Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 296 (2013) 127–143

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Multi-Core BDD Operations for Symbolic Reachability

Tom van Dijk[1], Alfons Laarman[1] and Jaco van de Pol[1]

Formal Methods and Tools, Dept. of EEMCS, University of Twente
P.O.-box 217, 7500 AE Enschede, The Netherlands

Abstract

This paper presents scalable parallel BDD operations for modern multi-core hardware. We aim at increasing the performance of reachability analysis in the context of model checking. Existing approaches focus on performing multiple independent BDD operations rather than parallelizing the BDD operations themselves. In the past, attempts at parallelizing BDD operations have been unsuccessful due to communication costs in shared memory.
We solved this problem by extending an existing lockless hashtable to support BDDs and garbage collection and by implementing a lockless memoization table. Using these lockless hashtables and the work-stealing framework Wool, we implemented a multi-core BDD package called Sylvan.
We provide the experimental results of using this multi-core BDD package in the framework of the model checker LTSmin. We measured the runtime of the reachability algorithm on several models from the BEEM model database on a 48-core machine, demonstrating speedups of over 30 for some models, which is a breakthrough compared to earlier work.
In addition, we improved the standard symbolic reachability algorithm to use a modified BDD operation that calculates the relational product and the variable substitution in one step. We show that this new algorithm improves the performance of symbolic reachability and decreases the memory requirements by up to 40%.

Keywords: multi-core, BDD, symbolic reachability, parallel model checking, lockless hashtable, garbage collection, LTSmin, WOOL, Sylvan

1 Introduction

In model checking, we create abstractions of complex systems to verify that they function according to certain properties. Systems are modelled as a set of possible states the system can be in and a set of transitions between these states. States and transitions form a *transition system* that describes system behavior. The core of model checking is the *reachability* algorithm, which calculates all reachable states, i.e., all possible states a system can be in, based on the initial states and the transitions.

One major problem in model checking is the size of the transition system. Even with small systems, the memory required to store all explored states increases

[1] Email: {tdijk,a.w.laarman,vdpol}@cs.utwente.nl.
The first author is supported by the NWO project MaDriD, grant nr. 612.001.101

http://dx.doi.org/10.1016/j.entcs.2013.07.009

exponentially. One way to deal with this is to represent all states using Boolean functions, instead of storing them individually. This is called *symbolic model checking* [7]. Boolean functions can be stored in memory efficiently using Binary Decision Diagrams (BDDs) [1,6].

To manipulate Boolean functions stored using BDDs a large variety of BDD algorithms exist. To calculate all reachable states, only four algorithms are necessary: $\wedge$, $\vee$, $\exists$ and *variable substitution*. Common BDD implementations also include a special algorithm to calculate the *relational product* which combines $\wedge$ and $\exists$. The first contribution of this paper is a new algorithm that combines this relational product with variable substitution. With experiments we show that this algorithm is faster and requires up to 40% less memory than performing the two operations separately.

Since model checking has huge computational requirements, techniques that increase the performance of model checking tools are constantly being developed. 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. Recent developments in hardware introduce the necessity of multi-core and multi-processor architectures for future performance gains. In order to use the computational power of all cores we need to parallelize our software, i.e., divide algorithms into smaller parts that can be executed in parallel by multiple workers to achieve maximum speedup. In the literature, limited speedups for BDD operations have been attributed to the irregular memory access pattern. Symbolic state-space generation results in high parallel overhead, due to load imbalance and the scheduling of many small computations. Also, synchronisation on the symbolic data structure [16] incurs extra overhead. To maximize speedup we need to minimize this overhead by developing new data structures and algorithms.

The second contribution of this paper is Sylvan, a multi-core implementation of BDD algorithms using the task-based work-stealing framework Wool and scalable data structures that we developed. These data structures are based on the *lockless* paradigm, which avoids mutual exclusion and depends on atomic operations. We have performed experiments with state-space generation on several models from the BEEM database [31] using the `LTSmin` toolset [4] extended to support our experimental BDD package Sylvan. We obtain a speedup of up to 32 on 48 cores with the best benchmark model (average of 5 runs) relative to the runtime on 1 core. We compared the results to the performance of the same reachability algorithm using the popular BDD package BuDDy as the backend for symbolic model checking. The results show that compared to an optimized sequential package, our approach still gives a significant speedup of up to 12 times on 48 cores.

This paper is structured as follows. We summarize preliminaries on BDDs and reachability in Section 2 and present a new BDD algorithm `RelProdS` that reduces the memory requirements of symbolic model checking in Section 3. Section 4 discusses two approaches to parallelizing the BDD operations and we present a lockless memoization table and a lockless hashtable that supports garbage collection

with reference counting in Section 5. In Section 6 we present our experimental results. We finish this paper with related work and conclusions in Section 7 and Section 8.

2 Preliminaries

2.1 Symbolic reachability using Boolean functions

Let $S = \mathbb{B}^n$ be the set of all states, consisting of vectors of n Booleans. A transition relation is a binary relation $R \subseteq S \times S$, representing transitions between states. A transition system given vector size n is a pair (S_I, R), where $S_I \subseteq S$ is a set of initial states and $R \subseteq S \times S$ is a transition relation. The set of reachable states is the reflexive, transitive closure of R applied to S_I.

Generally, sets of states are either stored *explicitly*, i.e., every state is stored individually, or *symbolically*, i.e., the set of states is represented by a Boolean function [7]. A subset $V \subseteq S$ can be denoted by a Boolean function $F\colon \mathbb{B}^n \to \mathbb{B}$, such that, given a state s, $F(s) \Leftrightarrow s \in V$. The transition relation $R \subseteq S \times S$ can be denoted by a Boolean function $T\colon \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$, such that, given states s and s', $T(s, s') \Leftrightarrow (s, s') \in R$.

Given Boolean functions $F(s)$ and $T(s, s')$, the T-successors of F are obtained as $F'(s) = \exists s'.\, F(s') \wedge T(s', s)$. The set of reachable states is computed with symbolic breadth-first search as the fixed point of the following series:

$$F_{i+1}(s) = F_i(s) \vee (\exists s'.\, F_i(s') \wedge T(s', s)) \tag{1}$$

Given state vector s, we write $s[i \leftarrow v]$ for the vector equal to s, except $s_i = v$. With $\overline{s_i}$ we denote $s_i = 0$. We define the *restriction* (also called *cofactor*) of a function as $F_{i=v}(s) \stackrel{\text{def}}{=} F(s[i \leftarrow v])$. The following identity is known as Shannon's expansion [35].

$$F(s) \iff \big((s_i \wedge F_{i=1}(s)) \vee (\overline{s_i} \wedge F_{i=0}(s))\big) \tag{2}$$

2.2 Binary decision diagrams

Binary decision diagrams (BDDs) were introduced by Akers [1] and developed by Bryant [6]. Their major advantage is that sets of states are often concisely represented. In addition, since reduced ordered BDDs are canonical, testing equality of two sets is trivial.

A BDD is a directed acyclic graph with leaves 0 and 1, and a set of internal vertices V, equipped with a variable label and two outgoing edges. So BDDs are defined as tuples $(V, high, low, var)$, where $high, low\colon V \to V \cup \{0, 1\}$ are functions representing the high and low edges of a node, and var indicates the variable associated to a vertex. Every node in a BDD represents a Boolean function according to its Shannon expansion (2). In particular, if $var(B) = x$, $high(B) = B_1$ and $low(B) = B_0$, then

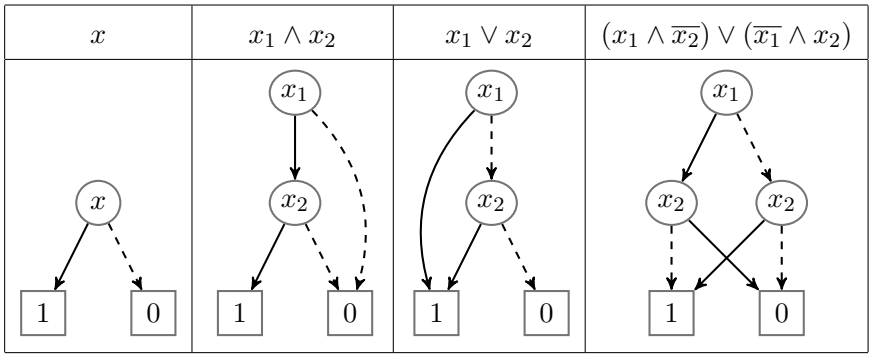

Figure 1. *Binary decision diagrams* for some Boolean functions. Internal nodes are drawn as circles with variables, and leaves as boxes. High edges are drawn solid, and low edges are drawn dashed.

B represents the function F, such that $F_{x=1}$ represents B_1 and $F_{x=0}$ represents B_0. Examples of simple BDDs are given in Figure 1.

Given a total ordering $<$ of the variables, an *ordered BDD* is a BDD in which the variables occur in increasing order along all paths from root to leaf. An ordered BDD is called *reduced*, if it has no redundant nodes (with two identical children), and no duplicate nodes (with the same variable, high and low edges). All examples in Figure 1 are ordered and reduced. Reduced and ordered BDDs are *canonical* representations of Boolean functions.

Implementation. BDD nodes are stored using memory arrays. An edge or reference to a BDD is the index in that memory array [22]. A single BDD node consists of three integers, representing the variable and the outgoing edges.

A BDD package must ensure the invariant that BDDs are reduced and ordered all the time. To this end, BDD implementations typically contain a method $\texttt{MK}(x, T, F)$ that returns a unique BDD node with variable x, a high outgoing edge to BDD T and a low outgoing edge to BDD F. This function guarantees that the returned BDD is a reduced BDD. To implement `MK`, a *Unique Table* is necessary, usually implemented using a hashtable. Alternatively, one can also store the nodes in this hashtable, eliminating the node array. This simplifies the implementation.

Garbage collection is essential for BDDs. Modifying a subgraph in a BDD typically implies modifying all ancestors, since BDD nodes are usually immutable. Therefore, BDD operations modify entire BDDs. The consequence is that the data structures used to store BDDs need to support garbage collection, for example using reference counting or mark-and-sweep approaches. However, Somenzi mentions that unused BDD nodes are often reused later and that garbage collection should only be performed when there are enough unused BDD nodes to justify the cost of garbage collection and recreating nodes that were deleted during garbage collection [36].

2.3 Relational product

The set of successors $F'(s) = \exists s'.F(s') \wedge T(s', s)$ in Equation (1) is usually computed in two steps. The starting point are BDDs $F(X)$ and $T(X, X')$. First, the BDD algorithm `RelProd` efficiently combines conjunction and existential quantification,

to obtain a BDD representing $\exists X.\, F(X) \wedge T(X, X')$. Note that this BDD is phrased in variables X'. In the second step, the variables X are substituted for X'. As a consequence, the BDD is created twice, using different sets of variables.

Definition 2.1 [`RelProd` algorithm] Given a set of variables $X_n = \{x_1, \ldots, x_n\}$, a set $X_\exists \subseteq X_n$, and BDDs $F(X)$ and $G(X)$, the `RelProd` algorithm returns a BDD $R(X_n \setminus X_\exists)$, representing

$$R(X_n \setminus X_\exists) = \exists X_\exists \big(F(X_n) \wedge G(X_n)\big)$$

A simplified (non-optimized) implementation of this algorithm is given in Algorithm 1. Here x is a variable, and X is a collection of variables. In l. 9, when $x \in X$, we compute $\exists x\, R$ as the disjunction $R_{x=0} \vee R_{x=1}$. When $x \notin X$, the result is calculated as a BDD with a root node with variable x.

Algorithm 1 `RelProd`: Calculate $\exists X(F \wedge G)$

Input: BDD F, BDD G, Set X

1: **if** $F = 1 \wedge G = 1$ **then return** 1
2: **if** $F = 0 \vee G = 0$ **then return** 0
3: **if** `memo.get`(F, G, X, R) **then return** R
4: $x \leftarrow$ `first`(`var`(F), `var`(G))
5: $\langle F_0, F_1 \rangle \leftarrow$ **if** $x =$ `var`(F) **then** $\langle$`low`(F), `high`$(F)\rangle$ **else** $\langle F, F \rangle$
6: $\langle G_0, G_1 \rangle \leftarrow$ **if** $x =$ `var`(G) **then** $\langle$`low`(G), `high`$(G)\rangle$ **else** $\langle G, G \rangle$
7: $R_0 \leftarrow$ `RelProd`(F_0, G_0, X)
8: $R_1 \leftarrow$ `RelProd`(F_1, G_1, X)
9: **if** $x \in X$ **then** $R \leftarrow R_0 \vee R_1$ **else** $R \leftarrow$ `MK`(x, R_1, R_0)
10: `memo.put`(F, G, X, R)
11: **return** R

Dynamic programming is used to make the algorithm polynomial in the size of the input BDDs. To this end, `memo.get` and `memo.put` (l. 3, 10) manipulate the memoization table, which is used to store all intermediate results for later reference. `low` and `high` follow the low and high edges of a BDD node, `var` returns the variable of a BDD node, `first` returns the first variable according to $<$ and `MK` is the method that creates or retrieves unique BDD nodes.

3 Improving reachability using `RelProdS`

We present a new algorithm that combines the relational product and substitution, eliminating the unnecessary creation of the BDD in X'. It is a modification of the original `RelProd` algorithm. We use a variable substitution (an injective function $S\colon X \to X$) which is directly applied when creating the BDD nodes of the result.

Note that in MDD-based model checking in SMART [11], as described elsewhere [13], the creation of these unnecessary BDD nodes is already avoided by storing normal and primed variables in the transition relation together and evaluating them in one step. Our solution is more general, allowing any substitution S as long as it preserves $<$.

Definition 3.1 [`RelProdS` algorithm] The `RelProdS` takes as input BDDs F and G, a set of variables X, a set of variables $X_\exists \subseteq X$, and an injective function $S\colon X \to X$,

Algorithm 2 `RelProdS`: Calculate $\exists X(F \wedge G)$ and apply substitution S

Input: BDD F, BDD G, Set X, Substitution S

1: **if** $F = 1 \wedge G = 1$ **then return** 1
2: **if** $F = 0 \vee G = 0 \vee F = \texttt{complement}(G)$ **then return** 0
3: **if** $G = 1$ **then return** $\texttt{RelProdS}(1, F, X, S)$
4: **if** $F = G$ **then return** $\texttt{RelProdS}(1, G, X, S)$
5: **if** $F > G$ **then**
6: **return** $\texttt{RelProdS}(G, F, X, S)$
7: **if** $\texttt{memo.get}(F, G, X, S, R)$ **then return** R
8: $x \leftarrow \texttt{first}(\texttt{var}(F), \texttt{var}(G))$
9: $\langle F_0, F_1 \rangle \leftarrow$ **if** $x = \texttt{var}(F)$ **then** $\langle \texttt{low}(F), \texttt{high}(F) \rangle$ **else** $\langle F, F \rangle$
10: $\langle G_0, G_1 \rangle \leftarrow$ **if** $x = \texttt{var}(G)$ **then** $\langle \texttt{low}(G), \texttt{high}(G) \rangle$ **else** $\langle G, G \rangle$
11: **if** $x \in X$ **then**
12: $R_0 \leftarrow \texttt{RelProdS}(F_0, G_0, X, S)$
13: **if** $R_0 = 1$ **then**
14: $R \leftarrow 1$
15: **else**
16: $R_1 \leftarrow \texttt{RelProdS}(F_1, G_1, X, S)$
17: $R \leftarrow R_0 \vee R_1$
18: **else**
19: $R_0 \leftarrow \texttt{RelProdS}(F_0, G_0, X, S)$
20: $R_1 \leftarrow \texttt{RelProdS}(F_1, G_1, X, S)$
21: $R \leftarrow \texttt{MK}(S(x), R_1, R_0)$
22: $\texttt{memo.put}(F, G, X, S, R)$
23: **return** R

which preserves the variable ordering $<$. `RelProdS` returns a BDD of function $R \stackrel{\text{def}}{=} \big(\exists X_\exists\, F \wedge G\big)[S]$,

Let $x_F \in X$ and $x_G \in X$ be the variables of the root BDD nodes of F and G, respectively, and let x be the smallest of x_F and x_G according to ordering $<$. Let $\texttt{RPS}_{x=v}$ denote the recursive execution of `RelProdS` that calculates $R_{x=v}$ with $v \in \{0, 1\}$. Then we define the `RelProdS` algorithm is as follows:

$$\texttt{RelProdS}(F, G, X_\exists, S) = \begin{cases} 1 & F = 1 \wedge G = 1 \\ 0 & F = 0 \vee G = 0 \\ \texttt{RPS}_{x=0} \vee \texttt{RPS}_{x=1} & x \in X_\exists \\ \texttt{MK}(S(x), \texttt{RPS}_{x=1}, \texttt{RPS}_{x=0}) & \text{otherwise} \end{cases}$$

The full algorithm of `RelProdS` is given in Algorithm 2. This algorithm is identical to the algorithm of `RelProd` (see Algorithm 1 for a simplified version) except for l. 21, where the variable is substituted. To guarantee that the result is still ordered according to $<$, the ordering $<$ must be preserved under S. Here $>$ can be any total ordering, e.g. the index in the hashtable. We use a memoization table (l. 7, 22) to memorize the results. Normalization rules are added (l. 3-6), so similar operations use the same entry in the memoization table. We also insert a shortcutting optimization that omits calculating R_1 when $R_0 = 1$ (l. 14).

Table 1
Comparison of `RelProd`+`S` and `RelProdS` (numbers rounded to 10^6)

Model	#states	#trans	Units of work ($\cdot 10^6$)			BDD nodes ($\cdot 10^6$)		
			`RP + S`	`RPS`	Decr.	`RP + S`	`RPS`	Decr.
`bakery.4`	$1.5\ 10^5$	$4.1\ 10^5$	5	4	18.3%	2	1	38.1%
`bakery.8`	$2.5\ 10^8$	$9.8\ 10^8$	1,188	997	14.0%	353	219	38.0%
`collision.5`	$4.3\ 10^8$	$1.6\ 10^9$	1,187	983	18.2%	470	297	36.9%
`iprotocol.7`	$9.8\ 10^6$	$2.0\ 10^8$	759	601	20.8%	344	204	40.8%
`lifts.4`	$1.1\ 10^5$	$2.4\ 10^5$	41	38	8.1%	8	5	36.5%
`lifts.7`	$5.1\ 10^6$	$1.4\ 10^7$	533	489	8.2%	107	65	39.0%
`sched_world.2`	$1.6\ 10^6$	$1.4\ 10^7$	15	14	10.4%	5	3	32.4%
`sched_world.3`	$1.7\ 10^8$	$2.0\ 10^9$	200	178	11.0%	68	48	29.7%

We compared the computational and memory requirements of reachability using `RelProdS` to using `RelProd` and a separate variable substitution. Our implementation of `RelProd` includes the same optimizations as `RelProdS`. Both implementations use complement edges [27,5], which is a technique that represents F and $\neg F$ using the same graph and allows negation and comparison of F and $\neg F$ in constant time. For this experiment we used a subset of the BEEM database [31]. We selected models of various sizes from this database.

Table 1 shows the total number of non-trivial BDD suboperations. These are $\vee$, `RelProd` and `Substitute` suboperations that do not immediately return a result, but consult the memoization table or calculate the result based on the Shannon decomposition. We only counted the number of suboperations required to calculate the successors in every iteration of the reachability algorithm. Table 1 also shows the total number of BDD nodes in the BDD table after execution of the reachability algorithm. We disabled garbage collection to calculate this number. For `iprotocol.7` the amount of work reduces by 20%, and the number of BDD nodes decreases by 40%.

4 Parallelizing BDD operations

We parallelized `RelProdS` and $\vee$, which are the required BDD operations for reachability. This section presents two parallelization approaches that we applied. We will use the following terminology: An *algorithm* consists of a number of *operations*, which can be decomposed into small *tasks* or *suboperations*. Tasks require the results of other tasks in order to progress. This can be visualized in a *task dependency graph*. See also Figure 2.

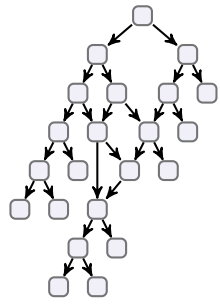

Figure 2. Task dependency graph

Tasks are executed by multiple *workers*. Typically, the number of workers is 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 faster than with 1 worker, we say the speedup for 20 workers relative to 1 worker is 5. The ideal speedup in that case would be 20. In this example, the efficiency is $5/20 = 25\%$.

Algorithm 3 Parallelizing `RelProdS` (Alg. 2) using Wool

19: **SPAWN** `RelProdS`(F_0, G_0, X, S)
20: $R_1 \leftarrow$ **CALL** `RelProdS`(F_1, G_1, X, S)
21: $R_0 \leftarrow$ **SYNC**
22: $R \leftarrow$ `MK`$(S(x), R_1, R_0)$

4.1 Parallelization using work stealing

The primary goal of parallelizing an algorithm is speedup. Ideally, work is distributed evenly among workers and a speedup is obtained equal to the number of workers. The problem of distributing work evenly is called *load balancing*. One approach is to store subtasks in queues and to let workers "steal" tasks from the queue of other workers when they run out of work. After executing a stolen task, the result must be returned to the original task owner.

Several frameworks implement task-based parallelism, e.g. the compiler-based frameworks Cilk and OpenMP and the library-based framework Wool [17]. These frameworks support creating tasks (`spawn`) and waiting for their completion (`sync`) to use the results. We selected Wool for the parallelization of symbolic reachability for several reasons. According to [32], Wool offers superior scalability in fine-grained task-based parallelism, compared to Cilk and OpenMP. There is also a blog reporting on parallelizing the BDD package BuDDy using Cilk [20] and using Wool we expect similar results. Finally, it is quite straightforward to implement parallelism using the Wool framework.

We parallelize `RelProdS` and $\vee$ by creating new tasks, whenever there are two recursive calls in Algorithm 2. To this end, we use the C macro `SPAWN` provided by Wool, followed by the matching macro `SYNC` to retrieve the results. Whenever the `SPAWN` would immediately be followed by a `SYNC`, macro `CALL` is used instead. Note that `CALL` causes the task to be immediately executed by the owner, while `SPAWN` will add a new task to the task queue. In particular, to parallelize `RelProdS`, we replace l. 19-21 from Algorithm 2 by the lines in Algorithm 3. The subtask at line 19 is put on the task queue, so that it can be stolen, and the subtask at line 20 is executed by the current worker.

Note that we could also have used `SPAWN` and `SYNC` on lines 12 and 16 in Algorithm 2. However, this would disable the shortcutting optimization, increasing the total amount of work. A performance gain is only expected for models that have insufficient work to steal otherwise, and do not benefit from the optimization. As in Algorithm 2, a memoization table is used to store results of suboperations. This table is shared globally, i.e., there is only one memoization table per operation.

4.2 Parallelization using result sharing and randomized load balancing

We also considered a simplified method for parallel BDD operations. It avoids the overhead of explicit load balancing, based on work stealing from task queues. Instead, all workers start with the same task, and execute subtasks in random order. The only synchronization between workers is that the results of suboperations are stored in a shared memoization table. This prevents workers to compute a suboperation

that was finished already by some worker.

Of course, it can be the case that multiple workers start the same suboperation, as is always the case for the initial task. However, due to the random order of handling suboperations, the workers will quickly branch off to different subtasks. So load balancing depends purely on randomization. For example, if a task has two subtasks, workers start on different subtasks with 50% probability. This increases rapidly with a larger number of subtasks.

5 Lockless data structures for BDDs

In parallel BDD operations, most of the communication between workers occurs in the hashtable containing BDD nodes and in the memoization table. It is essential that these data structures are designed for optimum scalability.

Traditionally, concurrency conflicts like data races are solved by locks, providing mutual exclusion. Since blocked processes must wait, locks have a negative impact on the speedup of parallel programs. Recent research has been dedicated to developing non-blocking data structures and algorithms. Herlihy and Shavit [21] distinguish *lock-free* algorithms, *wait-free* algorithms and *lockless* algorithms. Our algorithms fall in the last category. Here explicit locks are avoided by using atomic processor instructions like `compare_and_swap`.

The `compare_and_swap(ptr,old,new)` instruction atomically compares the value of `*ptr` to `old` and, if equal, sets `*ptr` to the value `new`. It returns `true` if this succeeded, or `false` if `*ptr` did not equal `old`. In the latter case, the value of `*ptr` remains unchanged.

Below, we discuss the lockless implementations of a lossy memoization table and a hashtable that supports garbage collection by reference counting.

5.1 Lockless lossy memoization table

The lockless lossy memoization table is a hashtable consisting of two arrays. One array contains the hash values of the keys plus one bit for a local short-lived lock on the bucket. The other contains the data, consisting of a key, i.e., a representation of the parameters of each task, and the result value.

The main requirement is that one cannot `get` results from the table that have not been `put` in the table. This is guaranteed by controlling access to specific buckets in the hashtable using the local locks in the hash array. This lock is set using the `compare_and_swap` instruction and released using a normal memory write. Since the memoization table is lossy, results may be overwritten. The result of a hash collision is that the new entry will overwrite the existing entry. Since recalculating results of a single task is not expensive in our case, occasionally overwriting results should not cause a significant performance loss.

The algorithm for `put` is given in Algorithm 4. The algorithm is designed to abort the operation immediately if some other worker uses the bucket. If there is a lock on the bucket or if `compare_and_swap` fails, then there is already some relevant

Algorithm 4 `put`: Insert an entry into the memoization table

Input: key, data *(note: key is a subset of data)*

```
1: hash ← calculate_hash(key)
2: index ← hash % tablesize
3: ⟨curhash, curlock⟩ ← hasharray[index]
4: if curlock = 1 then return
5: if curhash = hash then if key matches the key in data array then return
6: if not compare_and_swap(hasharray[index], ⟨curhash, 0⟩, ⟨hash, 1⟩) then return
7: write data to data array
8: hasharray[index] ← ⟨hash, 0⟩
9: return
```

Algorithm 5 `get`: Retrieve an entry from the memoization table

Input: key

```
1: hash ← calculate_hash(key)
2: index ← hash % tablesize
3: ⟨curhash, curlock⟩ ← hasharray[index]
4: if curhash ≠ hash or curlock = 1 then return NOTHING
5: if not compare_and_swap(hasharray[index], ⟨hash, 0⟩, ⟨hash, 1⟩) then return NOTHING
6: if key matches the key in data array then
7:     read result from data array
8:     hasharray[index] ← ⟨hash, 0⟩
9:     return result
10: else
11:    hasharray[index] ← ⟨hash, 0⟩
12:    return NOTHING
```

result in that bucket and we return immediately. Waiting until the lock is released and then replacing a relevant result by a new result is probably inefficient. Also, it is always allowed not to store the data, therefore it is not necessary to protect line 5.

The algorithm for `get` is given in Algorithm 5. This algorithm compares the hash, acquires the lock, compares the parameters and returns the result value. If any of these steps fail, `NOTHING` is returned. We do not wait until the lock is released. These algorithms obey the requirement, since the returned data is only read when there is a lock on the bucket, in which case it is not possible that another worker is modifying the data.

5.2 Lockless hashtable with reference counting

To store BDD nodes we implemented a lockless hashtable that supports garbage collection using reference counting. We extended a data structure for monotonically growing shared hash-tables [24] with the possibility to delete nodes and allow garbage collection.

The lockless hashtable in [24] is based on open addressing. It supports one operation, `find_or_put`, which notifies if some data was present, and inserts it if it was new. It works as follows. When inserting data, its hash value is stored in the hash array, at the first empty bucket according to the *probe sequence*. This is some fixed list of buckets, calculated deterministically from the hash value of the data. The data is stored in the data-array at the same index; the data array is protected by a short-lived lock-bit in the hash array. When retrieving data, the same probe

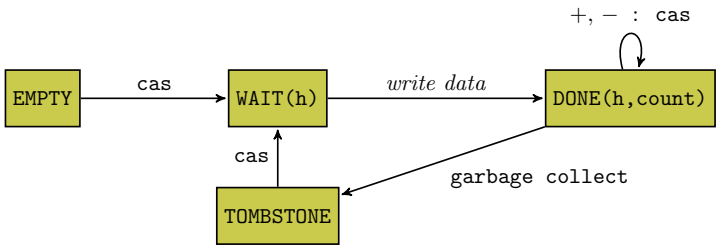

Figure 3. State transitions of hashtable buckets

sequence is followed, until either an index with the correct hash value and data is found, or an empty bucket is encountered, which indicates that the data is not present.

Note that hash values cannot simply be deleted, since this would break the probe sequence, potentially leading to inserting identical data twice and reporting that it was new. We solve this by replacing data by a special value, instead of deleting it. For garbage collection, we also add a reference count to the hash array. So hash buckets assume one of the following values:

- `EMPTY` : empty bucket, and end of a probe sequence
- `TOMBSTONE` : empty bucket, but the probe sequence continues
- $\langle$`WAIT`, hash$\rangle$: some data with this hash is being written at this index
- $\langle$`DONE`, hash, count$\rangle$: complete data, with the given hash and reference count

We encode these values in 32 bits: 15 bits for the hash, 1 bit for the lock, and 16 bits for the reference count. The reference count is prevented from integer overflow by reserving a special value `SATURATED`. When the reference count is saturated, it will no longer be increased or decreased.

Figure 3 indicates the transitions that a bucket can perform. Transitions to `WAIT` should obtain an exclusive lock, hence they are implemented with `compare_and_swap`. So are modifications to the reference count, since they must happen atomically. The transition from `DONE` to `TOMBSTONE` is only allowed during a separate garbage collection phase (and only if count = 0).

Our extended version of `find_or_put` is called `lookup_or_insert`. The algorithm (Alg. 7) consists of two loops over the probe sequence. The first loop checks whether the data is already in the table. The second loop inserts the data in the first available bucket, either marked `EMPTY` or `TOMBSTONE`. Since we assume that garbage collection occurs in a separate phase, no new `TOMBSTONE` buckets can appear during the execution of `lookup_or_insert`.

Algorithm 6 `increase`: Increase the reference count of a given bucket

Input: bucket

1: **repeat**
2: $\langle$`DONE`, hash, count$\rangle \leftarrow$ bucket
3: **if** count = `SATURATED` **then return**
4: **until** `compare_and_swap`(bucket, $\langle$`DONE`, hash, count$\rangle$, $\langle$`DONE`, hash, count+1$\rangle$)

Algorithm 7 `lookup_or_insert`: Ensure that data is in the table

Input: data

```
 1: hash ← calculate_hash(data)
 2: for i ∈ probe_sequence(data) do
 3:    if bucket[i] = EMPTY then break
 4:    if bucket[i] = ⟨..., hash, ...⟩ then
 5:       while bucket[i] = ⟨WAIT, hash⟩ do nothing
 6:       if data matches data in data array then
 7:          increase(bucket[i])
 8:          return i
 9: for i ∈ probe_sequence(data) do
10:    value ← bucket[i]
11:    if value = EMPTY or value = TOMBSTONE then
12:       if compare_and_swap(bucket[i], value, ⟨WAIT, hash⟩) then
13:          write data to data array at i
14:          bucket[i] ← ⟨DONE, hash, 1⟩
15:          return i
16:    if bucket[i] = hash then
17:       while bucket[i] = ⟨WAIT, hash⟩ do nothing
18:       if data matches data in data array then
19:          increase(bucket[i])
20:          return i
21: return TABLE FULL
```

Algorithms `increase` (Alg. 6) and `decrease` modify the reference count. Their precondition is that the bucket is of the form $\langle$`DONE`, hash, count$\rangle$. They can be called externally (for instance by the BDD package), or internally by `lookup_or_insert` and garbage collection.

6 Results

We experimented with a representative selection of models from the BEEM database [31] using a symbolic BFS reachability algorithm of `dve2-reach` from the `LTSmin` toolset [4]. Experiments ran on a 48-core machine, consisting of 4 AMD Opteron^{TM} 6168 processors with 12 cores each. This machine has a NUMA architecture with 8 memory domains and 6 cores per domain. We first parallelized the BDD operations using work stealing with Wool (see Section 4.1) by implementing an experimental parallel BDD package Sylvan. [2]

We made Wool NUMA-aware by binding each worker to a memory domain and by allocating the task queue of each worker locally, i.e., on the selected domain. With less than 48 workers, we calculated a minimum subset of memory domains at minimal distance, as reported by the NUMA library and assigned workers to each memory domain in a round-robin fashion. For example, for 10 workers we would assign 5 workers to 2 domains each, selected at minimal distance. We used preallocated BDD hashtables and memoization tables, which were allocated interleaved over all selected memory domains. We also modified `LTSmin` to run symbolic reachability twice: in the first run the transition relation groups are learned on-the-fly and stored as BDDs. The second run reuses this precalculated transition relation to compute

[2] Sylvan is part of `LTSmin 2.0`, http://fmt.cs.utwente.nl/tools/ltsmin/

Table 2
Runtimes in seconds and speedups of reachability with Sylvan and BuDDy

Model	Sylvan								BuDDy	Sp.
	1	2	4	8	16	32	48	Sp.		
bakery.4	11.4	6.8	5.4	4.5	4.4	4.4	4.7	2.4	1.9	0.4
bakery.8	1370.0	681.5	348.1	184.7	102.4	62.0	49.8	27.5	517.7	10.4
collision.5	1828.4	920.8	505.6	256.5	138.6	76.6	57.2	32.0	623.3	10.9
iprotocol.7	1012.2	507.9	261.1	137.2	76.0	46.3	37.4	27.1	351.9	9.4
lifts.4	34.1	17.8	10.0	6.3	5.0	5.0	5.8	5.9	12.4	2.1
lifts.7	473.1	239.0	123.4	67.3	40.2	28.9	27.6	17.2	194.6	7.1
sched_world.2	17.8	9.5	5.6	3.6	2.7	2.4	2.4	7.4	6.5	2.7
sched_world.3	260.1	131.4	67.5	35.6	19.7	11.8	9.5	27.4	114.3	12.0

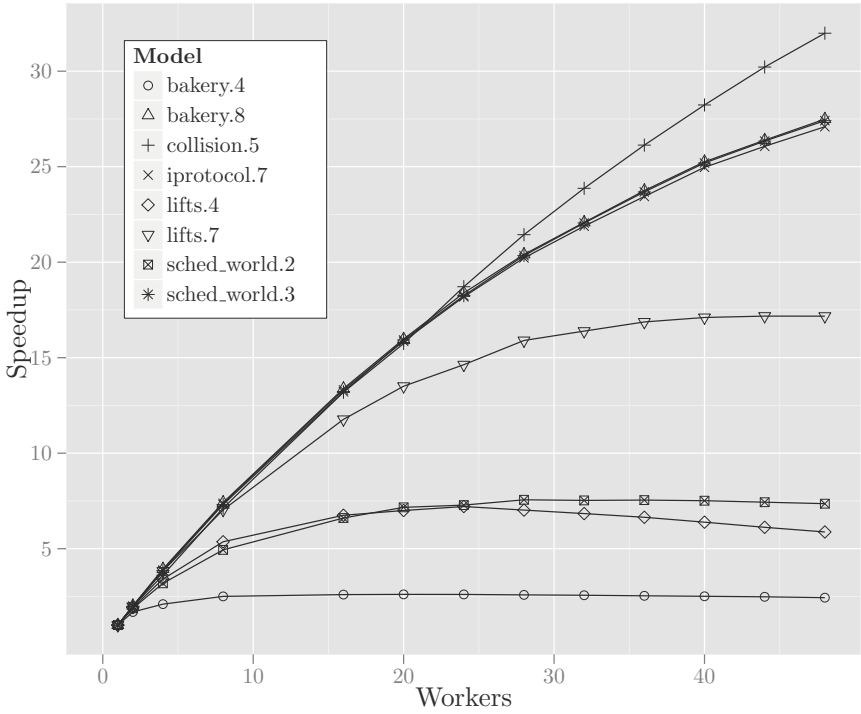

Figure 4. Speedups of reachability with Sylvan on a 48-core machine

the set of reachable states symbolically. We only measured the time spent in the second run, since we are interested in the speedup of the BDD operations only.

Table 2 and in Figure 4 show the results for several representative models. From these results we see a clear relation between the size of the model and the obtained speedup. Comparing the results to Table 1, we see that smaller models (less than 100,000,000 units of work, and less than 10,000,000 total created BDD nodes) have very limited speedups, while the largest models exhibit the best speedups. The smaller `sched_world.3` model is an exception that still shows a decent speedup. Note that the numbers average the speedups of all BDD operations during a full reachability analysis, hence the individual larger BDD operations likely scale better since the BDDs in initial BFS levels are small.

Although a relative speedup of 32 on 48 cores is already very nice, we investigated further to find reasons why this number is not higher. When running benchmarks of Wool parallelizing the Fibonacci algorithm without memoization, i.e., each task only consists of adding the results of two subtasks, we found that Wool itself scales to a speedup of about 34 on 48 cores. This may be increased in future work by redesigning the work-stealing algorithm to be lockless instead of using mutual exclusion on the task queues, as in [38]. We also experimented with using the memoization table only every 1 in N variable levels. With low values of N, this resulted in some increased performance (up to 10%) and significant reduction of the memoization table usage, but little improvement in relative speedup [15].

We compared the runtimes of the reachability algorithm of the `LTSmin` toolset using our parallel implementation Sylvan to the popular sequential BDD package BuDDy [25]. We witness a speedup of up to 12 times compared to BuDDy (Table 2). There are several differences between the implementation in BuDDy and the implementation in Sylvan that make comparing the performances difficult. BuDDy does not implement `RelProdS` or complement edges. Sylvan uses reference counting for garbage collection, while BuDDy uses mark-and-sweep. However, the preallocated tables were large enough that garbage collection did not occur with Sylvan nor with BuDDy. Sylvan still updated reference counts, so there is an advantage to BuDDy, since mark-and-sweep requires less bookkeeping. BuDDy also uses several other optimizations, such as increased memory locality by storing related BDD nodes near each other in the hashtable, while Sylvan stores BDD nodes at the same position as the hash in the hashtable. Finally, BuDDy is not thread safe and only uses normal memory transfers, while we replace some normal memory transfers by more expensive `compare_and_swap` operations to ensure thread safety.

We also experimented using randomized load balancing (see Section 4.2) and report decent performance and scalability elsewhere [15]. The conclusion there is that this alternative approach is viable, but the approach using Wool currently gives slightly higher performance and a larger speedup.

7 Related work

In the literature, there is some earlier work prior to 2000 that parallelizes BDD manipulation on massively parallel SIMD machines and on distributed architectures. There is no recent work on modern multi-core shared-memory architectures that parallelizes the actual BDD operations.

In the early 90's, several researchers tried to speed up BDD manipulation by parallel processing. The first paper [23] views BDDs as automata, and combines them by computing a product automaton followed by minimization. Parallelism arises by handling independent subformulae in parallel: the expansion and reduction algorithms themselves are not parallelized. Most other work in this era implemented BFS algorithms for vector machines [28] or massively parallel SIMD machines [8,18] with up to 64K processors. Experiments were run on supercomputers, like the Connection Machine. Other solutions were based on Distributed Shared Memory

abstractions, to implement the standard depth-first algorithm [30,9], or a hybrid depth/breadth-first approach [39].

Attention shifted towards Networks of Workstations, based on message passing libraries. The motivation was to combine the collective memory of computers connected via a fast network. Both depth-first [2,37,3] and breadth-first [34] traversal has been proposed. In the latter, BDDs are distributed according to variable levels. A worker can only proceed when its level has a turn, so these algorithms are inherently sequential. The experiments showed that very large BDDs can be manipulated, but no speedups were observed. Finally, BDDNOW [26] was the first system for distributed BDD manipulation claiming some speedup before physical memory is exhausted.

After 2000, research attention shifted from parallel implementations of BDD operations towards the use of BDDs for symbolic reachability in distributed [19,10] or shared memory [16,12]. Based on BDD partitioning strategies nice speedups could be obtained [33,19]. Also saturation, an optimal iteration strategy, was parallelized using Cilk [10,16]. A compositional algorithm that computes an overapproximation of the reachable state set was parallelized by conjunctively splitting invariants into local components, using separate BDD tables for each worker [14].

Published research on multi-core BDD algorithms is notably absent. In a thesis on JINC [29], Chapter 6 describes a multi-threaded extension. JINC's parallelism relies on concurrent tables and delayed evaluation. However, it doesn't parallelize the basic BDD operations. A Cilk-based parallel implementation of the Apply function is reported in a blog [20]. It reports some speedup on a single example. Detailed information is not online.

8 Conclusion

In this paper, we presented a new algorithm `RelProdS` that calculates the relational product and the variable substitution in one step. We showed that this algorithm reduces the amount of work of symbolic reachability by up to 20% and decreases the memory requirements by up to 40%.

We designed and implemented two data structures to support a parallel implementation of BDD operations: a lockless lossy memoization table and a lockless hashtable supporting garbage collection with reference counting. We implemented the parallel operations `RelProdS` and $\vee$ in our parallel BDD package Sylvan using these lockless data structures and the work-stealing framework Wool.

Performance measurements with this parallel implementation demonstrated relative speedups of up to 32 using 48 cores. Compared to the popular BDD package BuDDy we get a speedup of up to 12 using 48 cores. We demonstrated that parallelizing BDD operations on a low level is a viable method to get good speedups for symbolic reachability on multi-core multi-processors with a non-uniform shared-memory architecture.

References

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

[2] Arunachalam, P., C. M. Chase and D. Moundanos, *Distributed binary decision diagrams for verification of large circuit*, in: *ICCD* (1996), pp. 365–370.

[3] Bianchi, F., F. Corno, M. Rebaudengo, M. S. Reorda and R. Ansaloni, *Boolean function manipulation on a parallel system using BDDs*, in: *HPCN Europe*, LNCS **1225**, 1997, pp. 916–928.

[4] Blom, S., J. van de Pol and M. Weber, *LTSmin: distributed and symbolic reachability*, in: *Proc. of the 22nd int. conf. on Computer Aided Verification*, CAV'10 (2010), pp. 354–359.

[5] Brace, K. S., R. L. Rudell and R. E. Bryant, *Efficient implementation of a BDD package*, in: *DAC*, 1990, pp. 40–45.

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

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

[8] Cabodi, G., S. Gai and M. Sonza Reorda, *Boolean function manipulation on massively parallel computers*, in: *Proc. of 4th Symp. on Frontiers of Massively Parallel Computation*, 1992, pp. 508–509.

[9] Chen, J.-S. and P. Banerjee, *Parallel construction algorithms for BDDs*, in: *ISCAS (1)* (1999), pp. 318–322.

[10] Chung, M.-Y. and G. Ciardo, *Saturation NOW*, in: *QEST* (2004), pp. 272–281.

[11] Ciardo, G. and A. S. Miner, *Smart: The stochastic model checking analyzer for reliability and timing*, in: *QEST*, 2004, pp. 338–339.

[12] Ciardo, G., Y. Zhao and X. Jin, *Parallel symbolic state-space exploration is difficult, but what is the alternative?*, in: L. Brim and J. van de Pol, editors, *PDMC*, EPTCS **14**, 2009, pp. 1–17.

[13] Ciardo, G., Y. Zhao and X. Jin, *Ten years of saturation: A petri net perspective*, T. Petri Nets and Other Models of Concurrency **5** (2012), pp. 51–95.

[14] Cohen, A., K. Namjoshi, Y. Saar, L. Zuck and K. Kisyova, *Parallelizing a symbolic compositional model-checking algorithm*, in: *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science **6504**, Springer Berlin / Heidelberg, 2011 pp. 46–59.

[15] Dijk, T. v., "The Parallelization of Binary Decision Diagram operations for model checking," Master's thesis, University of Twente, Department of Computer Science (2012), available at http://fmt.cs.utwente.nl/tools/ltsmin/papers/thesis-sylvan-tvdijk.pdf.

[16] Ezekiel, J., G. Lüttgen and G. Ciardo, *Parallelising symbolic state-space generators*, in: *CAV*, LNCS **4590**, 2007, pp. 268–280.

[17] Faxén, K.-F., *Efficient work stealing for fine grained parallelism*, in: *2010 39th International Conference on Parallel Processing (ICPP)* (2010), pp. 313–322.

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

[19] Grumberg, O., T. Heyman and A. Schuster, *A work-efficient distributed algorithm for reachability analysis*, Formal Methods in System Design **29** (2006), pp. 157–175.

[20] He, Y., *Multicore-enabling a binary decision diagram algorithm* (October 27, 2009), intel blog, originally posted at www.cilk.com on May 29, 2009. Available at http://software.intel.com/en-us/articles/multicore-enabling-a-binary-decision-diagram-algorithm/.

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

[22] Janssen, G., *Design of a pointerless BDD package*, in: *Note at Int'l Workshop Logic and Synthesis (IWLS-2001)*, 2001.

[23] Kimura, S. and E. Clarke, *A parallel algorithm for constructing binary decision diagrams*, in: *Proc. of IC on Computer Design: VLSI in Computers and Processors ICCD*, 1990, pp. 220–223.

[24] Laarman, A., J. van de Pol and M. Weber, *Boosting multi-core reachability performance with shared hash tables*, in: *Formal Methods in Computer-Aided Design* (2010), pp. 247–255.

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

[26] Milvang-Jensen, K. and A. J. Hu, *BDDNOW: A parallel BDD package*, in: *FMCAD*, LNCS **1522**, 1998, pp. 501–507.

[27] Minato, S.-i., 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 (1990), pp. 52–57.

[28] Ochi, H., N. Ishiura and S. Yajima, *Breadth-first manipulation of SBDD of Boolean functions for vector processing*, in: *DAC*, 1991, pp. 413–416.

[29] Ossowski, J., "JINC – A Multi-Threaded Library for Higher-Order Weighted Decision Diagram Manipulation," Ph.D. thesis, Rheinischen Friedrich-Wilhelms-Universität Bonn (2010).

[30] Parasuram, Y., E. P. Stabler and S.-K. Chin, *Parallel implementation of BDD algorithms using a distributed shared memory*, in: *HICSS (1)*, 1994, pp. 16–25.

[31] Pelánek, R., *BEEM: benchmarks for explicit model checkers*, in: *SPIN* (2007), pp. 263–267.

[32] Podobas, A., 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).

[33] Sahoo, D., 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 (2005), pp. 467–470.

[34] Sanghavi, J. V., R. K. Ranjan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, *High performance BDD package by exploiting memory hiercharchy*, in: *DAC*, 1996, pp. 635–640.

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

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

[37] Stornetta, T. and F. Brewer, *Implementation of an efficient parallel BDD package*, in: *DAC*, 1996, pp. 641–644.

[38] Sundell, H. and P. Tsigas, *Brushing the locks out of the fur: A lock-free work stealing library based on wool*, in: *2nd Swedish Workshop on Multi-Core Computing MCC09* (2009), pp. 126–130.

[39] Yang, B. and D. R. O'Hallaron, *Parallel breadth-first BDD construction*, in: *PPOPP*, 1997, pp. 145–156.