Boosting Multi-Core Reachability Performance with Shared Hash Tables

Alfons Laarman, Jaco van de Pol, Michael Weber
{a.w.laarman,vdpol,michaelw}@cs.utwente.nl
Formal Methods and Tools, University of Twente, The Netherlands

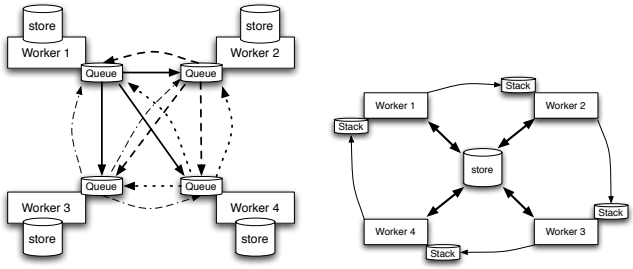
***Abstract*—This paper focuses on data structures for multi-core reachability, which is a key component in model checking algorithms and other verification methods. A cornerstone of an efficient solution is the storage of visited states. In related work, static partitioning of the state space was combined with thread-local storage. This solution leaves room for improvements. This paper presents a solution with a shared state storage. It is based on a lockless hash table implementation and scales better. The solution is specifically designed for the cache architecture of modern CPUs. Because model checking algorithms impose loose requirements on the hash table operations, their design can be streamlined substantially compared to related work on lockless hash tables. The resulting speedups are analyzed and compared with related tools. Our implementation outperforms two state-of-the-art multi-core model checkers, SPIN (presented at FMCAD 2006) and DiVinE, by a large margin, while placing fewer constraints on the load balancing and search algorithms.**

I. Introduction

Many verification problems are highly computational intensive tasks that can benefit from extra speedups. Considering the recent hardware trends, these speedups can only be delivered by exploiting the parallelism of the new multi-core CPUs.

Reachability, or full exploration of the state space, is a subtask of many verification problems [6], [8]. In model checking, reachability has in the past been parallelized using distributed systems [6]. With shared-memory systems, these algorithms can benefit from the low communication costs as has been demonstrated already [1]. In this paper, we show how the performance of state-of-the-art multi-core model checkers, like SPIN [13] and DiVinE [1], can be greatly improved using a carefully designed concurrent hash table as shared state storage.

Motivation: Holzmann and Bošnacki used a shared hash table with fine-grained locking in combination with the stack-slicing algorithm in their multi-core extension of the SPIN model checker [12], [13]. This shared storage enabled the parallelization of many of the model checking algorithms in SPIN: safety properties, partial order reduction and reachability. Barnat et al. implemented the same method in the DiVinE model checker [1]. They chose to implement the classic method of static state space partitioning, as used in distributed model checking [3]. They found the static partitioning method to scale better on the basis of experiments. The authors also mention that they were not able to develop a potentially better solution for shared state storage, namely the use of a lockless hash table. Thus it remains unknown whether reachability, based on shared state storage, can scale.



(a) Static partitioning (b) Stack slicing

Fig. 1. Different architectures for model checkers

TABLE I
Differences between architectures

Arch.	Sync. points	Pros / Cons
Fig. 1(a)	Queue	local (*cache efficient*) storage / *static* load balancing, *high* comm. costs, *limited* to BFS
Fig. 1(b)	Shared store, stack	low comm. costs / *specific* load balancing, limited to (pseudo) DFS
Shared store	Shared store, (queue)	low comm. costs, *flexible* load balancing, *flexible* exploration algorithm / scalability?

Using a shared state storage has further benefits. Fig. 1 shows the different architectures discussed thus far. Their differences are summarized in Table I and have been extensively discussed by Barnat et al. [3]. They also investigate a more general architecture with a shared storage and arbitrary load-balancing strategy (not necessarily stack-slicing). Such a solution is both simpler and more flexible, in the sense that it allows for more freedom in the choice of the exploration algorithm, including (pseudo) DFS, which enables fast searches for deadlocks and error states [20]. Holzmann already demonstrates this [12], but could not show desirable scalability of SPIN (as we will demonstrate). The *stack-slicing algorithm* [12], is a specific case of load balancing that requires DFS. In fact, any well-investigated load-balancing solution [21] can be used and tuned to the specific environment, for example, to support heterogeneous systems or BFS exploration. Inggs and Barringer use a *lossy* shared hash table [14], resulting in reasonable speedups at the cost of precision (states can potentially be revisited), but give little details on the implementation.

Contribution: We present a data structure for efficient concurrent storage of states. This enables scaling parallel implementations of reachability for many desirable exploration algorithms. The precise needs which parallel model checking algorithms impose on shared state storage are evaluated and a

fitting solution is proposed given the identified requirements. Experiments show that our implementation of the shared storage scales significantly better than an implementation using static partitioning, but also beats state-of-the-art model checkers. By analysis, we show that our design will scale beyond current state-of-the-art multi-core processors. The experiments also contribute to a better understanding of the performance of the latest versions of SPIN and DiVinE.

Overview: Section II presents background on reachability, load balancing, hashing, parallel algorithms and multi-core systems. Section III presents the lockless hash table, which we designed for shared state storage. But only after we evaluated the requirements that fast parallel algorithms impose on such a shared storage. In Section IV, the performance is evaluated against that of DiVinE 2 [2] and SPIN. A fair comparison can be made between the three model checkers on the basis of a set of models from the BEEM database which report the same number of states for both SPIN and DiVinE. We end this paper by putting the results we obtained into context, and an outlook on future work (Section V).

II. Preliminaries

Reachability in Model Checking: In model checking, a computational model of the system under verification (hardware or software) is constructed, which is then be used to compute all possible states of the system. The exploration of all states can be done symbolically, e.g., using *binary decision diagrams* (BDDs) to represent sets of states, or by enumerating and explicitly storing all states. While symbolic methods are attractive for a certain set of models, they are not a silver bullet: due to *BDD explosion*, sometimes plain enumerative methods are faster. In this paper, we focus on enumerative model checking.

Enumerative reachability analysis can be used to check for deadlocks and invariants and also to store the whole state space and verify multiple properties of the system at once. Reachability is an exhaustive search through the state space. The algorithm calls for each state the *next-state* function to obtain its successors until no new states are found (Alg. 1). We use an *open set* T, which can be implemented as a stack or queue, depending on the preferred exploration order: depth or breadth-first. The initial state s_0 is obtained from the model and added to T. In the loop starting on Line 1, a state is taken from T, its successors are computed using the model (Line 3) and each new successor state is put into T again for later exploration. To determine which state is new, a *closed set* V is used. V can be implemented with a hash table.

Possible ways to parallelize Alg. 1 have been discussed in the introduction. A common denominator of all these approaches is that the strict BFS or DFS order of the sequential algorithm is sacrificed in favor of *thread-local* open sets (fewer contention points). When using a shared state storage (in a general setup or with stack-slicing), a thread-safe set V is required, which will be discussed in the following section.

Load Balancing: A naive parallelization of reachability can be realized as follows: perform a depth-limited sequential BFS exploration and hand off the found states to several threads

```
Data: Sequence T = {s0}, Set V = ∅
1 while state ← T.get() do
2     count ← 0;
3     for succ in next-state(state) do
4         count ← count + 1;
5         if V.find-or-put(succ) then
6             T.put(succ);
7     if 0 == count ... report deadlock ...
```

Algorithm 1: Reachability analysis

that start executing Alg. 1 ($T = \{$*part of BFS exploration*$\}$ and V is shared). This is called *static load balancing*. For many models this will work due to common diamond-shaped state spaces. However, models with synchronization points or strict phase structure sometimes exhibit helix-shaped state spaces. Hence, threads run out of work when they reach a converge point in their exploration. A well-known problem that behaves like this is the *Towers of Hanoi* puzzle; when the smallest disk is on top of the tower only one move is possible.

Sanders [21] describes *dynamic load balancing* in terms of a problem P, a (reduction) operation $work$ and a $split$ operation. P_{root} is the initial problem. Sequential execution takes $T_{\text{seq}} = T(P_{\text{root}})$ time units. A problem is (partly) solved when calling $work(P, t)$, which takes $min(t, T(P))$ units of time. For parallel reachability, $work(P, t)$ is Alg. 1, where t has to be added as an extra input that limits the number of iterations of the *while* loop on Line 1 (and $P_{\text{root}} = T = \{s_0\}$). When threads become idle, they can poll others for work. The receiver will then split its own problem instance ($split(P) = \{P_1, P_2\}$, $T(P) = T(P_1) + T(P_2)$) and send one of the results to the polling thread.

Parallel architectures: We consider multi-core x86 server and desktop systems. These systems can process a large number of instructions per second, but have a relatively low memory bandwidth. Multiple levels of cache are used to continuously feed the cores with data. Some of these caches are shared among multiple cores (often L2) and others are local (L1), depending on the architecture of the CPU and number of CPUs. The cache coherence protocol ensures that each core in each CPU has a global view of the memory. It transfers blocks of memory to local caches and synchronizes them if a local block is modified by other cores. Therefore, if independent writes are performed on subsequent memory locations (on the same cache line), a problem known as *cache line sharing* (*false sharing*) occurs, causing gratuitous synchronization and overhead.

The cache coherence protocol cannot be preempted. To efficiently program these machines, few options are left. One way is to completely partition the input [3], thus ensuring per-core memory locality at the cost of increased inter-die communication. An improvement of this approach is to pipeline the communication using ring buffers, this allows prefetching (explicit or hardwired). This scheme was explored, e.g., by Monagan and Pearce [17]. The last alternative is to minimize the *memory working set* of the algorithm [19]. We define the memory working set as the number of different memory

locations that the algorithm updates in the time window that these usually stay in local cache. A small working set minimizes coherence overhead.

Locks: It is common to ensure mutual exclusion for a critical section of code by locks. However, for resources with high contention, locks become infeasible. *Lock proliferation* improves on this by creating more locks on smaller resources. *Region locking* is an example of this, where a data structure is split into separately locked regions based on memory locations. However, this method is still infeasible for computational tasks with very high throughput. This is caused by the fact that the lock itself introduces another synchronization point; and synchronization between processor cores takes time.

Lockless Algorithms: For high-throughput systems, *lock-free algorithms* (without mutual exclusion) are preferred. Lock-free algorithms guarantee system-wide progress, i.e., always *some* thread can continue. If an algorithm does not strictly provide progress guarantees (only statistically), but otherwise avoids explicit locks by the same techniques as used in lock-free solutions, it is called *lockless*. Lockless algorithms often have considerably simpler implementations, at no performance penalty. Lastly, *Wait-free algorithms* guarantee per-thread progress, i.e., *all* threads can continue.

Many modern CPUs implement a *Compare & Swap* operation (CAS) which ensures atomic memory modification while at the same time preserving data consistency if used in the correct manner. This can be done by reading the value from memory, performing the desired computation on it and writing the result back using CAS (Alg. 2). If the latter returns true, the modification succeeded, if not, the computation needs to be redone with the new value, or some other form of collision resolution should be applied.

Pre: $word \neq$ **null**
Post:$(*word_{pre} = testval \Rightarrow *word_{post} = newval) \wedge$
$(*word_{pre} \neq testval \Rightarrow *word_{post} = *word_{pre})$
atomic bool CAS(int *word, int testval, int newval)

Algorithm 2: "Compare&Swap" specification

Lockless algorithms can achieve a high level of concurrency. However, an instruction like CAS easily costs 100–1000 instruction cycles depending on the CPU architecture. Thus, abundant use defies its purpose.

Quantifying Parallelism: Parallelism is usually quantified by normalizing the performance gain with regard to a sequential run (*speedup*): $S = T_{seq}/T_{par}$. Linear speedups grow proportional to the number of cores and indicate that an algorithm scales well. Ideal speedup is achieved when $S \geq N$. For a fair comparison of scalability, it is important to use the fastest tool for T_{seq}, or speedups will not be comparable, since better optimized code is harder to scale (e.g., [13]).

Hashing: A well-studied method for storing and retrieving data with amortized time complexity $O(1)$ is *hashing* [16]. A hash function h is applied to the data, yielding an index in an array of *buckets* that contain the data or a pointer to the data. Since the domain of data values is usually unknown and much larger than the image of h, hash collisions occur when $h(D_1) = h(D_2)$, with $D_1 \neq D_2$. Structurally, collisions can be resolved either by inserting lists in the buckets (*chaining*) or by probing subsequent buckets (*open addressing*). Algorithmically, there is a wealth of options to maintain the "chains" and calculate subsequent buckets [9]. The right choice depends entirely on the requirements dictated by the algorithms that use the hash table.

III. A Lockless Hash Table

In principle, Alg. 1 seems easy to parallelize, in practice it is difficult to do this efficiently because of its memory intensive behavior, which becomes more obvious when looking at the implementation of set V. In this section, we present an overview of the options in hash table design. There is no silver bullet design and individual design options should be chosen carefully considering the requirements stipulated by the use of the hash table. Therefore, we evaluate the demands that the parallel model checking algorithms place on the state storage solution. We also mention additional requirements stemming from the targeted hardware and software systems. Finally, we present a specific hash table design.

A. Requirements on the State Storage

Our goal is to realize an efficient shared state storage for parallel model checking algorithms. Traditional hash tables associate a piece of data to a unique key in the table. In model checking, we only need to store and retrieve *state vectors*, therefore the key is the state vector itself. Henceforth, we will simply refer to it as *data*. Our specific model checker implementation introduces additional requirements, discussed later. First, we list the definite requirements on the state storage:

- The storage needs only one operation: `find-or-put`. This operation inserts the state vector if it is not found or yields a positive answer without side effects. We require `find-or-put` to be concurrently executable to allow sharing the storage among the different threads. Other operations are not necessary for reachability algorithms, since the state space is growing monotonically. By exploiting this feature we can simplify the algorithms, thus lowering the strain on memory, and avoiding cache line sharing. Our choice is in sharp contrast to standard literature on concurrent hash tables, which often favors a complete solution, which is optimized for more general access patterns [7], [19].
- The storage should not require continual memory allocation, for the obvious reasons that this behavior would increase the *memory working set*.
- The use of pointers on a per-state basis should be avoided. Pointers take a considerable amount of memory when large state spaces are explored (more than 10^8 states are easily reachable with today's model checkers), especially on 64-bit machines. In addition, pointers increase the memory working set.
- The time efficiency of `find-or-put` should scale with the number of processes executing it in parallel. Ideally,

the individual operations should — on average — not be slowed down by other operations executing at the same time, thus ensuring close-to linear speedup. Many hash table algorithms have a large memory working set due to their probing behavior or reordering behavior upon insertions. They suffer performance degradation in high throughput situations as is the case for us.

Specifically, we do not require the state storage to be resizable. The available memory on a system can safely be claimed for the table, because the largest part will be used for it eventually anyway. In sequential operation and especially in presence of a *delete* operation (shrinking tables), one would consider resizing for the obvious reason that it improves locality and thus cache hits. In a concurrent setting, however, these cache hits have the opposite effect of causing the earlier described cache line sharing among CPUs. We experimented with lockless and concurrent resizing mechanisms and observed large decreases in performance.

Furthermore, the design of the LTSmin tool [5], which we extended with a multi-core reachability, also introduces some specific requirements:

- The storage data consists only of integer arrays or vectors of known and fixed length. This is the encoding format for state vectors employed by our language front-ends.
- The storage is targeted at common x86 architectures, using only the available (atomic) instructions.

While the compatibility with the x86 architecture allows for concrete analysis, the applicability of our design is not limited to it. Lessons learned here are transferable to other architectures with similar memory hierarchy and atomic operations.

B. Hash Table Design

We determined that a low memory working set is one of the key factors to achieve maximum scalability. Also, we opt for simplicity whenever the requirements allow for it. From experience we know that complexity of a solution arises automatically when introducing concurrency. These considerations led us to the following design choices:

- **Open addressing**, since the alternative *chaining* hash table design would incur in-operation memory allocation or pre-allocation at different addresses, both leading to a larger memory working set.
- **Walking-the-line** is the name we gave to *linear probing* on a cache line, followed by *double hashing* (also employed elsewhere [7], [11]). Linear probing allows a core to benefit fully from a loaded cache line, while double hashing realizes better distribution.
- **Separated data** (vectors) in an indexed *data array* (of size $buckets \times |vector|$) ensures that the *bucket array* stays short[1] and subsequent probes can be cached.
- **Hash memoization** speeds up probing, by storing the hash (or part of it) in a bucket. This avoids expensive lookups in the data array as much as possible [7].
- **Lockless** operation on the bucket array using a dedicated value to indicate unused buckets. One bit of the hash can be used to indicate whether the vector was already written to the data array or whether writing is still in progress [7].
- **Compare-and-swap** is used as an atomic primitive on the buckets, which are precisely in either of the following distinguishable states: *empty*, *being written* and *complete*.

[1]E.g., 1 GB for a 32-bit memoized hash and 2^{28} buckets

C. Hash Table Operations

Alg. 3 shows the `find-or-put` operation. Buckets are represented by the $Bucket$ array, the separate data by the $Data$ array and hash functions used for double hashing by $\texttt{hash}_i$. Probing continues (Line 4) until either a free bucket is found for insertion (Line 8–10), or the data is found to be in the hash table (Line 14). Too many probes indicate a table size mismatch, which simply causes the application to abort. The *for* loop on Line 5 handles the walking-the-line probing behavior (Alg. 4). The other code inside this loop handles the synchronization among threads. We explain this part of the algorithm now in detail.

Buckets store memoized hashes and the *write status* bit of the data in the $Data$ array. The possible values of the buckets are thus: EMPTY, $\langle h, \text{WRITE}\rangle$ and $\langle h, \text{DONE}\rangle$, where h is the memoized hash. If an empty bucket is encountered on a probe sequence, the algorithm tries to claim it by atomically writing $\langle h, \text{WRITE}\rangle$ to it (Line 7). After finishing the writing of the data, $\langle h, \text{DONE}\rangle$ is written to the bucket (Line 9). Non-empty buckets prompt the algorithm to compare the memoized hashes (Line 11). Only if they match, the value in the data array is compared with the vector (Line 13).

```
   Data: size, Bucket[size], Data[size]
   input  : vector
   output : seen
1  count ← 1;
2  h ← hash_count(vector);
3  index ← h  mod size;
4  while count < THRESHOLD do
5      for i in walkTheLineFrom(index) do
6          if EMPTY = Bucket[i] then
7              if CAS(Bucket[i], EMPTY, ⟨h, WRITE⟩) then
8                  Data[i] ← vector;
9                  Bucket[i] ← ⟨h, DONE⟩;
10                 return false;
11         if ⟨h, −⟩ = Bucket[i] then
12             while ⟨−, WRITE⟩ = Bucket[i] do ..wait.. done
13             if Data[i] = vector then
14                 return true;
15     count ← count + 1;
16     index ← hash_count(vector)  mod size;
```

Algorithm 3: The `find-or-put` algorithm

Several aspects of the algorithm guarantee correct lockless operation:

Data: *cache_line_size*, *Walk*[*cache_line_size*]
input : *index*
output: *Walk*[*cache_line_size*]
1 $start \leftarrow \lfloor index/cache\_line\_size \rfloor \times cache\_line\_size$;
2 **for** $i \leftarrow 0$ **to** $cache\_line\_size - 1$ **do**
3 $\quad Walk[i] \leftarrow start + (index + i) \mod cache\_line\_size$;

Algorithm 4: Walking the (cache) line

- Whenever a write started for a hash value, the state of the bucket can never become empty again, nor can it be used for any other hash value. This ensures that the probe sequence remains deterministic and cannot be interrupted.
- The CAS operation on Line 7 ensures that only one thread can claim an empty bucket, marking it as non-empty with the hash value to memoize and with state WRITE.
- The *while* loop on Line 12 waits until the write to the data array has been completed.

Critical synchronization between threads occurs when multiple threads try to write to an empty bucket. The CAS operation ensures that only one will succeed. The others carry on in their probing sequence, either finding another empty bucket or finding the state vector in another bucket. This design can be seen as a lock on the lowest possible level of granularity (individual buckets), but without a true locking structure and associated additional costs. The algorithm implements the "lock" as *while* loop, which resembles a spinlock (Line 12). Although it could be argued that this algorithm is therefore not lock-free, it is possible to ensure local progress in the case that the "blocking" thread dies or hangs (making the algorithm wait-free). Wait-freeness is commonly achieved by making each thread fulfil local invariants, whenever they are not (yet) met by other threads [10]. Our measurements show, however, that under normal operation the loop on Line 12 is rarely hit due to the preceding hash memoization check (Line 11). Thus, we took the pragmatic choice of keeping the implementation as simple as possible.

Our implementation of the described algorithm requires exact guarantees from the underlying memory model. Reordering of operations by compilers and processors needs to be avoided across the synchronization points, otherwise the implementation becomes incorrect. It is, for example, a common optimization to execute the body of an *if* statement before the actual branching instruction. Such a speculative execution would keep the processor pipeline busy, but would be a disastrous reordering when applied to Line 7 and Line 8: the actual writing of the data would happen before the bucket is marked as full, allowing other threads to write to the same bucket. Likewise, reordering Line 8 and Line 9 would prematurely indicate that writing the data has completed.

Unfortunately, the ANSI C99 standard does not state any requirements on the memory model. The implementation would depend on the combination of CPU architecture and compiler. Our implementation uses the GNU gcc compiler for 64-bit x86 target platforms. A gcc built-in is used for the CAS operation and reads and writes from and to buckets are marked volatile.

Alg. 3 was modeled in PROMELA and checked for deadlocks with SPIN. One bug concerning the combination of write bit and memoized hash was found and corrected.

IV. Experiments

A. Methodology

We implemented the hash table of the previous section in our own model checking toolset LTSmin, which we discuss further in the following section. For our experiments, we reuse not only the input models, but also the `next-state` *implementation* of DiVinE 2.2. Therefore, a fair comparison with DiVinE 2.2 can be made. Furthermore, we performed experiments with the latest multi-core capable version of the model checker SPIN 5.2.4 [13] (DiVinE models were mechanically translated to SPIN's PROMELA input language). For our experiments, we chose full state space exploration via reachability as load generator for our state storage. Reachability exhibits similar access patterns as more complex verification algorithms, but reduces the code footprint and therefore potential pollution of our measurements with noise.

All model checkers were configured for maximum performance. For all tools, we compiled models to C with high optimization settings (`-O3`) (DiVinE also contains a model interpreter). SPIN's models were compiled with the following flags: `-O3 -DNOCOMP -DNOFAIR -DNOREDUCE -DNOBOUNDCHECK -DNOCOLLAPSE -DNCORE=`N `-DSAFETY -DMEMLIM=100000`; To run the models we used the options: `-m10000000 -c0 -n -w28`.

We performed our experiments on AMD Opteron 8356 16-core servers with 64 GB RAM, running a patched Linux 2.6.32 kernel.[2] All tools were compiled using gcc 4.4 in 64-bit mode with maximal compiler optimizations (`-O3`).

A total of 31 models from the BEEM database [18] have been used in the experiments (we filtered out models which were too small to be interesting, or too big to fit into the available memory). Every run was repeated at least four times, to exclude any accidental fluctuation in the measurements. Special care has been taken to keep all the parameters across the different model checkers the same. Especially SPIN provides a rich set of options with which models can be tuned to perform optimal. Using these parameters on a per-model basis could give faster results than presented here. It would, however, say little about the scalability of the core algorithms. Therefore, we decided to leave all parameters the same for all the models. We avoid resizing of the state storage in all cases by increasing the initial hash table size to accommodate 2^{28} states (enough for all benchmarked input models).

B. Results

Figure 2 shows the run times of only three models for all model checkers. We observe that DiVinE is the fastest model

[2]Experiments showed large regressions in scalability on newer 64-bit Linux kernels (degrading runtimes with 10+ cores). Despite being undetected since at least version 2.6.20 (released in 2007!), they were easily exhibited by our model checker. With a repeatable test case, the Linux developers quickly provided a patch: https://bugzilla.kernel.org/show_bug.cgi?id=15618

checker for sequential reachability. Since the last published comparison between DiVinE and SPIN [1], DiVinE has been improved with a model compiler. SPIN is only slightly slower than DiVinE and shows the same linear curve but with a gentler slope. We suspect that the gradual performance gains are caused by the cost of the inter-thread communication (see Table I).

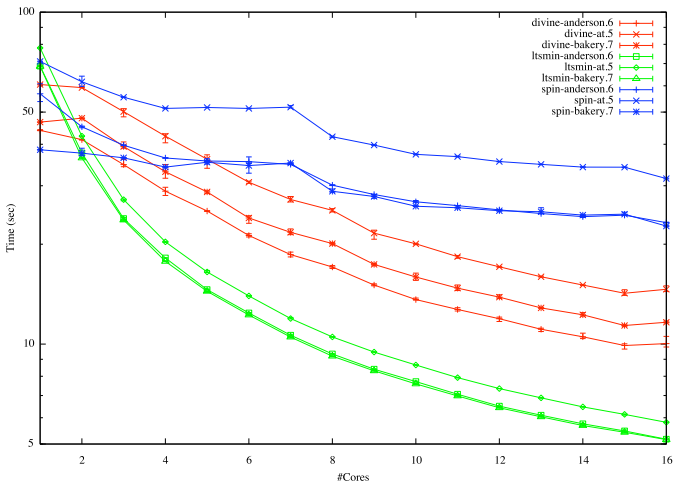


Fig. 2. (Log-scale) Runtimes in SPIN, LTSmin and DiVinE 2 (3 models)

LTSmin is slower in the sequential cases. We verified that the allocation-less hash table design causes this behavior; with smaller hash table sizes the sequential runtimes match those of DiVinE. We did not bother optimizing these results, because with two cores, LTSmin is already at least as fast as DiVinE.

Fig. 7, 8 and 9 show the speedups measured with LTSmin[3], DiVinE and SPIN (note that we normalize with T_{seq} of DiVinE, the fastest sequential tool). On 16 cores, LTSmin shows a two-fold improvement over DiVinE and a four-fold improvement over SPIN. We attribute the difference in scalability for DiVinE to the extra synchronization points needed for the inter-process communication by DiVinE. Recall that the model checker uses static state space partitioning, hence most successor states are enqueued at other cores than the one which generated them. Another disadvantage of DiVinE is its use of a management thread, which causes the regression at 8 and 16 cores.

SPIN shows inferior scalability even though it uses (like LTSmin) a shared hash table. SPIN also balances load based on *stack slicing*. We can only guess that the locking mechanism used in SPIN's hash table (region locking) are not as efficient as our lockless hash table. However, in LTSmin we obtained far better results even with the slower `pthread` locks. It might also be that stack slicing does not have a consistent granularity, because it uses the (irregular) search depth as a time unit (using the terms from Sec. II: $T(work(P_0, depth)) \gg T(work(P_1, depth))$).

Remark. A potential reason for the limited scalability of SPIN could be a memory bandwidth bottleneck. We tested this hypothesis by enabling SPIN's smaller, *collapsed* state vectors (`-DCOLLAPSE`). We carried out a full SPIN benchmark run

[3]Additional figures and more detail can be found in extended report [15].

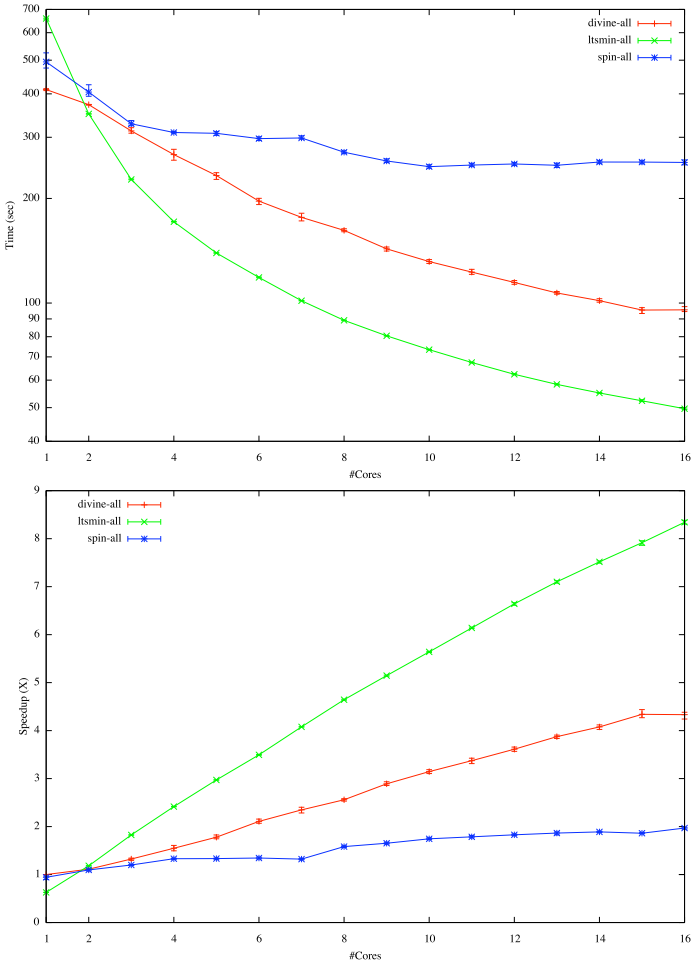


Fig. 3. Total runtime/speedup of SPIN, DiVinE 2.2 and LTSmin

with collapsing enabled and saw little improvement compared to the speedup results without `COLLAPSE`.[4] These results are consistent with the observation that LTSmin is faster, despite generally producing larger state vectors than both, SPIN and DiVinE (Table II): in LTSmin, each state variable gets 32-bit aligned (for API reasons, not performance).

Fig. 3 shows the total times and average speedups over all models and for all model checkers. Nineteen models could only be used because only for those, all tools report similar state counts (less than 20% difference; recall that for SPIN, models are translated from DVE to PROMELA).

C. Shared Storage Parameters

To verify our claims about the hash table design, we collected internal measurements and performed synthetic benchmarks for stress testing. First, we measured how often the write "lock" was hit. Fig. 4 plots the lock hits against the number of cores for several different sized models. For readability, only the worst-performing, and thus most interesting, models were chosen. Even then, the number of lock hits is a very small fraction of the number of `find-or-put` calls (equal to the number of transitions, typically in the hundreds of millions).

[4]http://fmt.cs.utwente.nl/~laarman/spin/

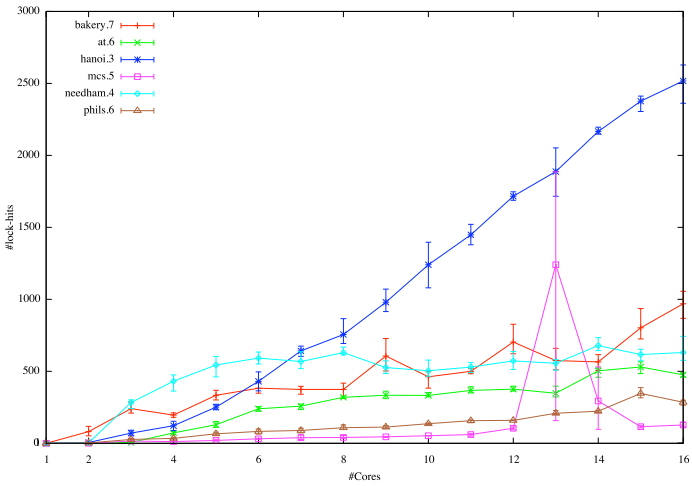


Fig. 4. Counting how often the algorithm "locks"

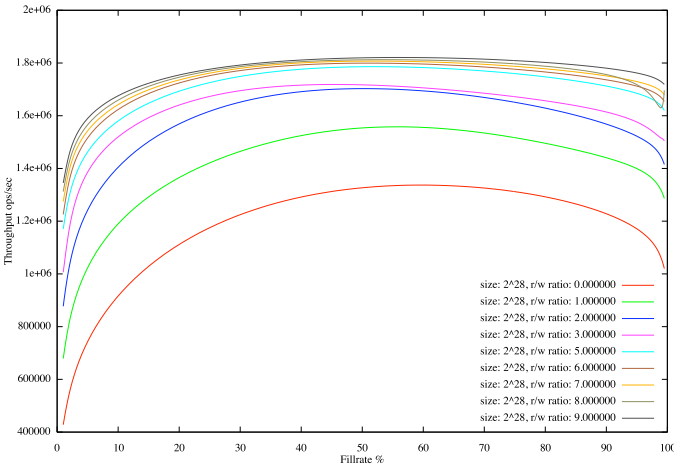


Fig. 5. Effect of fill rate and r/w-ratio on average throughput

We measured how the average throughput of Alg. 3 (number of `find-or-put` calls) is affected by the table *fill rate*, the table size and the read/write ratio. Fig. 5 illustrates the effects of different read/write ratios on the hash table using synthetic input data. The average throughput remains largely unaffected by a high fill rate, even up to 95 % (as for Fig. 6 in the Appendix, which plots the same lines for different table sizes). We conclude that the asymptotic time complexity of open-addressing hash tables poses little real problems in practice. However, an observable side effect of *oversized hash tables* is lower throughput for low fill rates due to increased cache misses. Our hash table design amplifies this effect because it uses a pre-allocated data array and no pointers. This explains the lower sequential performance of LTSmin.

We also measured the effect of varying the state vector size and did not find any noticeable change in the speedup behavior (except for the expected lower throughput due to higher data movement). This shows that hash memoization and a separate data array perform well. Walking-the-line probing shows better performance and scalability than double hashing alone, due to cache effects. Although slower on average, walking-the-line followed by double hashing beats simple linear probing at fill-rates above 95 % (in particular, on slower memory subsystems), because it leads to better distribution and thus fewer probes.

V. Discussion and Conclusions

We designed a hash table suitable for application in reachability analysis. We implemented it as part of a model checker together with different exploration algorithms (pseudo BFS and pseudo DFS) and explicit load-balancing. We demonstrated the efficiency of the complete solution by comparing the absolute speedups to SPIN 5.2.4 and DiVinE 2.2, both leading tools in this field. We claim two times better scalability than DiVinE and four times better than SPIN *on average* (Fig. 3), with individual results far exceeding these numbers. We also investigated the behavior of the hash table under different fill rates and found it to live up to the imposed requirements.

Limitations: Without the use of pointers the current design cannot easily cope with variably sized state vectors. In our model checker, this does not pose a problem because states are always represented by a vector of a static length. Our model checker LTSmin[5] can handle different front-ends. It connects to DiVinE-cluster, DiVinE 2.2, PROMELA (via NIPSVM [22]), mCRL, mCRL2 and ETF (internal symbolic representation of state spaces). Some of these input languages require variably sized vectors (NIPS). We solve this by an initial exploration which continues until a vector of stable size is found, and aborts when none can be found up to a fixed bound. So far, this limitation did not pose a problem.

For LTSmin, the results in the sequential case turn out to be around 20% slower than DiVinE 2.2. One of the culprits for this performance loss are the already mentioned suboptimal utilization of cache effects for small models (we verified that larger models suffer much less from this effect). Embracing pointers and allocation could be a potential remedy, however, it is unclear whether such a solution still scales when it actually matters (i.e., for large models).

Further performance is lost in an extra level of indirection (function calls) due to the design of LTSmin, which strictly separates the language front-end from the exploration algorithms. We are willing to pay this price in exchange for the increased modularity of our tool.

Discussion: We make several observations:

- We provide evidence that centralized state storage can be made to scale at least as well as static state space partitioning, contrary to prior belief [3].
- We also show that scalability is not as dependent on long state vectors and transition delays as earlier thought [12]. In fact, we argue that a scaling implementation performs better with smaller state vectors, because the number of operations performed per loaded byte is higher, thus closer to the strengths of modern multi-core systems.
- Shared state storage is also more flexible [3], for example allowing pseudo DFS (like the stack-slicing algorithm) and fast deadlock/invariant searches [20]. Moreover, it facilitates explicit load balancing algorithms, enabling the

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

exploitation of *heterogeneous systems*. From preliminary experiments with load balancing we conjecture that overhead is negligible compared to static load balancing.

- Performance-critical parallel software needs adaptation to modern architectures (steep memory hierarchies). The performance difference between DiVinE, SPIN and LTSmin is an indication. DiVinE uses an architecture which is directly derived from distributed model checking and the goal of SPIN was for "these new algorithms [...] to interfere as little as possible with the existing algorithms for the verification of safety and liveness properties" [12]. With LTSmin, we had the opportunity to tune our design to the architecture of our target machines, with excellent pay-off. We noticed that avoiding cache line sharing and keeping a simple design was instrumental in the outcome.
- Holzmann conjectured that optimized sequential code does not scale well [13]. In contrast, our parallel implementation is faster in absolute numbers and also exhibits excellent scalability. We suspect that the (entirely commendable) design choice of SPIN's multi-core implementation to support most of SPIN's existing features *unchanged* is detrimental to scalability.

Applicability: The components of our reachability can be reused directly for other model checking applications. The hash table and the load balancing algorithms can be reused to realize scalable multi-core (weak) LTL model checking [1], [2], symbolic exploration and space-efficient enumerative exploration. We experimented with the latter using *tree compression* [4] based on our hash table. Results are very promising and we intend to follow up on that.

Future work: By exploring the possible solutions and gradually improving this work, we found a wealth of variables hiding in the algorithms and the models of the BEEM database. As can be seen from the figures, different models show different scalability. A valid question is how much this can be improved.

By now we have some ideas where these differences come from. For example, an initial version of the exploration algorithm employed static load balancing by means of an initial BFS exploration and handing off the states from the last level to all threads. Several models where insensitive to this static approach, others, like `hanoi` and `frogs`, are very sensitive due to the shape of their state spaces. Dynamic load balancing did not come with a noticeable performance penalty for the other models, but `hanoi` and `frogs` are still in the bottom of the figures. However, we have yet to see the options exhausted to improve these results by shared-memory load balancing techniques, at least to the point that the shape of the state space of these models allow.

We did not consider other types of hash tables, like *Cuckoo hashing* or *Hopscotch hashing* [11]. Cuckoo hashing is an unlikely candidate, since it requires updates on many locations upon inserts, easily resulting in extraneous cache coherence overhead. Hopscotch hashing could be considered because it combines a low memory working set with constant lookup times even under higher load factors. However, Hopscotch hashing increases the memory working set for insertions, potentially sacrificing some speedup. It would still be interesting to investigate its performance relative to our hash table.

VI. Acknowledgements

We thank Anton Starikov and the CMS group at UTwente for making their cluster available for our experiments. We thank Petr Ročkai and Jiří Barnat for their support on the DiVinE toolkits, and for reading a draft of this paper. Cliff Click and Gerard Holzmann also gave helpful comments on a draft version. Elwin Pater implemented the bridge between DiVinE and LTSmin. The Linux developers provided patches remedying performance regressions on newer kernels.

References

[1] Jiří Barnat, Luboš Brim, and P. Ročkai. Scalable multi-core LTL model-checking. In *Model Checking Software*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[2] Jiří Barnat, Luboš Brim, and Petr Ročkai. DiVinE 2.0: High-Performance Model Checking. In *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*, pages 31–32. IEEE Computer Society Press, 2009.

[3] Jiří Barnat and Petr Ročkai. Shared hash tables in parallel model checking. *Electronic Notes in Theoretical Computer Science*, 198(1):79 – 91, 2008. Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007).

[4] Stefan Blom, Bert Lisser, Jaco van de Pol, and Michael Weber. A database approach to distributed state space generation. *Electron. Notes Theor. Comput. Sci.*, 198(1):17–32, 2008.

[5] Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and symbolic reachability. In *Proceedings of CAV 2010*, LNCS. Springer, 2010. (accepted for publication).

[6] Luboš Brim. Distributed verification: Exploring the power of raw computing power. In Luboš Brim, Boudewijn Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 23–34. Springer, August 2006.

[7] Cliff Click. A lock-free hash table. Talk at JavaOne 2007, http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf, 2007.

[8] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *Proceedings of CAV 2006*, pages 415–418, 2006.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3 edition, September 2009.

[10] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[11] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. *Distributed Computing*, pages 350–364, 2008.

[12] Gerard J. Holzmann. A stack-slicing algorithm for multi-core model checking. *Electronic Notes in Theoretical Computer Science*, 198(1):3 – 16, 2008. Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007).

[13] Gerard J. Holzmann and Dragan Bošnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Softw. Eng.*, 33(10):659–674, 2007.

[14] Cornelia P. Inggs and Howard Barringer. Effective state exploration for model checking on a shared memory architecture. *Electr. Notes Theor. Comput. Sci.*, 68(4), 2002.

[15] Alfons Laarman, Jaco van de Pol, and Michael Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. *ArXiv e-prints*, 1004.2772, April 2010.

[16] Witold Litwin. Linear hashing: a new tool for file and table addressing. In *VLDB '1980: Proceedings of the sixth international conference on Very Large Data Bases*, pages 212–223. VLDB Endowment, 1980.

[17] Michael Monagan and Roman Pearce. Parallel sparse polynomial multiplication using heaps. In *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, pages 263–270, New York, NY, USA, 2009. ACM.

[18] Radek Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[19] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. *Distributed Computing*, pages 108–121, 2005.
[20] Nageshwara V. Rao and Vipin Kumar. Superlinear speedup in parallel state-space search. *Foundations of Software Technology and Theoretical Computer Science*, pages 161–174, 1988.
[21] Peter Sanders. Lastverteilungsalgorithmen fur parallele tiefensuche. number 463. In *in Fortschrittsberichte, Reihe 10. VDI*. Verlag, 1997.
[22] Michael Weber. An embeddable virtual machine for state space generation. In D. Bosnacki and S. Edelkamp, editors, *Proceedings of the 14th International SPIN Workshop*, volume 2595 of *Lecture Notes in Computer Science*, pages 168–186, Berlin, 2007. Springer Verlag.

APPENDIX

Additional analysis can be found in an extended report [15].

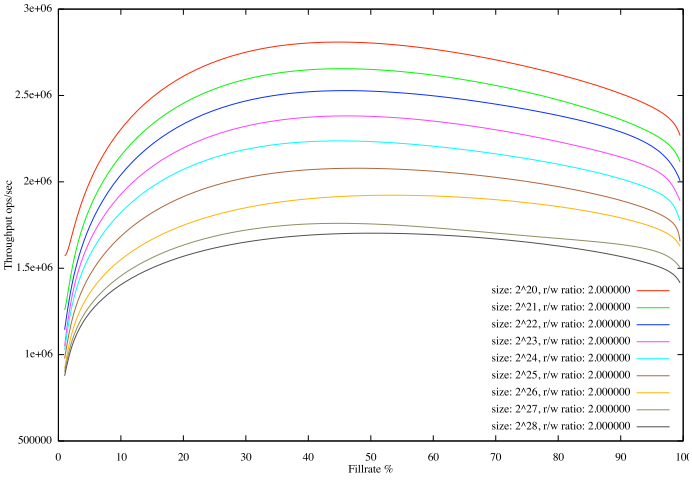


Fig. 6. Effect of size vs r/w-ratio on average throughput

TABLE II
MODEL DETAILS FOR DIVINE, LTSMIN AND SPIN

BEEM Model	Reachable States		State Vector Size [Byte]		
	DiVinE, LTSmin	SPIN	DiVinE	SPIN	LTSmin
anderson.6	18,206,917	18,206,919	25	68	76
at.5	31,999,440	31,999,442	20	68	56
at.6	160,589,600	—	20	—	56
bakery.6	11,845,035	11,845,035	24	48	80
bakery.7	29,047,471	27,531,713	24	48	80
blocks.4	104,906,622	88,987,772	23	44	88
brp.5	17,740,267	—	24	—	72
cambridge.7	11,465,015	—	60	—	208
elevator_planning.2	11,428,767	11,428,769	36	52	140
firewire_link.5	18,553,032	—	66	—	200
fischer.6	8,321,728	8,321,730	27	92	72
frogs.4	17,443,219	17,443,221	33	68	120
frogs.5	182,772,126	182,772,130	38	68	140
hanoi.3	14,348,907	14,321,541	63	116	228
iprotocol.6	41,387,484	—	43	—	148
iprotocol.7	59,794,192	—	47	—	164
lamport.8	62,669,317	62,669,317	22	52	68
lann.6	144,151,628	—	28	—	80
lann.7	160,025,986	—	35	—	100
leader_filters.7	26,302,351	26,302,351	36	68	120
loyd.3	239,500,800	214,579,860	18	44	64
mcs.5	60,556,519	53,779,475	26	68	84
needham.4	6,525,019	—	51	—	112
peterson.7	142,471,098	142,471,100	30	56	100
phils.6	14,348,906	13,956,555	45	140	120
phils.8	43,046,720	—	48	—	128
production_cell.6	14,520,700	—	42	—	104
szymanski.5	79,518,740	79,518,740	30	60	100
telephony.4	12,291,552	12,291,554	24	56	80
telephony.7	21,960,308	21,960,310	28	64	96
train-gate.7	50,199,556	—	43	—	128

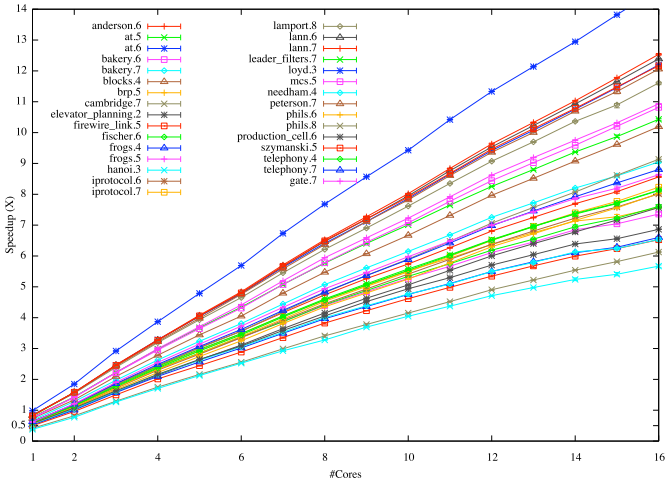


Fig. 7. Speedup of BEEM models with LTSmin (DiVinE as base case)

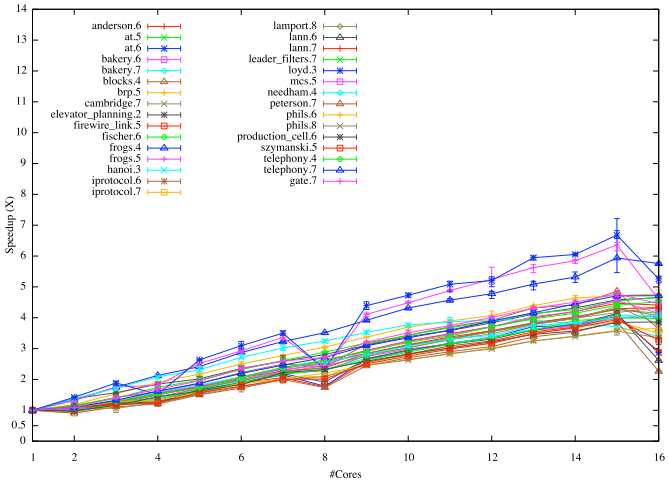


Fig. 8. Speedup of BEEM models with DiVinE 2.2 (DiVinE as base case)

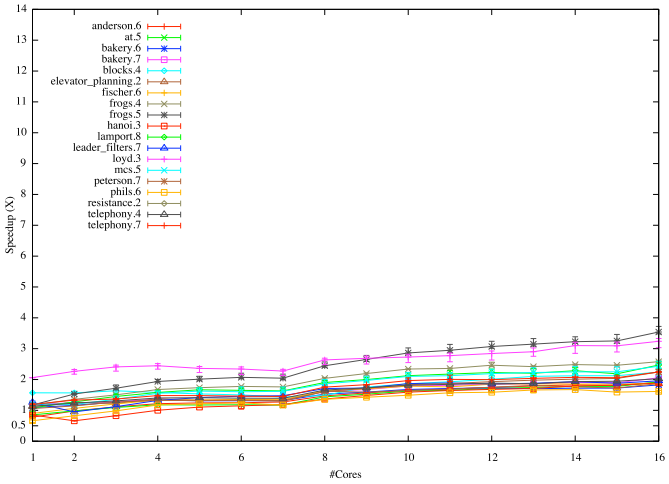


Fig. 9. Speedup of BEEM models with SPIN (DiVinE as base case)