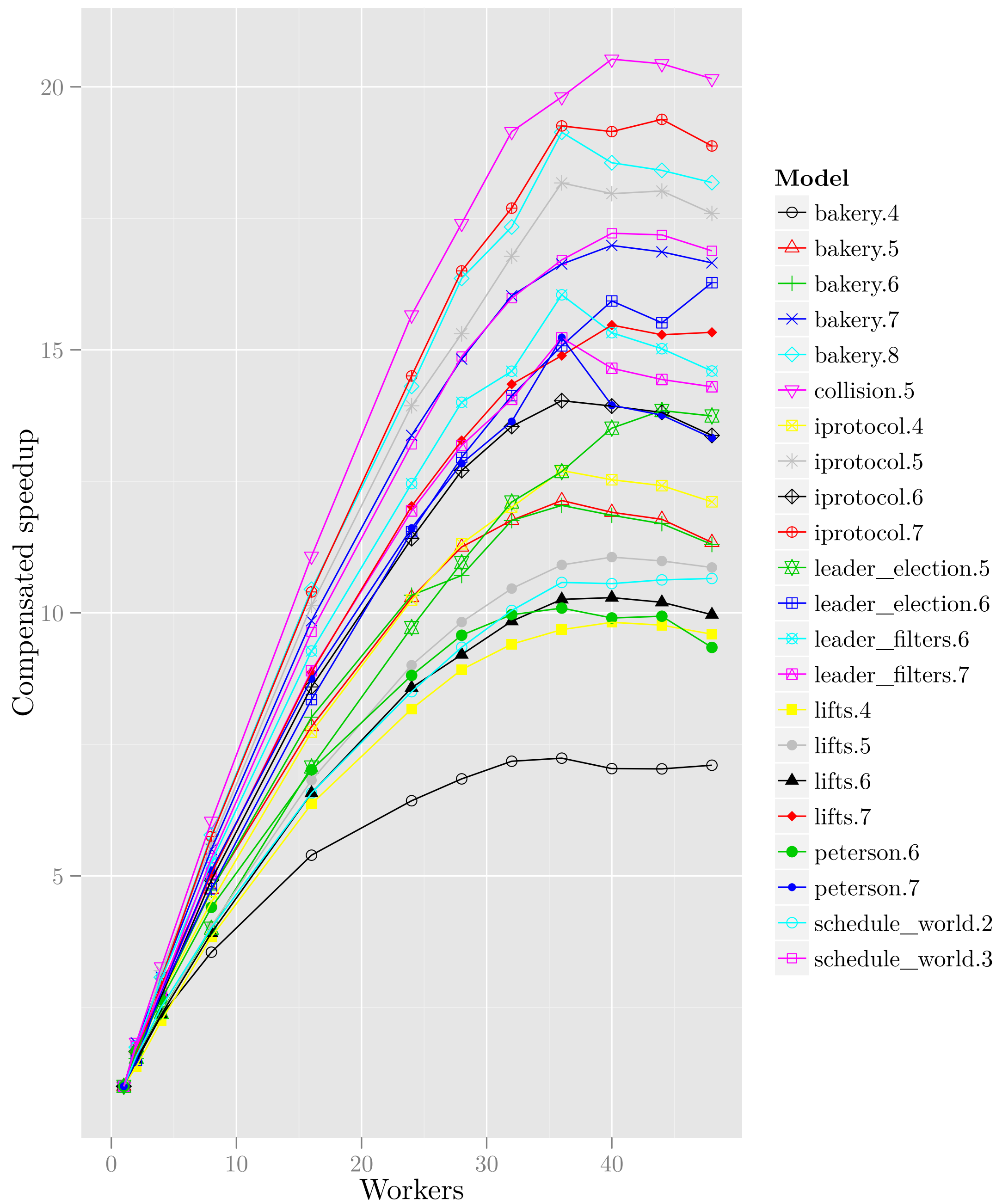


Figure 6.9: Results compensated for redundant work

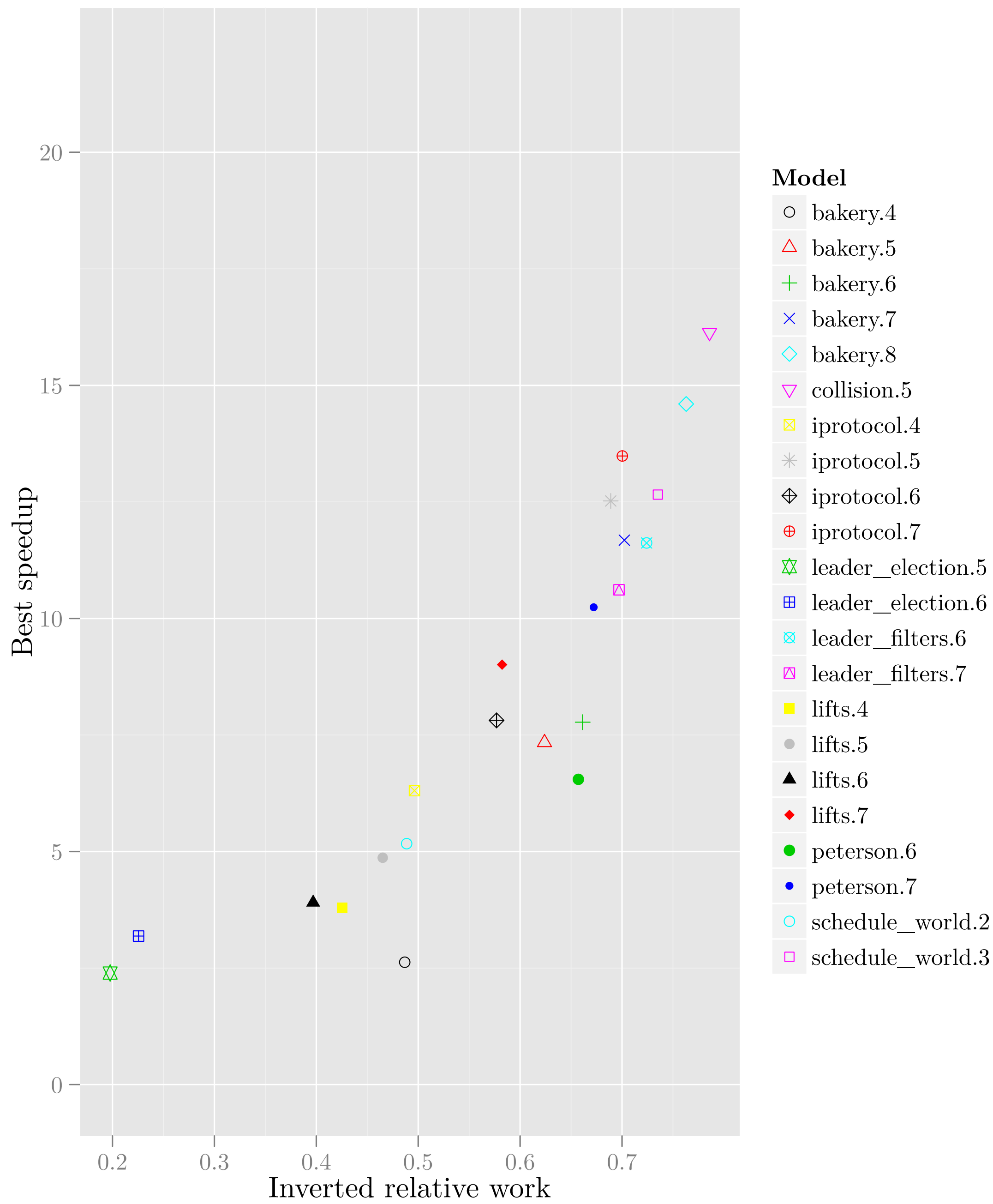


Figure 6.10: Speedup versus relative work using result sharing (only best result)

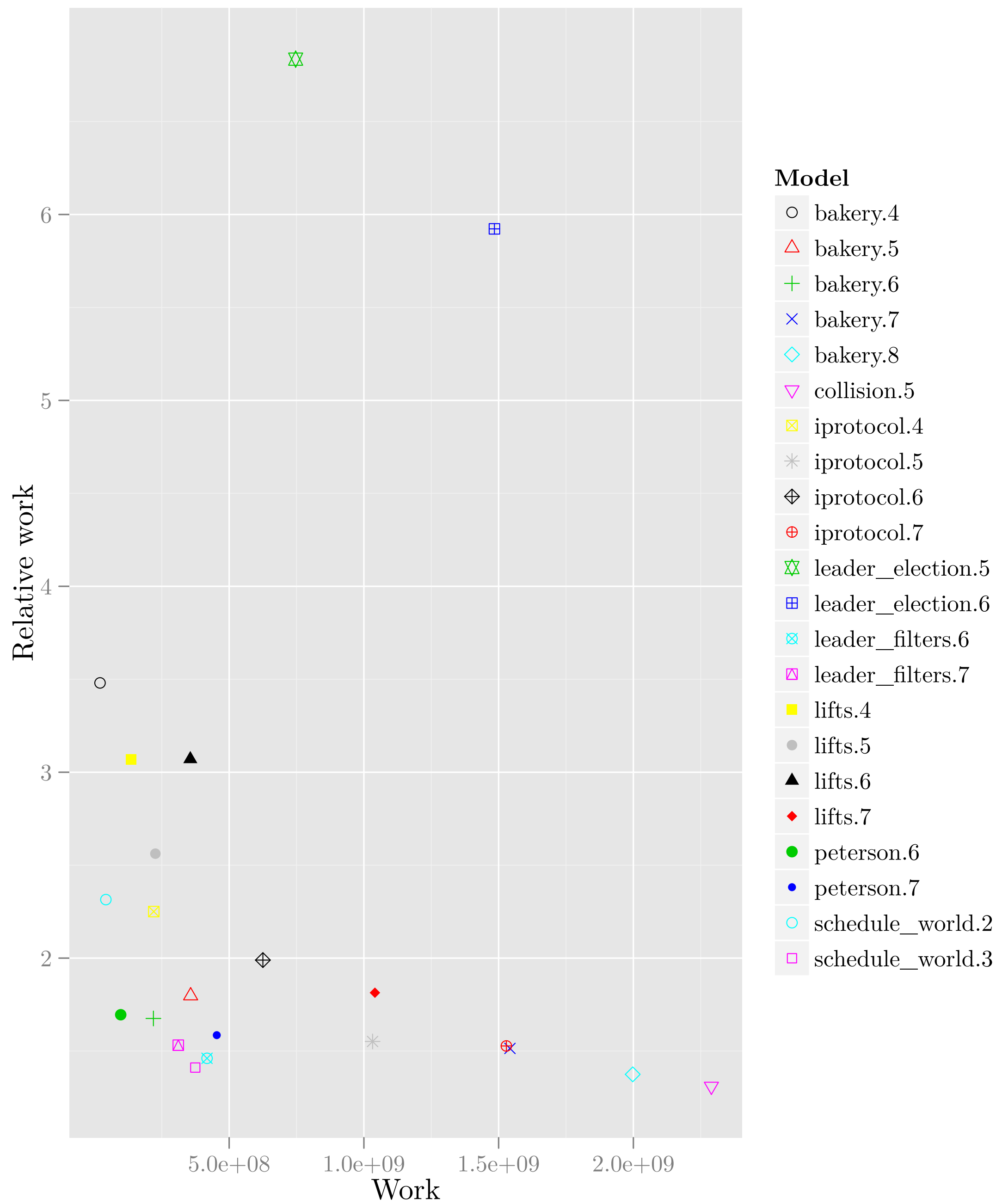


Figure 6.11: Relative work versus work

However, it is possible that some workers finish the calculation much earlier than other workers. Since the caller waits until all workers have finished the calculation, this may cause performance loss. In theory, once any worker has finished the calculation, it is expected that all results of the operation and suboperations are in the memoization cache and all workers should be able to finish quickly. Depending on the depth of the calculation, this may still require a significant amount of time.

Experimenting on the `schedule_world.3` model selecting the most expensive `RelProdS` operation, we measured the elapsed time between starting and finishing the operation for each worker. Then, we used a flag as a signal for each worker to abort when one worker has finished the operation. This flag was checked at the start of every `RelProdS` and `ITE` call and after every call to `RelProdS` and `ITE`. Using this flag comes at essentially no cost [31]. The results of this small experiment are in Table 6.5. For example, using 8 workers, the first worker was finished after 1359 time units, and the last worker after 1820 time units. The exact times can be different for every execution, since the workers execute suboperations in a random order. This randomness is significant when looking at one single operation, but is less significant when looking at reachability as a whole. What is clear from Table 6.5 is that without the flag, there is a significant random performance loss. With the flag, all operations finish at the same time.

Table 6.5: Results of using a kill flag for a single operation

Workers	Time to calculate	Time using flag
4	1889 - 1889	
8	1359 - 1820 (+34%)	1559 - 1559
24	900 - 1150 (+28%)	901 - 901
36	972 - 1065 (+9.5%)	959 - 959
48	1728 - 2069 (+21%)	1127 - 1127

Based on these results, we investigated whether using a kill flag improves performance. When looking at the times of the reachability operations on the `collision.5` model, we get the results of Table 6.6. These results are disappointing, since there is little difference between the times with multiple workers, where we would expect benefit. The small differences in times are expected measurement deviations.

Table 6.6: Results of using a kill flag for model `collision.5`

Workers	Time without kill flag (seconds)	Time with kill flag (seconds)
1	2306	2252
2	1271	1308
4	730	793
8	410	414
16	236	237
24	175	176
28	160	158
32	148	150
36	146	147
40	143	142
44	146	143
48	150	149

We compared all the results of using the kill flag for all models to the original result without the kill flag. See Figure 6.12. It appears that there is no significant benefit from using the kill flag, despite the expected improvement based on Table 6.5. This means that either only few operations exhibit the performance loss due to workers not finishing at the same time, or using a kill flag causes a performance loss negating the benefits. This could be investigated in the future using CPU profiling tools.

3) Similar to the experiment with Wool, memory alignment on the NUMA system may influence the performance of our algorithm. To see if memory alignment on the NUMA

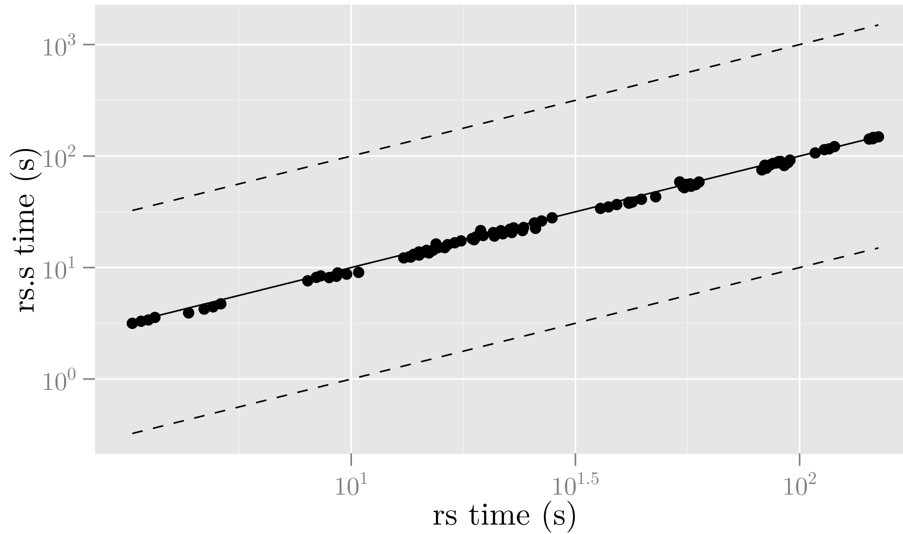


Figure 6.12: Results of using the kill flag (36-48 workers, all models)

system influences performance, we ran an experiment where the hash tables were allocated interleaved using the function `numa_alloc_interleaved` from the NUMA library.

The results (with at least 32 workers) are presented in Figure 6.13. From Figure 6.13 it is clear that there is no significant benefit of using the NUMA library instead of the standard allocator.

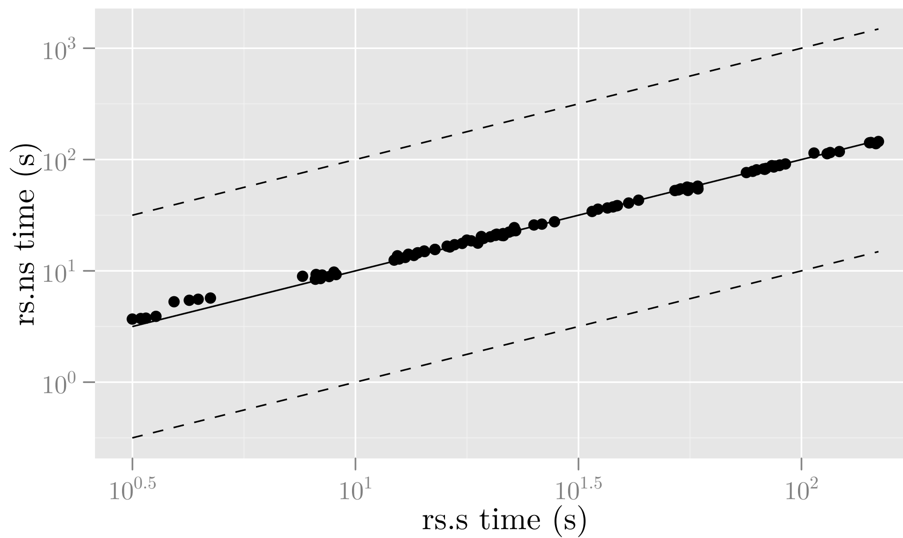


Figure 6.13: Results of using interleaved memory allocation (36-48 workers, all models)

4) Thread migration due to the Linux kernel sometimes allocating workers to different CPU cores during the experiment may cause performance degradation. In addition, migrating a worker also implies moving local data. This can be prevented by setting the CPU affinity of each worker. We tested this and the result is in Figure 6.14. It appears there is no benefit. Instead of a benefit, there is a significant performance loss on one model. We do not know what may cause this loss and since there seems to be no benefit in the other cases, we did not pursue this further.

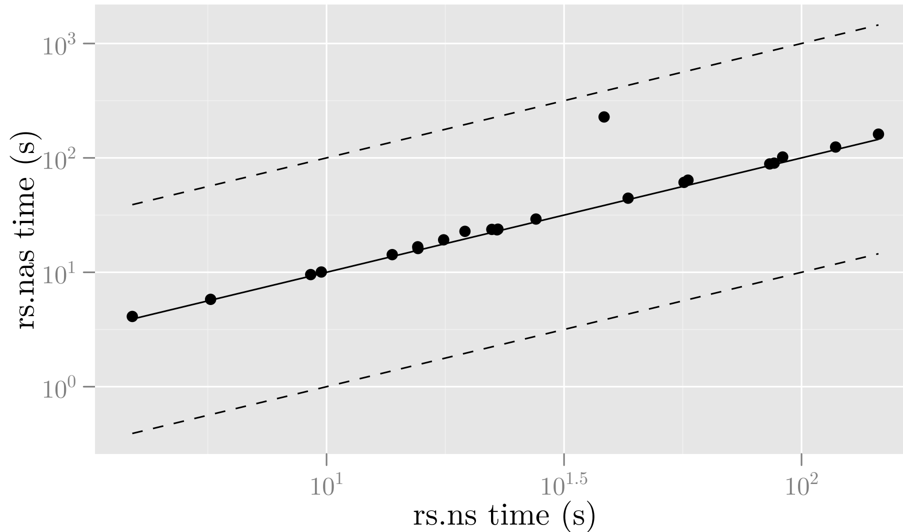


Figure 6.14: Results of setting CPU affinity (only 48 workers, all models)

Conclusions In conclusion, we found that memory allocation on the NUMA architecture, possible thread migration, and the overhead from the framework are not significant in limiting the speedups of our algorithm. There is a significant amount of redundant work (20-500%) that we believe is a result from limited parallelism. When parallelizing using result sharing limited parallelism results in redundant work. However, the redundant work caused by limited parallelism is not sufficient to explain lower speedups on smaller models. We need to investigate other mechanisms that cause reduced speedups due to limited parallelism. Figure 6.15 suggests a relation between highest speedup and the amount of work of each model. Figure 6.15 also suggests there are other factors influencing highest speedup.

As with Wool, we believe that the shape of the speedup graph is a result of the memory bandwidth requirements of the BDD operations when there is sufficient work to parallelize.

6.4 Comparing using result sharing to using Wool

Both approaches give similar results, as suggested by the results in Figure 6.7 and the results obtained using Wool in Figure 6.1. Conceptually, the approach using result sharing is based on all workers performing the same operations and reducing redundant work using result sharing and by calculating suboperations in a random order. The approach using Wool also uses result sharing, but eliminates all redundant work (except those caused by subgraph sharing) by explicitly distributing tasks among workers.

We also compared the results of using result sharing and using Wool directly. See Figures 6.16 to 6.19. Figures 6.16 to 6.19 show the comparison of the best result for each model using Wool and using result sharing and the comparison with 1 worker, 36 workers and 48 workers. As could be expected, the results with 1 worker are nearly identical, slightly in favor of result sharing since it lacks the overhead of Wool. We can conclude that the performance using Wool is up to 150% better (for the `lifts.6` model, see also Table 6.8) compared to the performance using result sharing in this experiment.

Compared to result sharing, Wool reduces the amount of redundant work while adding some overhead for task distribution. We can conclude that the benefit of reduced work is greater than the cost of the overhead. Although using Wool results in better performance, the results are similar. It may be possible to optimize the result sharing approach as well as optimize Wool for better results.

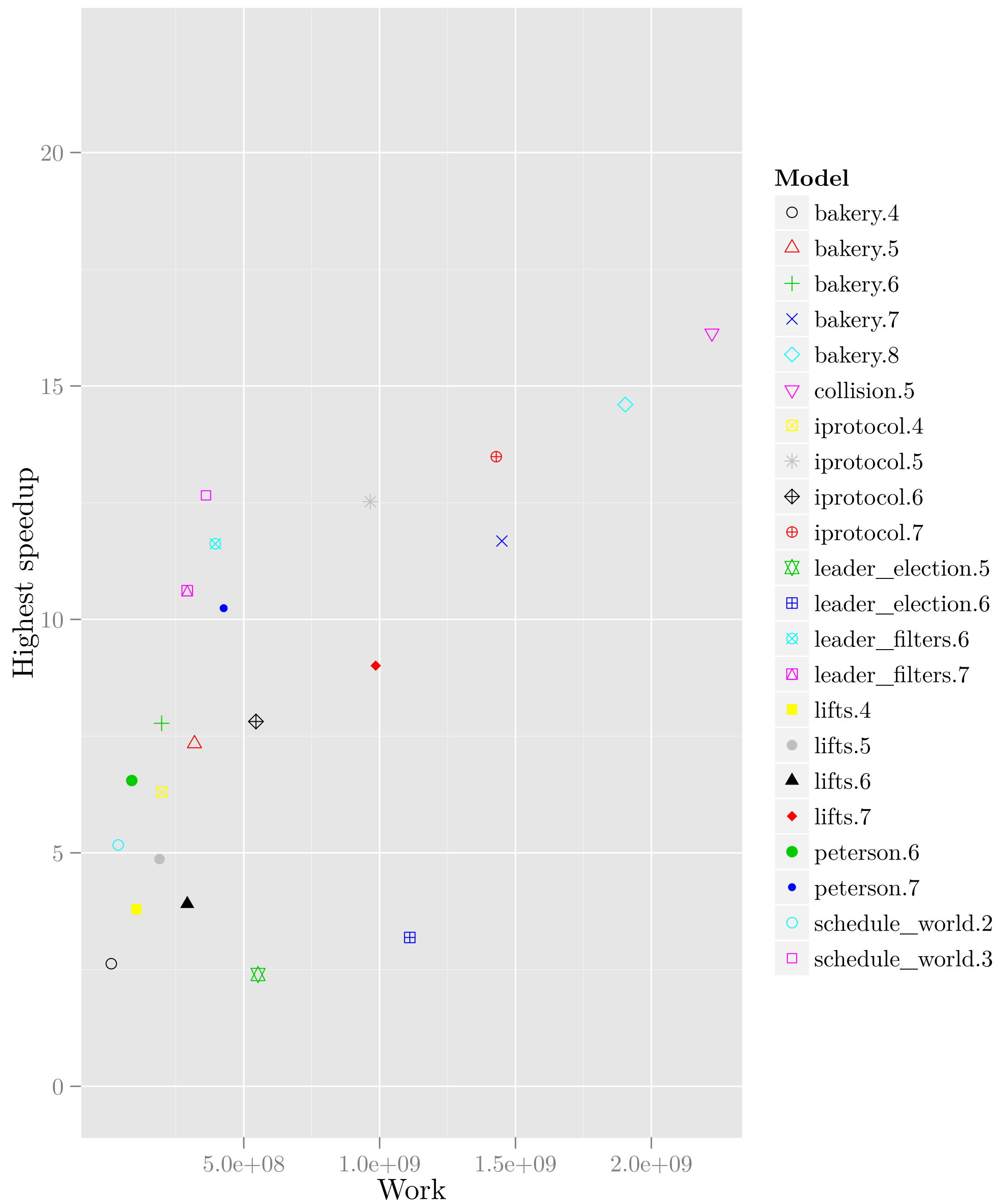


Figure 6.15: Best speedup for result sharing compared to work per model

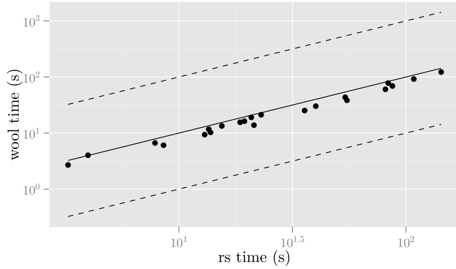


Figure 6.16: Result sharing vs Wool (best)

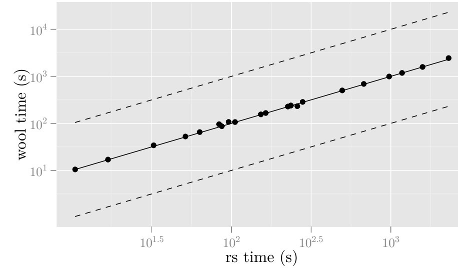


Figure 6.17: Result sharing vs Wool (1)

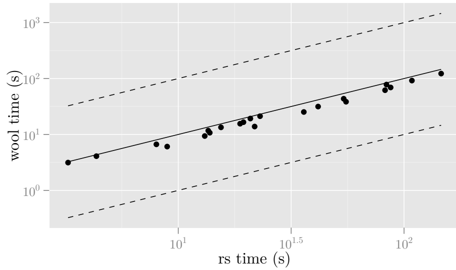


Figure 6.18: Result sharing vs Wool (36)

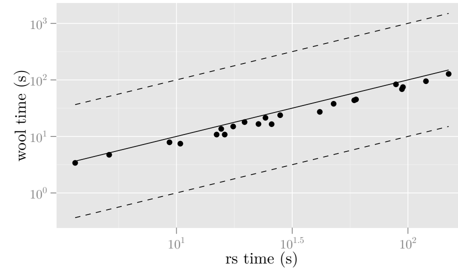


Figure 6.19: Result sharing vs Wool (48)

6.5 Changing memoization cache strategies

Using the memoization cache is expensive, since for every added entry, several `compare_and_swap` operations are necessary for getting locks and increasing reference counts. The memoization cache is necessary to prevent redundant calculations. However, the cost of performing a calculation twice may be less than the cost of using the memoization cache. If we don't use a memoization cache at all, the calculation and all subcalculation are recalculated, but if we only use the memoization cache when a certain property holds, for example when $x \in \{x_0, x_2, x_4, \dots\}$, this may result in a performance gain. In addition to the reduced cost of locking and manipulating reference counts, the size requirements for the memoization cache are significantly reduced.

We tested this hypothesis using a fairly straightforward algorithm. In this experiment, we use the memoization cache only when $x \in \{x_0, x_N, x_{2N}, x_{3N}, \dots\}$ where $N \in \{2, 3, 4\}$. See Figures 6.20 to 6.25 for the results using 1-4 workers and Figures 6.26 to 6.31 for the results using 32-48 workers. From these plots it is clear that there is some benefit for performance, but that scalability is not increased. The benefit is the same for a few workers as for many workers. See also Figure 6.32 for a speedup graph with $N = 4$. The calculation is performed faster, but it does not scale better.

We also compared the amount of work between N=1 and N=4. See Table 6.7. See also Figures 6.33 and 6.34. It is clear that there is some extra work, but this amount is small: from 0.9% up to 22.6% extra work.

6.6 Comparison to using BuDDy

Our goal is to use parallelism to improve the performance of BDD algorithms. It may be possible that our solution is inefficient with a single worker, resulting in possibly impressive speedups that are still slower than an optimized sequential algorithm. It is therefore useful to compare our results to the performance of other BDD implementations. We compared our results to the performance of BuDDy, which is one of the existing backends for symbolic model checking in the `LTSmin` toolset. The results (execution time in seconds) are in Table 6.8. In this table we used the data without a different cache strategy.

From these numbers we can see that BuDDy is 2 to 4 times faster than Sylvan for all models when using a single worker. One of the reasons for this is that BuDDy uses a mark-and-sweep garbage collection algorithm, instead of reference counting as in Sylvan. Since the memoization cache is cleared during garbage collection, there is no need to update reference

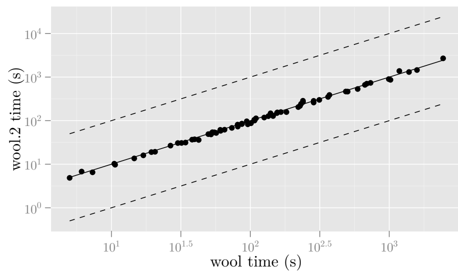


Figure 6.20: Wool with N=2 (1-4 workers)

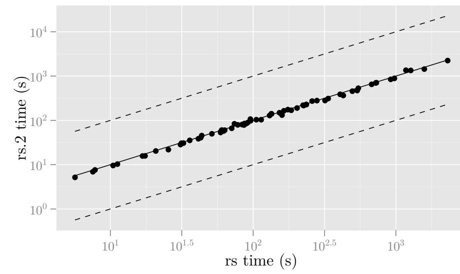


Figure 6.21: Result sharing with N=2 (1-4 workers)

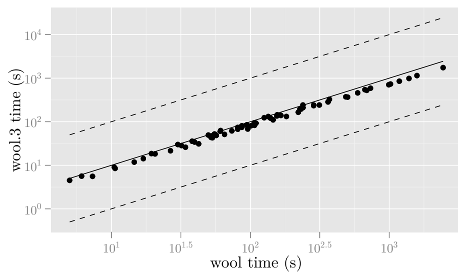


Figure 6.22: Wool with N=3 (1-4 workers)

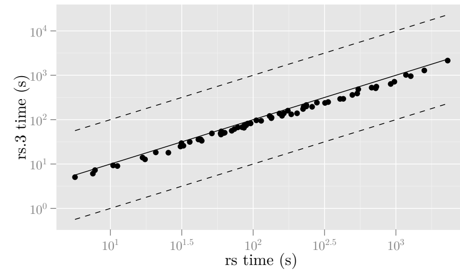


Figure 6.23: Result sharing with N=3 (1-4 workers)

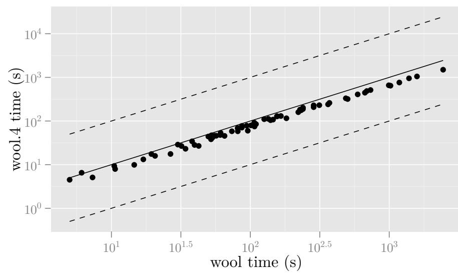


Figure 6.24: Wool with N=4 (1-4 workers)

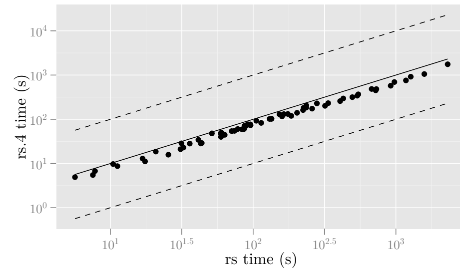


Figure 6.25: Result sharing with N=4 (1-4 workers)

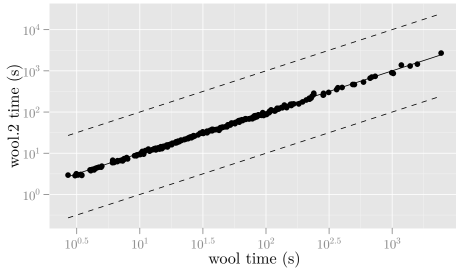


Figure 6.26: Wool with N=2 (36-48 workers)

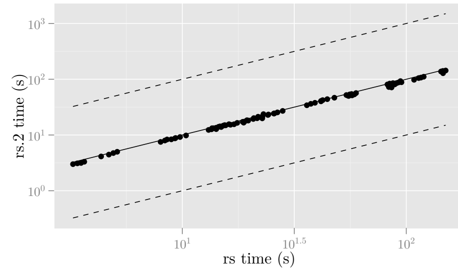


Figure 6.27: Result sharing with N=2 (36-48 workers)

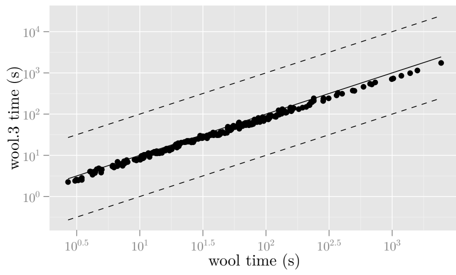


Figure 6.28: Wool with N=3 (36-48 workers)

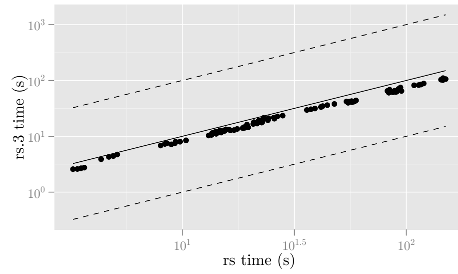


Figure 6.29: Result sharing with N=3 (36-48 workers)

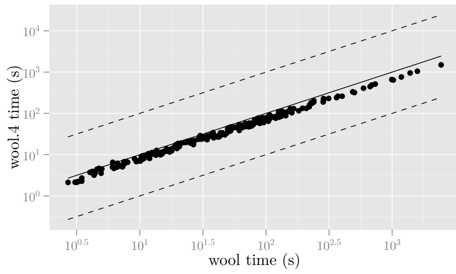


Figure 6.30: Wool with N=4 (36-48 workers)

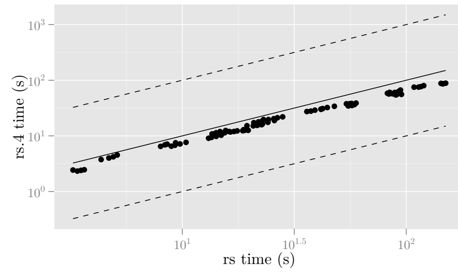


Figure 6.31: Result sharing with N=4 (36-48 workers)

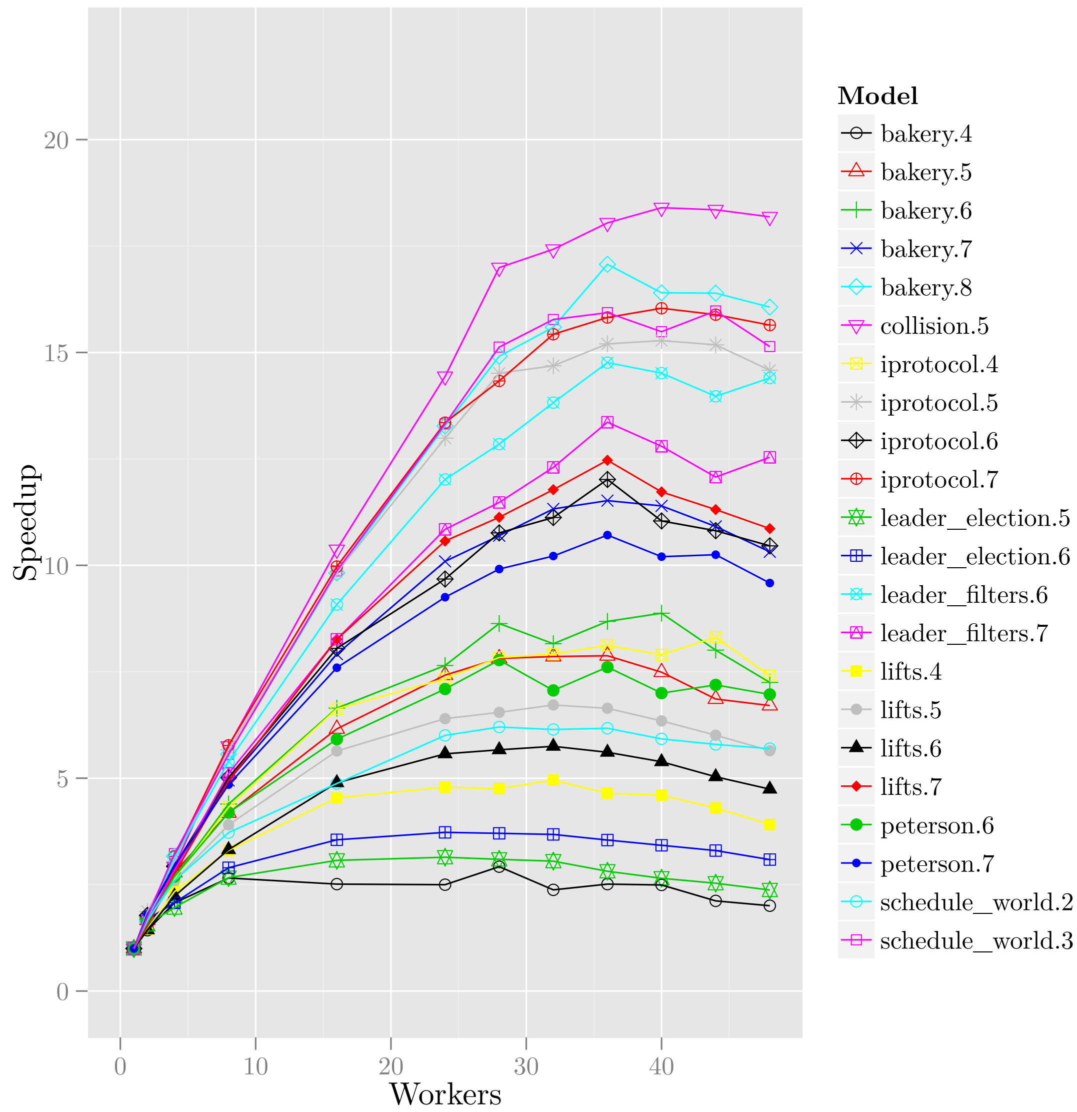


Figure 6.32: Wool speedup with N=4 (36-48 workers)

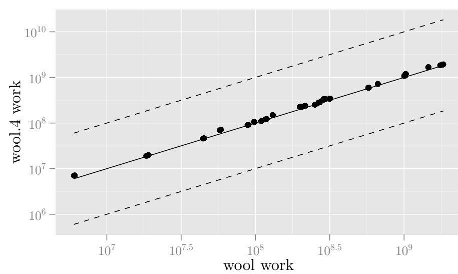


Figure 6.33: Wool extra work (N=4)

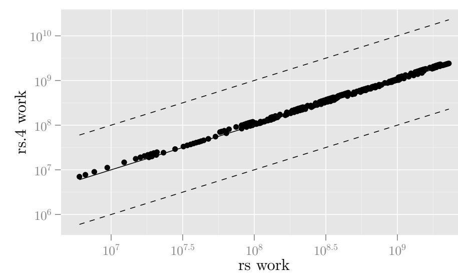


Figure 6.34: Result sharing extra work (N=4)

Table 6.7: Extra work for $N = 4$

Model	Wool extra work N=4	Result sharing extra work N=4
bakery.4	17.0%	19.7%
bakery.5	14.4%	17.5%
bakery.6	13.9%	15.9%
bakery.7	15.8%	16.1%
bakery.8	15.3%	15.6%
collision.5	5.5%	5.6%
iprotocol.4	8.1%	8.6%
iprotocol.5	8.0%	8.1%
iprotocol.6	8.3%	9.7%
iprotocol.7	7.6%	7.7%
leader_election.5	0.9%	1.1%
leader_election.6	0.9%	1.4%
leader_filters.6	13.1%	13.7%
leader_filters.7	10.4%	11.0%
lifts.4	3.1%	3.4%
lifts.5	2.9%	3.2%
lifts.6	3.4%	3.6%
lifts.7	3.2%	3.2%
peterson.6	21.6%	22.6%
peterson.7	15.8%	17.6%
schedule_world.2	3.8%	4.3%
schedule_world.3	5.5%	5.7%

Table 6.8: Results using BuDDy, Wool, result sharing

Model	BuDDy	Wool-1	Wool-best	RS-1	RS-best
bakery.4	1.87	10.48	4.03 (24)	10.45	3.98 (16)
bakery.5	73.71	155.34	19.05 (32)	152.92	20.84 (32)
bakery.6	44.77	106.71	11.69 (36)	105.28	13.54 (32)
bakery.7	319.83	992.61	77.91 (40)	976.26	83.59 (36)
bakery.8	517.66	1583.83	91.96 (36)	2051.18	97.71 (44)
collision.5	623.31	2443.36	122.58 (36)	2305.75	142.94 (40)
iprotocol.4	-	86.87	10.29 (32)	86.97	13.79 (36)
iprotocol.5	305.89	689.25	43.52 (36)	676.47	54.03 (36)
iprotocol.6	107.75	285.69	25.23 (36)	279.71	35.80 (32)
iprotocol.7	351.87	1182.47	69.24 (36)	1176.05	87.21 (36)
leader_election.5	-	107.22	30.96 (32)	96.00	40.10 (32)
leader_election.6	-	231.84	60.57 (32)	258.94	81.24 (32)
leader_filters.6	92.62	227.78	16.32 (32)	225.88	19.44 (36)
leader_filters.7	71.58	165.95	13.40 (36)	164.11	15.46 (36)
lifts.4	12.37	34.13	6.07 (36)	32.49	8.56 (28)
lifts.5	24.76	65.21	9.38 (36)	63.22	12.99 (32)
lifts.6	40.51	95.85	13.85 (36)	83.64	21.41 (32)
lifts.7	194.60	502.00	38.46 (26)	495.51	55.00 (40)
peterson.6	21.74	52.54	6.66 (36)	51.41	7.85 (32)
peterson.7	131.75	239.35	21.20 (36)	235.83	23.03 (36)
schedule_world.2	6.49	16.97	2.70 (28)	16.80	3.25 (36)
schedule_world.3	114.26	239.13	15.57 (36)	235.54	18.61 (40)

counts in BuDDy. Our current implementation of reference counting garbage collection requires that reference counts of BDD nodes are increased when results are added to the memoization cache. Therefore using the memoization cache is much cheaper in BuDDy. However, Sylvan uses a more optimal `RelProdS` algorithm. Also, Sylvan uses complement edges to reuse subgraphs, while BuDDy does not. On the other hand, BuDDy has been optimized for several years.

Table 6.8 also shows that for all models except `bakery.4` the parallel versions with sufficient workers are faster than BuDDy, showing that parallelizing BDD operations with both approaches results in better performance than the existing optimized BDD library BuDDy.

6.7 Conclusions

Both the approach using Wool and the approach using result sharing give promising results. The speedups using Wool are slightly better than the speedups using result sharing, but both methods are viable for the parallelization of BDD operations.

We partially understand the causes of limited speedups. We get the best performance in most cases with about 36 workers. With more than 36 workers the performance decreases. After excluding several possible explanations, we hypothesise that this is due to the memory bandwidth requirements of the BDD algorithms and lockless data structures. Improving the lockless data structures by reducing the number of `compare_and_swap` operations may reduce the bandwidth requirements of the algorithm as well as reduced the number of `compare_and_swap` operations compared to other memory transfers, and thus improve scalability in the future.

The actual scalability of the reachability algorithm greatly depends on the specific model. The reachability algorithm consists of a large number of smaller BDD operations, partially due to the division of the transition relation into multiple transition groups in `LTSmin`. The average scalability of these operations determines the scalability of the reachability algorithm. In future work, merging transition relations to increase scalability could be investigated.

We investigated the relation between the size of the models in terms of the number of non-trivial `RelProdS` and `ITE` suboperations and the scalability. Smaller operations often exhibit insufficient parallelism for all workers. Limited parallelism results in worker idling in `steal` and `do_work` functions when using Wool and in a higher amount of redundant work when using result sharing. With result sharing, the higher amount of redundant work is not sufficient to explain limited scalability. When we compensate the speedup plots for the extra work (see Figure 6.9), there is still a significant difference between models. We believe there are more mechanisms that cause worse performance due to limited parallelism or model features such as the size or shapes of involved BDDs. This could be investigated in the future.

We investigated whether bad memory allocation and thread migration could cause worse performance and determined that this is not the case.

We determined that when using Wool, redundant work due to subgraph sharing is not significant (only 7% in the worst case) in the models we used. With result sharing it is harder to establish how much redundant work was due to subgraph sharing. Subgraph sharing can be detected by using a magic value in the memoization cache indicating that a worker has started the subcalculation. In future work, we may be able to modify Wool to do other work when subgraph sharing is detected. We could also investigate methods to improve the performance of result sharing, for example backtracking when subgraph sharing is detected.

We found that using a flag to abort calculations when one worker has finished the calculation with result sharing did not improve the performance.

We studied the effects of selectively disabling the use of the memoization cache to improve performance and determined that in our experiments, while the overall performance improved, there was no effect on scalability.

Compared to BuDDy we have better performance when using multiple workers and lower performance when using one worker. In future work this could be improved by removing reference counting for entries in the memoization cache.

7

Related work

There are many different implementations of BDDs. Two well-known packages are BuDDy [30], which is used in the symbolic model checker NuSMV and in LTSmin, and CuDD [42] from Colorado University.

In the past (1995-1998) there have been several publications regarding the parallelization of BDD operations, for example [20, 44, 49, 34]. They used different architectures, such as massively parallel SIMD or distributed architectures. They had mixed results that are hard to translate to modern machines, because of the huge differences in design and processor and memory speeds.

Some of the research in that time was directed at using breadth-first approaches to improve memory locality during BDD traversal, which might be interesting to study as well. In a performance study of BDD-based model checking, Yang at al. [48] found that breadth-first expansion (in which sub-operations of the same variables are processed together) has performance drawbacks in two studied packages (CAL and PBF) such as increased memory overhead. In their study, they found no evidence that the breadth-first based packages are better than the depth-first based packages when the computation fits in main memory.

There have been at least three different initiatives in literature to implement parallel BDDs on modern multicore machines.

In an article published in 2009 on the website of Cilk and on the website of Intel, Yuxiong He presented the result of using the multicore framework Cilk++ to parallelize the apply operator in BuDDy [24]. See Figure 7.1(source: [24]). Their results are promising, but they did not provide a speedup graph relative to the single-threaded version. They only parallelized BuDDy as a proof of concept and only provided a small number of results. They suggest that limited linear speedup is due to memory constraints.

In his thesis in 2010, Jörn Ossowski presented the parallel BDD package JINC [36]. In JINC, the operations are not parallelized. Parallelism is only exploited on the high-level operator view. Each worker in JINC has its own computed tables and memory pools, eliminating communication overhead. To address the issue that this results in more lookup misses in the computed table, JINC implements a multi-operand Apply that reduces the need of computed tables, because it eliminates the need for temporary BDD nodes. Garbage collection and variable reordering are also implemented in parallel. JINC uses a worker pool with a limited number of workers, to reduce the amount of context switches. Workers request tasks from a shared queue and sleep until new tasks are available. The shared queue is implemented using mutual exclusion, based on the boost framework for C++.

Ossowski mentions that he did not investigate parallelizing the BDD operators for a number of reasons. The primary reason Ossowski states is that the unique tables and computed tables are the major obstacle to implement an efficient parallel BDD library and that Cilk's approach is based on independent calculations with few or no shared memory [36]. As we demonstrated, a lockless data structure helps solve this issue. Our lockless data structures may also remove the need to create a different computed table for every worker.

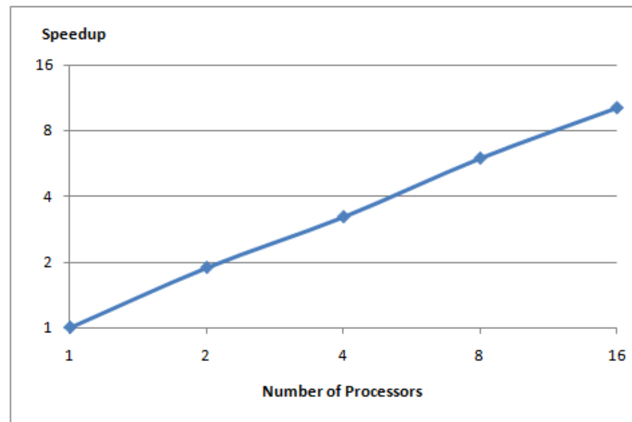


Figure 7.1: Speedup graph of 13-bit Multiplier Construction using Cilk

Finally, Sahoo et al. presented a multithreaded solution that used partitioned BDDs [40]. They partition the BDDs into smaller BDDs and calculate the individual operations. Grumberg et al. [21] use a similar method to partition BDDs to distribute the operations between multiple machines, but their results cannot be compared since their solution is tailored towards distributed architectures while ours targets multicore architectures.

8

Conclusions and Future Work

In this chapter we present the conclusions of our work and some ideas for future work.

8.1 Conclusions

In Chapter 3, we designed and implemented a new specialized algorithm `RelProdS` to calculate the successors of a set of states and an algorithm `RRelProdS` that calculates the predecessors of a set of states, given a transition relation. We proved that these operations are correct. These algorithms reduce the number of BDD nodes created to calculate the successors and predecessors compared to the use of the individual algorithms `RelProd` and `Substitute` as in implementations of model checkers using BDD packages BuDDy [30] and CuDD [42].

In Chapter 4, we extended Amdahl's Law as a qualitative theoretical approach to parallel performance, to include the critical path of the calculation and to include the communication overhead due to adding workers. Our theoretical model suggests that there is a peak in speedup graphs when the communication overhead of adding a worker is equal to the performance gain of adding a worker, after which the speedup decreases.

In Chapter 5, we implemented a lockless lossy memoization cache. We also implemented a lockless hashtable with garbage collection using reference counting. Every bucket also stores a reference count and when a reference count is decreased to 0, the bucket index is inserted into a buffer (implemented using the lockless cache) that facilitates lazy garbage collection. Both data structures use the `compare_and_swap` atomic processor instruction to manipulate local short-lived locks.

We implemented a framework for result sharing.

In Chapter 6, we parallelized the BDD operations `ITE` and `RelProdS` using the framework Wool and using our result sharing framework. We measured the performance of both parallelized algorithms by performing reachability calculations on a number of models from the BEEM database [37] using the `LTSmin` toolset [9] on a NUMA architecture with 48 processor cores. Both approaches give promising results and are viable methods to parallelize BDD operations. With sufficient workers, we get a speedup of up to 20 using Wool and 17 using result sharing, depending on the input model. Compared to BuDDy we have higher performance when using multiple workers and lower performance when using one worker. The speedup using Wool is in almost all models better than the speedup using result sharing. We get the best performance in most cases with about 36 workers. With more than 36 workers the performance decreases.

We explored several explanations for the differences in speedup and the decrease in speedup after 36 workers. We found that in smaller models there is limited parallelization, which is expressed in the approach using Wool as more time spent in Wool functions `do_work` and `steal`, and in the approach using result sharing as more redundant work.

After excluding several possible explanations, we hypothesise that a major factor is the memory bandwidth requirements of the BDD algorithms and lockless data structures.

The actual scalability of the reachability algorithm greatly depends on the specific model. The reachability algorithm consists of hundreds of smaller BDD operations, partially due to the division of the transition relation into multiple transition groups in `LTSmin`. The average scalability of these operations determines the scalability of the reachability algorithm.

There is some redundant work due to subgraph sharing when multiple workers calculate the same suboperation before any worker has calculated the result, but we found that the amount of redundant work is negligible in our experiments.

We studied the effects of using different result memoization strategies and found that the performance improves for all number of workers if we sometimes do not use the memoization cache, but that there is little effect on scalability. We measured a speedup of up to 1.8 by reducing the use of the memoization cache by 25%.

8.2 Future Work

We have several ideas for future investigations:

1) Both lockless data structures can be improved. Improving the lockless data structures by reducing the number of `compare_and_swap` operations may reduce the bandwidth requirements of the algorithm as well as reduced the number of `compare_and_swap` operations compared to other memory transfers, and thus improve scalability.

2) We could modify our implementation of garbage collection such that the memoization cache is completely emptied prior to garbage collection. This removes the need to update reference counts when using the memoization cache.

3) It would be useful to study the speedup of the individual BDD operations in the reachability operation rather than the full reachability algorithm. More theoretical analysis, for example the critical path length of the individual BDD operations, may also provide more insight, especially for models like `leader_election`.

4) We do not fully understand the exact mechanisms how certain models only give low speedups. This should also be further investigated. For example, using CPU profiling we could find out which operations are relative bottlenecks during the execution of the experiments.

5) We could merge transition groups in `LTSmin` to increase the size of BDD operations and thus hopefully improve the scalability.

6) We could calculate the `RelProdS` operations of the different transition groups in `LTSmin` in parallel, for example for 6 transition groups using 8 workers per operation.

7) We should investigate why using a kill flag with the result sharing approach has limited benefit, despite our initial research on an isolated BDD operation suggesting that there would be a benefit.

8) We may be able to modify Wool to do other work when there is subgraph sharing. We could also investigate methods to improve the performance of result sharing, for example backtracking when subgraph sharing is detected.

9) We should implement `RelProd` and `Substitute` in Sylvan and measure the effect of using `RelProdS` instead of using `RelProd` and `Substitute` separately when calculating the set of next states in model checking. We should also extend `LTSmin` to support CuDD, so the parallel performance of Sylvan can be compared to an additional BDD library.

Bibliography

[1] http://lse.sourceforge.net/numa/faq/.

[2] http://code.google.com/p/gperftools/.

[3] http://fmt.cs.utwente.nl/tools/ltsmin/.

[4] A. Aggarwal, C. Rangan, and P. Sanders, *Asynchronous random polling dynamic load balancing*, in Algorithms and Computation, vol. 1741 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1999, pp. 37–48.

[5] S. Akers, *Binary Decision Diagrams*, Computers, IEEE Transactions on, C-27 (1978), pp. 509 –516.

[6] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, in Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring), New York, NY, USA, 1967, ACM, pp. 483–485.

[7] D. Bader, V. Kanade, and K. Madduri, *Swarm: A parallel programming framework for multicore processors*, in Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, march 2007, pp. 1–8.

[8] R. Belllman, *The theory of dynamic programming*, 1954.

[9] S. Blom, J. van de Pol, and M. Weber, *LTSmin: distributed and symbolic reachability*, in Proceedings of the 22nd international conference on Computer Aided Verification, CAV'10, Berlin, Heidelberg, 2010, Springer-Verlag, pp. 354–359.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: an efficient multithreaded runtime system*, in Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPOPP '95, New York, NY, USA, 1995, ACM, pp. 207–216.

[11] K. Brace, R. Rudell, and R. Bryant, *Efficient implementation of a BDD package*, in Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE, jun 1990, pp. 40 –45.

[12] R. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, Computers, IEEE Transactions on, C-35 (1986), pp. 677 –691.

[13] R. E. Bryant, *Symbolic Boolean manipulation with ordered binary-decision diagrams*, ACM Comput. Surv., 24 (1992), pp. 293–318.

[14] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill, *Symbolic model checking for sequential circuit verification*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 13 (1994), pp. 401 –424.

[15] Q. Chen, Z. Huang, M. Guo, and J. Zhou, *CAB: Cache Aware Bi-tier Task-stealing in Multi-socket Multi-core Architecture*, in Proceedings of International Conference on Parallel Processing (ICPP) 2011, 2011.

[16] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, *NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking*, in Proc. International Conference on Computer-Aided Verification (CAV 2002), vol. 2404 of LNCS, Copenhagen, Denmark, July 2002, Springer.

[17] L. Dagum and R. Menon, *Openmp: an industry standard api for shared-memory programming*, Computational Science Engineering, IEEE, 5 (1998), pp. 46 –55.

[18] R. Drechsler and D. Sieling, *Binary decision diagrams in theory and practice*, International Journal on Software Tools for Technology Transfer (STTT), 3 (2001), pp. 112–136.

[19] K.-F. Faxen, *Efficient work stealing for fine grained parallelism*, Parallel Processing, International Conference on, (2010), pp. 313–322.

[20] S. Gai, M. Rebaudengo, and M. Reorda, *A data parallel algorithm for boolean function manipulation*, in Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers '95., Fifth Symposium on the, feb 1995, pp. 28 –34.

[21] O. Grumberg, T. Heyman, and A. Schuster, *A work-efficient distributed algorithm for reachability analysis*, in Computer Aided Verification, W. Hunt and F. Somenzi, eds., vol. 2742 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2003, pp. 54–66.

[22] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, *SLAW: A scalable locality-aware adaptive work-stealing scheduler*, in Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, april 2010.

[23] J. L. Gustafson, *Reevaluating amdahl's law*, Commun. ACM, 31 (1988), pp. 532–533.

[24] Y. He, *Multicore-enabling a Binary Decision Diagram algorithm*, May 2009.

[25] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[26] G. J. Holzmann, R. Joshi, and A. Groce, *Swarm verification*, in Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, Washington, DC, USA, 2008, IEEE Computer Society, pp. 1–6.

[27] G. Janssen, *Design of a pointerless BDD package*, (2001).

[28] O. Kupferman, M. Y. Vardi, and P. Wolper, *An automata-theoretic approach to branching-time model checking*, in JOURNAL OF THE ACM, Springer-Verlag, 1998, pp. 142–155.

[29] A. Laarman, J. P. van de, and M. Weber, *Boosting multi-core reachability performance with shared hash tables*, April 2010. Technical Report of paper published at FMCAD 2010 with the same name.

[30] J. Lind-Nielsen, *BuDDy: A Binary Decision Diagram library.* http://buddy.sourceforge.net.

[31] J. M. Mellor-Crummey and M. L. Scott, *Algorithms for scalable synchronization on shared-memory multiprocessors*, ACM Trans. Comput. Syst., 9 (1991), pp. 21–65.

[32] D. Miller and R. Drechsler, *Negation and duality in reduced ordered binary decision diagrams*, in Communications, Computers and Signal Processing, 1997. '10 Years PACRIM 1987-1997 - Networking the Pacific Rim'. 1997 IEEE Pacific Rim Conference on, vol. 2, aug 1997, pp. 692 –696 vol.2.

[33] ——, *Dual edge operations in reduced ordered binary decision diagrams*, in Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on, vol. 6, may-3 jun 1998, pp. 159 –162 vol.6.

[34] K. Milvang-Jensen and A. Hu, *Bddnow: A parallel bdd package*, in Formal Methods in Computer-Aided Design, G. Gopalakrishnan and P. Windley, eds., vol. 1522 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1998, pp. 532–532.

[35] S.-i. Minato, N. Ishiura, and S. Yajima, *Shared binary decision diagram with attributed edges for efficient Boolean function manipulation*, in Proceedings of the 27th ACM/IEEE Design Automation Conference, DAC '90, New York, NY, USA, 1990, ACM, pp. 52–57.

[36] J. Ossowski, *JINC - A Multi-Threaded Library for Higher-Order Weighted Decision Diagram Manipulation*, Bonn, 2010.

[37] R. Pelánek, *BEEM: benchmarks for explicit model checkers*, in Proceedings of the 14th international SPIN conference on Model checking software, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 263–267.

[38] A. Podobas, M. Brorsson, and K.-F. Faxen, *A comparison of some recent task-based parallel programming models*, 3rd Workshop on Programmability Issues for Multi-Core Computers, (2010).

[39] J. A. Roback and G. R. Andrews, *Gossamer: a lightweight programming framework for multicore machines*, in Proceedings of the 2nd USENIX conference on Hot topics in parallelism, HotPar'10, Berkeley, CA, USA, 2010, USENIX Association, pp. 14–14.

[40] D. Sahoo, J. Jain, S. K. Iyer, D. L. Dill, and E. A. Emerson, *Multi-threaded reachability*, in Proceedings of the 42nd annual Design Automation Conference, DAC '05, New York, NY, USA, 2005, ACM, pp. 467–470.

[41] C. E. Shannon, *A Symbolic Analysis of Relay and Switching Circuits*, American Institute of Electrical Engineers, Transactions of the, 57 (1938), pp. 713 –723.

[42] F. Somenzi, *Cudd : Colorado university decision diagram package, release 2.3.1*. http://vlsi.colorado.edu/~fabio/CUDD/.

[43] F. Somenzi, *Efficient manipulation of decision diagrams*, International Journal on Software Tools for Technology Transfer (STTT), 3 (2001), pp. 171–181.

[44] T. Stornetta and F. Brewer, *Implementation of an efficient parallel bdd package*, in Proceedings of the 33rd annual Design Automation Conference, DAC '96, New York, NY, USA, 1996, ACM, pp. 641–644.

[45] H. Sutter, *The free lunch is over: A fundamental turn toward concurrency in software*, Dr. Dobb's Journal, (2005).

[46] M. Y. Vardi, *An automata-theoretic approach to linear temporal logic*, in Logics for Concurrency: Structure versus Automata, volume 1043 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 238–266.

[47] S. Williams, A. Waterman, and D. Patterson, *Roofline: an insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76.

[48] B. Yang, R. Bryant, D. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. Ranjan, and F. Somenzi, *A Performance Study of BDD-Based Model Checking*, in Formal Methods in Computer-Aided Design, G. Gopalakrishnan and P. Windley, eds., vol. 1522 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1998, pp. 533–533.

[49] B. Yang and D. R. O'Hallaron, *Parallel breadth-first bdd construction*, in Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPOPP '97, New York, NY, USA, 1997, ACM, pp. 145–156.