Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 198 (2008) 79–91
www.elsevier.com/locate/entcs

Shared Hash Tables in Parallel Model Checking[1]

Jiří Barnat Petr Ročkai

Faculty of Informatics, Masaryk University, Brno, Czech Republic
barnat,xrockai@fi.muni.cz

Abstract

In light of recent shift towards shared-memory systems in parallel explicit model checking, we explore relative advantages and disadvantages of shared versus private hash tables. Since usage of shared state storage allows for techniques unavailable in distributed memory, these are evaluated, both theoretically and practically, in a prototype implementation. Experimental data is presented to assess practical utility of those techniques, compared to static partitioning of state space, more traditional in distributed memory algorithms.

Keywords: Hash tables, locking schemes, parallel

1 Introduction

Much of the extensive research on the parallelisation of model checking algorithms followed the distributed-memory programming model [5,4,12] and the algorithms were parallelised for networks of workstations, largely due to easy access to networks of workstations. Recent shift in architecture design toward multi-cores has intensified research pertaining to shared-memory paradigm as well.

A mostly straightforward transformation of distributed-memory algorithm into a shared-memory one, using several tailored techniques, is explored in [2]. In this paper, we intend to build on these results, further augmenting the selected distributed algorithms with extensions specific to shared-memory systems, especially those based on using a single shared storage for the explored graph.

For the experimental implementation, we have used DiVinE [3], specifically the multi-threaded, shared-memory version – as created for [2] – using the original DVE state-space generator. The code has been modified for the purposes of this paper.

[1] This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

doi:10.1016/j.entcs.2007.10.021

1.1 Shared-Memory Platform

Since in the paper, we will work with several assumptions about the targeted hardware architecture, we will describe it briefly first.

We work with a model based on threads that share all memory, although they have separate stacks in their shared address space and a special thread-local storage to store thread-private data. Our working environment is POSIX, with its implementation of threads as lightweight processes. Switching contexts among different threads is cheaper than switching contexts among full-featured processes with separate address spaces, so using more threads than there are CPUs in the system incurs only a minor penalty.

Critical Sections, Locking and Lock Contention. In a shared-memory setting, access to memory, that may be used for writing by more than a single thread, has to be controlled through use of mutual exclusion, otherwise, race conditions will occur. This is generally achieved through use of a "mutual exclusion device", so-called mutex. A thread wishing to enter a critical section has to lock the associated mutex, which may block the calling thread if the mutex is locked already by some other thread. An effect called resource or lock contention is associated with this behaviour. This occurs, when two or more threads happen to need to enter the same critical section (and therefore lock the same mutex), at the same time. If critical sections are long or they are entered very often, contention starts to cause observable performance degradation, as more and more time is spent waiting for mutexes.

Processor Cache: Locality and Coherence. There are currently two main architectures in use for Level 2 cache. One is that each processing unit has its completely private Level 2 cache (for the Symmetric Multiprocessing case) or there is a shared Level 2 cache for a package of 2 cores (designs with a Level 2 cache shared among 4 cores are not commercially available as of this writing). In bigger shared-memory computer systems, it is usual to encounter split cache, since they often contain on the order of 8-64 cores attached to a single memory block. In recent hardware, the basic building units are dual-core CPUs with shared cache, but among the different units, the caches are still separate. This idiosyncrasy of these architectures has important effects on performance and these will be discussed later in more detail.

Shared Memory Bus. Since the memory in SMP machines is attached to a single shared memory bus, the RAM access from different processors needs to be serialized. This caps total memory throughput of the system and at some point, the available memory bandwidth becomes the bottleneck of computation. This is an important factor for memory-intensive workloads, to which model-checking definitely belongs.

1.2 Algorithms

The algorithms used are not the main concern of this paper, but we nevertheless summarise OWCTY, as it was used in the implementation. Also, since we are using

the algorithm in somewhat non-standard setting, we have slightly modified some of its non-vital aspects – more details on those modifications will be described in Section 3.3. Short description of the original algorithm follow.

The algorithm [8] is an extended enumerative version of the **One Way Catch Them Young Algorithm** [11]. The idea of the algorithm is to repeatedly remove vertices from the graph that cannot lie on an accepting cycle. The two removal rules are as follows. First, a vertex is removed from the graph if it has no successors in the graph (the vertex cannot lie on a cycle), second, a vertex is removed if it cannot reach an accepting vertex (a potential cycle the vertex lies on is non-accepting). The algorithm performs removal steps as far as there are vertices to be removed. In the end, either there are some vertices remaining in the graph meaning that the original graph contained an accepting cycle, or all vertices have been removed meaning that the original graph had no accepting cycles.

The time complexity of the algorithm is $\mathcal{O}(h \cdot m)$ where $h = h(G)$. Here the factor m comes from the computation of elimination rules while the factor h relates to the number of global iterations the removal rules must be applied. Also note, that an alternative algorithm is obtained if the rules are replaced with their backward search counterparts.

2 Hash Tables in Model Checking

One of the traditional approaches, when exploring the state-space of an implicitly specified model, is that the algorithm starts from the initial state and using a transition function, generates successors of every explored state. Visited states are stored in a hash-table, to facilitate quick insertion of newly visited states and quick lookup of states that already have been visited.

The usual approach in distributed algorithms is to partition the state space statically, using a partition function [7,9] (which is usually in turn based on a hash function over the state representation). This partition function unambiguously assigns each state to one of the computation nodes. Same approach can be leveraged in shared-memory computation, where each thread of control assumes ownership of a private hash table, and potentially also a private memory area for storing actual state representations.

The described configuration is often the only feasible option, when dealing with distributed memory system, since cross-node memory access has to be either manually simulated using message passing, or even if available, is prohibitively expensive.

However, the situation in shared-memory systems is somewhat different, since all processors (and therefore threads of control) share a single continuous block of local memory, with uniform accessibility from all the CPUs and/or cores. This gives us two new options, compared to situation in distributed environment, namely, if several hash tables are used, threads can look into tables they don't own, and second, probably more interesting option is to have a single shared hash table, used by all the threads.

2.1 Implementation

The threaded version of DiVinE implements an internal collision resolution hash table with quadratic probing. The table is dynamically-sized with exponential growth (i.e., the size of the table doubles every time more space is needed). Originally, the threshold triggering table growth has been set as half-full, which gives minimum overhead of 2 key-sized cells per valid item, where in our case the key is a single pointer. Growing the table starts with allocating a new, double-sized table, iterating over all entries in the old table and rehashing them into the new, bigger one. This is a linear-time operation, amortised over insertions into the table. However, this property may have more far-fetched consequences in a setting where the table is shared among multiple threads.

A somewhat different approach for triggering the growth of the table has been implemented as part of the work on shared hash tables. The conditions are now twofold, first is that table is 75% full, the second is that there have been too many collisions upon insert, where too many is defined as $32 + \mathrm{sqrt}(\mathit{size})/16$. This parameter may be subject to further adujstment, although we haven't observed significant impact. This latter trigger produces more tighly packed tables, which may sometimes save time, especially since during the growth, all other processing is halted. Another possibility to reduce the number of grows is to increase the growth factor (this is an user-overridable setting and subject to empirical tuning).

2.2 Region Locking

There is a need for locking when multiple threads perform concurrent reads and updates of the table. Since the table is accessed very frequently, it is completely unfeasible to lock the whole table for each access, as this would lead to very high lock contention and, consequently, reduced performance. Therefore, a region locking scheme is devised, to only lock the region within which the update or lookup takes place. Special precautions are necessary for growing the table, since no updates at all are allowed during this window. The regions are fixed-size, so the number of regions grows linearly with the table size. There are two other options on how to organise locking, one being of fixed number of locks, which means the locking unit increases linearly with the hash table size, the second being a square-root based growth of both region size and number of locks.

Theoretical benefits of the first approach are that lock granularity and therefore contention should remain very low throughout program execution. Fixed number of locks makes competition for any given lock higher, although in theory, it should remain constant, as long as number of competing threads is constant. The square-root approach is a compromise between those two. All the methods are evaluated in the experimental section.

2.3 Lockless Shared Table

If implemented with no locking at all, an insertion may silently fail, i.e. it may be overwritten by a subsequent insert to a colliding position due to a race condi-

tion. However, this is not a fatal problem for reachability analysis, as observed in [17]. We have implemented a lockless hash table, but we have encountered severe scalability problems with large, statically sized tables (as opposed to dynamically growing tables). Since growing a lockless table is not implemented, this makes it hard to compare against the locking implementations, which can resize tables and therefore don't suffer from the large table problem. However, even lock-based tables, when statically sized, are highly detrimental to any scalability the system may be exhibiting. As of this writing, we haven't found the cause of the scalability issues with pre-sized tables, therefore more investigation is due.

3 State Space Partitioning

To distribute the workload of graph exploration (in case of safety checking) or cycle detection (in case of liveness checking), the state space is divided into parts, one for each of the worker threads (in the case of distributed computation, one for each cluster node).

3.1 Static Partitioning

The original shared-memory implementation used a partitioning scheme coming directly from the distributed world. Each state is uniquely assigned to a thread, based solely on the state representation. This means, that every time a state is generated, it is assigned to the same thread. Consequently, each thread can maintain its private hash table, where it stores all states it owns. This has an important side-effect of the thread being able to operate on the table without resorting to locking or critical sections. Same goes for the auxiliary state data (like predecessor count in OWCTY elimination) – no locking is necessary.

Another benefit is highly efficient use of processor cache, by making the ratio of hash table size to processor cache size much more favourable, than in the case of shared hash table. This consequently reduces memory load and improves throughput.

3.2 Dynamic Partitioning

The above static partitioning scheme suffers from high communication overhead, since as threads are added, number of cross-transitions (transitions that require inter-thread communication, because one of the states belongs to different thread than the other) grows rapidly.

A scheme using a different partitioning approach may be devised, when we are dealing with a single, shared table. Since the shared table allows any thread to lookup or update any state, it is no longer necessary to maintain the rule requiring each state to be unambiguously assigned to one of the threads. Instead, the thread that is examining a transition can decide on-the-fly whether to process it locally, or send it over to another CPU.

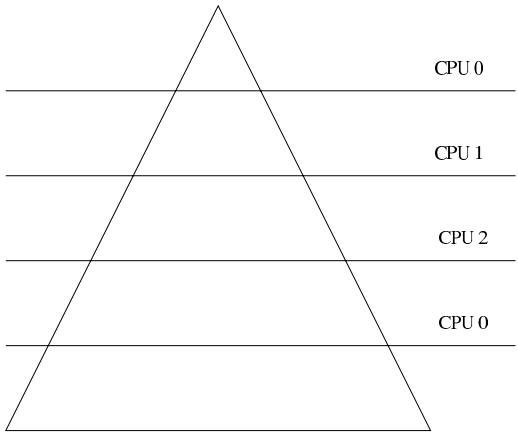

Fig. 1. Illustration of handoff partitioning scheme.

Modular partitioning. There are several possible approaches on how to partition the state space. A naïve implementation is to make every n-th transition a cross-transition, i.e. send it to a different thread. This makes for a great way to control the amount of cross transitions (and therefore explicit communication overhead). However, there are two problems with this approach. First, it reduces cache locality drastically, compared to that provided by static partitioning. In addition to losing the benefit of smaller hash tables (due to using a single big hash table), it also assigns states to threads virtually randomly, so it causes access to single state from different threads very often. This again reduces cache efficiency.

Handoff partitioning. This partitioning technique useful with DFS-based reachability analysis proposed in [13] is based on sending transitions to next thread when a certain "handoff depth" (depth of local DFS stack) is reached. This efficiently limits the amount of cross-transitions encountered, as they only appear every N levels of the pseudo-DFS tree, where N is the handoff depth or threshold. The actual threshold value is an option that needs to be empirically determined.

The technique has a much better state locality than the previous one, i.e. the chance that a given state is visited from a single thread several times is much higher. In Figure 1, a scheme of the resulting state distribution may be seen.

Another remarkable benefit of this scheme is the possibility to implement fairly efficient partial order reduction [15,16], as claimed in [13]. However, we have no such implementation and no comparison with other partial order reduction techniques, like [1,6].

Shared queue. Another possibility is to distribute states not using a partition function, but place them in a single shared BFS queue. This approach should achieve optimum load-balancing, although compromises may be necessary to strike a balance with locking overhead and contention.

3.3 Algorithm Impact

Through use of proper locking, all distributed algorithms can be used unmodified with shared hash table. However, the individual partitioning schemes place additional requirements on the algorithms, specifically on the visit order. The handoff technique requires a DFS stack and shared queue is specific to BFS.

Both the algorithms we have implemented are independent of order of visits, so can be run in both BFS and DFS order. These are reachability and OWCTY, although several other distributed algorithms share this property and could be therefore used in this setting. The parallel versions of Nested DFS [10] are not considered, since they do not use partitioning at all.

4 Experiments

Since there are no satisfactory profiling tools available for the kind of parallel workload we work with, we are mostly limited to measuring overall runtime of the algorithm implementations on various models using different parameters.

4.1 Methodology

The main testing machine we have used is a 16-way AMD Opteron 885 (8 CPU units with 2 cores each). All timed programs were compiled using gcc 4.1.2 20060525 (Red Hat 4.1.1-1) in 32-bit mode, using -O2. This limits addressable memory to 3GB, which was enough for our testing. The machine has 64GB of memory installed, meaning that none of the runs were affected by swapping.

For this paper, our main concern is speed and scalability, therefore we focus on these two parameters. Measurement was done using standard UNIX `time` command, which measures real and cpu times used by program. The real runtime is of particular interest, since this is the figure describing how long will the user wait for their results.

Acronym	Description	Property (LTL formula)
elevator	Motivated by elevator promela model from distribution of SPIN. The cab controller chooses the next floor to be served as the next requested floor in the direction of the last cab movement. If there is no such floor then the controller consider the oposite direction. (3 floors)	If level 0 is requested, the cab passes the level without serving it at most once. $G(r0 \implies (\neg l_0 U (l_0 U (\neg l_0 U (l_0 U (l_0 \wedge open))))))$
leader	Leader election algorithm based on filters. A filter is a piece of code that satisfy the two following conditions: a) if m processes enter the filter, then at most $m/2$ processes exit; b) if some process enter the filter, then at least one of them exits. (5 processes)	Eventually a leader will be elected. $F(leader)$
rether	Software-based, real-time Ethernet protocol whose purpose is to provide guaranteed bandwidth and deterministic, periodic network access to multimedia applications over commodity Ethernet hardware. It is a contention-free token bus protocol for the datalink layer of the ISO protocol stack. (5 Nodes)	Infinitely many NRT actions of Node 0. $G(F(nact0))$
peterson	Peterson's mutual exclusion protocol for N processes. (N=4)	Someone is in critical section infinitely many times. $G(F(SomeoneInCS))$
anderson	Anderson's mutual exlusion protocol for N processes. (N=6)	N/A

Table 1
Models and verified properties.

All the models we have used are listed in Table 1 including the verified properties. The models come from the `BEEM` database [14] that contains the models in DiVinE-native modeling language.

4.2 Comparison of Partitioning Methods

First, we have measured reachability timings for the model peterson_1 using four approaches: BFS with static partitioning, BFS with modular partitioning, DFS with handoff partitioning and DFS with handoff partitioning and preallocated hash table (5 million cells, to accomodate the model easily). Also note that since the separate hash tables for BFS get smaller as the number of threads increases, the growth overhead drops slightly. We have also measured runtimes of OWCTY on the same model using analogical conditions. The results may be seen in Figure 2. From the figures, we see that for small number of cores, the dynamic partitioning schemes perform better, but are consistently "outscaled" by the statically partitioned BFS. Surprizingly, the modular partitioning scheme is not as far behind handoff as we have expected in some cases, although it still is the slowest and least scalable one.

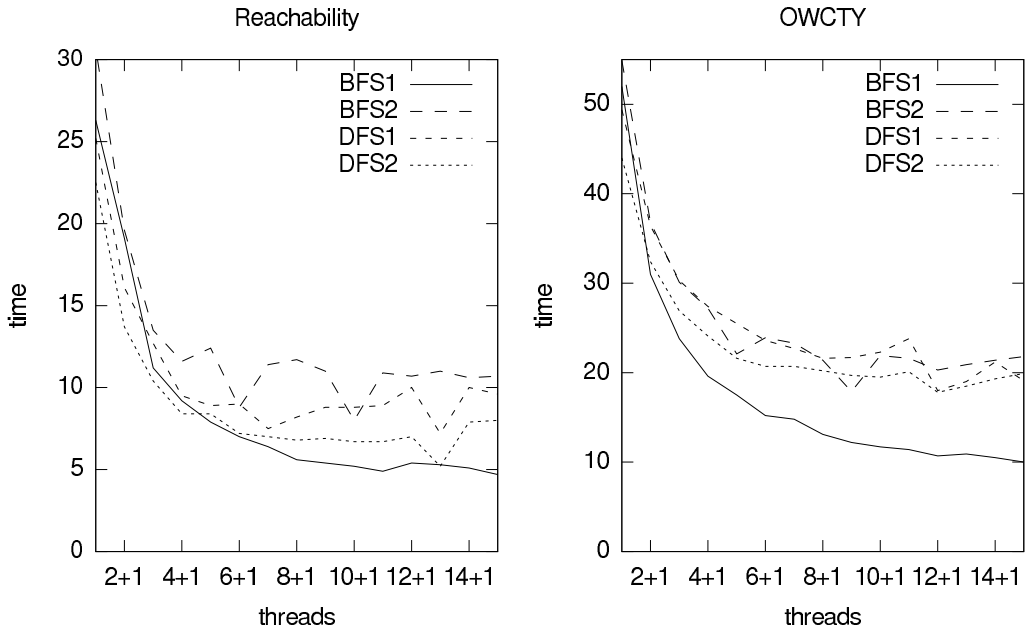

Fig. 2. Comparing scalability of reachability and OWCTY, using static and dynamic partitioning. BFS1 uses static partitioning, BFS2 uses modular partitioning, DFS1 uses handoff partitioning and finally DFS2 uses handoff with preallocation. Model used is peterson$_1$.

4.3 Effect of Handoff Threshold

To determine the practical effect of handoff threshold on actual runtimes of the algorithms, we have measured runtimes of reachability and OWCTY with a matrix of parameter combinations using DFS and handoff partitioning. Figures 4 and 5 visualise data from the smaller model (peterson, on the order of 2 million states). We observe, that handoff does not affect the runtime significantly, unless set to very high – around 200, it starts to negatively affect scalability, being unable to provide sufficient load balancing.

We have also tried with a bigger model (anderson, on the order of 18 million states) using reachability. The results are available in Figure 6. Here, handoff depths up to 4096 seem to manage to spread the load evenly across threads, while at very low handoff (1-4), the number of cross-transitions slows the computation down significantly.

4.4 Effect of Locking Scheme

In Figure 7, we present the behaviour of DFS reachability using various locking schemes, on top of a shared storage, using handoff partitioning (using default handoff depth of 50). All locking schemes were evaluated both using preallocated hash table and a growing hash table. From the picture, we see that the locking scheme basically does not affect runtime in any significant way.

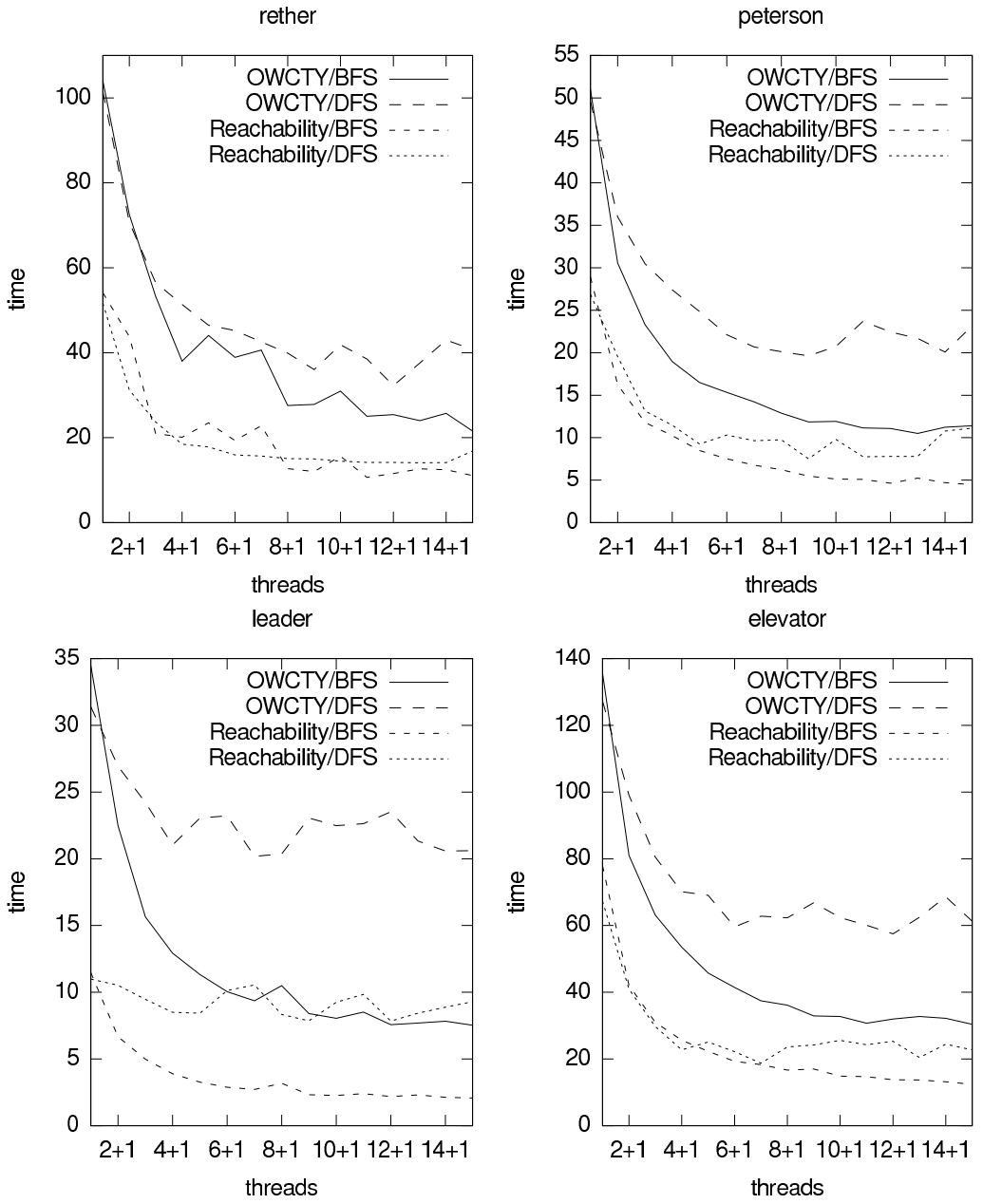

Fig. 3. Comparing scalability of reachability and OWCTY, using static and dynamic partitioning. BFS uses static partitioning, DFS1 uses handoff partitioning.

5 Conclusions

We have implemented several techniques dealing with use of shared hash tables in shared-memory parallel model checking. They have been compared, both theoretically and practically, to approaches known from distributed world.

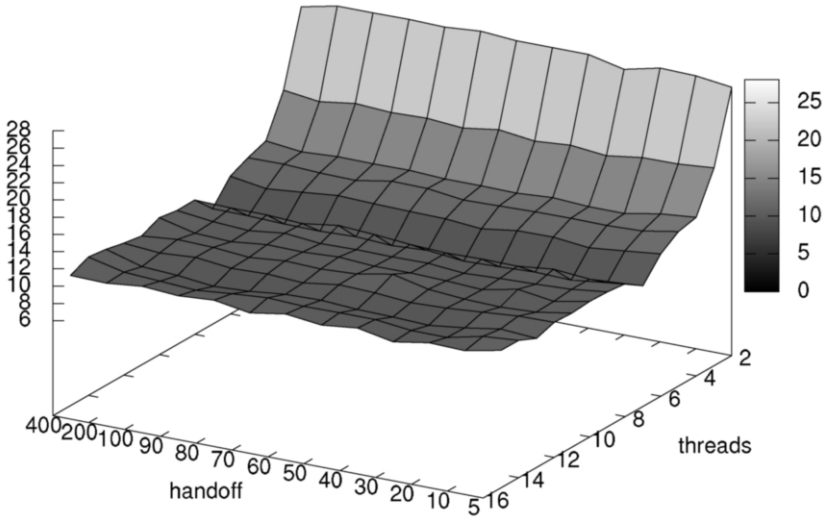

Fig. 4. Measuring effect of different handoff depths at runtimes and scalability of reachability on a small model (peterson_1). Note that the handoff axis is reversed!

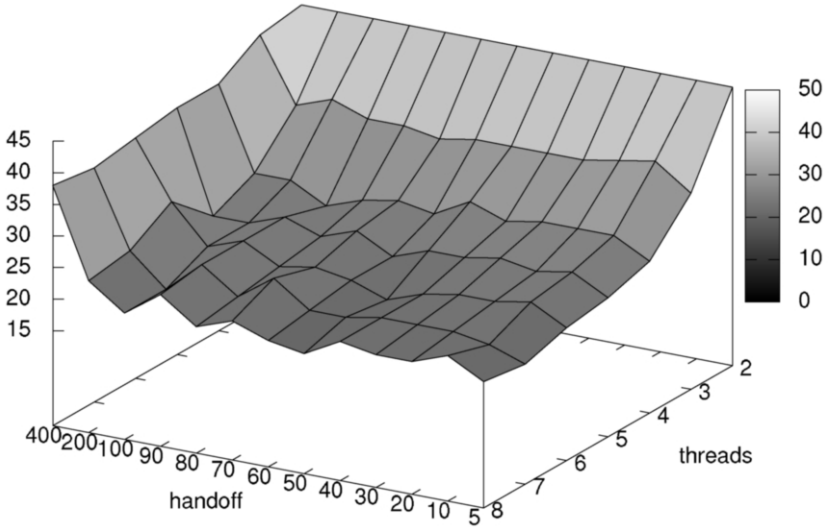

Fig. 5. Measuring effect of different handoff depths at runtimes and scalability of OWCTY on a small model (peterson_1). Note that the handoff axis is reversed!

In an environment with fairly low communication overhead, the different schemes did not vary as much as we have originally anticipated. The motivation behind the research was to improve performance and scalability of our parallel, shared-memory model checking platform based on DiVinE. However, the results have been less than convincing.

Although the schemes based on shared hash table, depth-first traversal and handoff partitioning have performed better on smaller number of threads (in the range of 1-8 threads), their utility in improving scalability over 8 cores is basically nonexistent. Breadth-first traversal with static partitioning, as used in distributed-

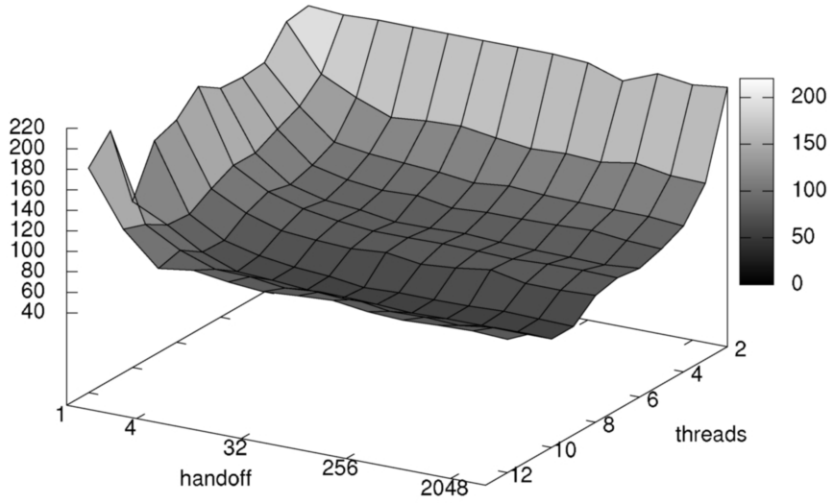

Fig. 6. Measuring effect of different handoff depths at runtimes and scalability of reachability. Big model (anderson). The handoff scale is logarithmic.

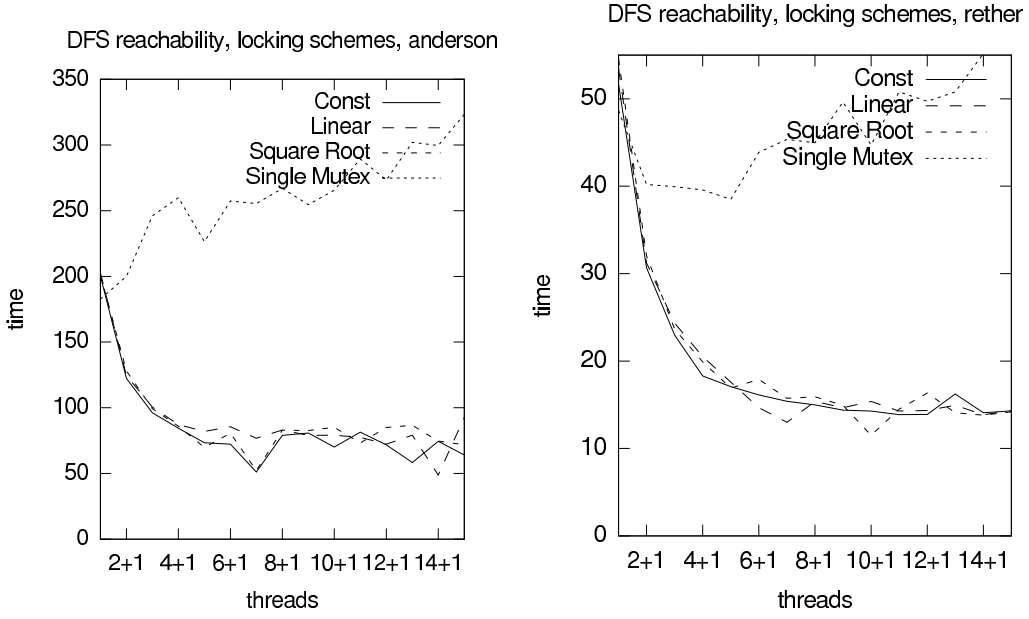

Fig. 7. Comparing locking methods on a reachability run over anderson and rether.

memory systems, out-scales them in these situations, by a not insignificant margin in some cases.

The main results therefore are, that communication overhead plays a role less important in scalability of shared-memory implementation, than previously believed. Second, that approaches known from distributed-memory architectures may be of practical utility to projects pursuing scalable shared-memory model-checking tool.

Since the results hint at a different source of limited scalability in shared memory

systems than pure communication overhead, we will pursue further research on this problem. The candidates for investigation include suboptimal implementation (eg. false sharing or locking problems) and hardware architecture limitations.

We have already identified and mitigated several problems impeding scalability in various scenarios, including false sharing and excessive thread migration among available cores caused by kernel scheduler. The general pattern we have observed is, that improvements in scalability are gradual and that there is no proverbial silver bullet, that would solve all the scalability issues at once.

References

[1] Barnat, J., L. Brim and J. Chaloupka, *From Distributed Memory Cycle Detection to Parallel LTL Model Checking*, Electronic Notes in Theoretical Computer Science **133** (2005), pp. 21–39.

[2] Barnat, J., L. Brim and P. Ročkai, *Scalable Multi-Core LTL Model-Checking*, in: *Proc. of SPIN 2007, to appear*, LNCS **4595** (2007), pp. 197–203.

[3] Barnat, J., L. Brim, I. Černá, P. Moravec, P. Ročkai and P. Šimeček, *DiVinE – A Tool for Distributed Verification (Tool Paper)*, in: *Computer Aided Verification*, LNCS **4144/2006** (2006), pp. 278–281.

[4] Behrmann, G., T. S. Hune and F. W. Vaandrager, *Distributed timed model checking — how the search order matters*, in: *Proc. 12th Conference on Computer-Aided Verification CAV00*, LNCS **1855** (2000), pp. 216–231.

[5] Brim, L. and J. Barnat, *Distribution of explicit-state ltl model-checking*, in: T. Arts and W. Fokkink, editors, *8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, Electronic Notes in Theoretical Computer Science **80** (2003).

[6] Brim, L., I. Černá, P. Moravec and J. Šimša, *Distributed Partial Order Reduction of State Spaces*, in: *3rd International Workshop on Parallel and Distributed Methods in verifiCation*, 2004.

[7] Caselli, S., G. Conte and P. Marenzoni, *Parallel state space exploration for GSPN models*, in: G. de Michelis and M. Diaz, editors, *Applications and Theory of Petri Nets 1995*, LNCS **935** (1995), pp. 181–200.

[8] Černá, I. and R. Pelánek, *Distributed explicit fair cycle detection (set based approach)*, in: T. Ball and S. Rajamani, editors, *Model Checking Software. 10th International SPIN Workshop*, Lecture Notes in Computer Science **2648** (2003), pp. 49 – 73.

[9] Ciardo, G., J. Gluckman and D. Nicol, *Distributed State Space Generation of Discrete-State +Stochastic Models*, INFORMS Journal on Computing **10** (1998), pp. 82–93.

[10] Courcoubetis, C., M. Vardi, P. Wolper and M. Yannakakis, *Memory-Efficient Algorithms for the Verification of Temporal Properties*, Formal Methods in System Design **1** (1992), pp. 275–288.

[11] Fisler, K., R. Fraer, G. Kamhi, M. Y. Vardi and Z. Yang, *Is there a best symbolic cycle-detection algorithm?*, in: *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, LNCS **2031** (2001), pp. 420–434.

[12] Garavel, H., R. Mateescu and I. Smarandache, *Parallel State Space Construction for Model-Checking*, in: M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model +Checking of Software (SPIN'2001)*, LNCS **2057** (2001), pp. 216–234.
URL citeseer.nj.nec.com/474094.html

[13] Holzmann, G., *The Design of a Distributed Model Checking Algorithm for SPIN*, in: *FMCAD, Invited Talk*, 2006.

[14] Pelánek, R., *BEEM: BEnchmarks for Explicit Model checkers*, http://anna.fi.muni.cz/models/index.html (2007).

[15] Peled, D., *Ten years of partial order reduction*, in: *Proceedings of the 10th International Conference on Computer Aided +Verification* (1998), pp. 17–28.

[16] Valmari, A., *Stubborn set methods for process algebras*, in: *Proceedings of the DIMACS workshop on Partial order methods in +verification* (1997), pp. 213–231.

[17] Weber, M., private communication (2007).