A Stack-Slicing Algorithm for Multi-Core Model Checking

Gerard J. Holzmann [1,2]

NASA/JPL Laboratory for Reliable Software
4800 Oak Grove Drive, Pasadena, CA 91109, USA

Abstract

The broad availability of multi-core chips on standard desktop PCs provides strong motivation for the development of new algorithms for logic model checkers that can take advantage of the additional processing power. With a steady increase in the number of available processing cores, we would like the performance of a model checker to increase as well – ideally linearly. The new trend implies a change of focus away from cluster computers towards shared memory systems. In this paper we discuss the multi-core algorithms that are in development for the SPIN model checker.

Key words: Multi-core systems. Distributed systems. Multi-threaded programming. Software verification. Logic model checking. Cluster computers.

1 Introduction

A new set of algorithms [7,8] is currently in development to support multi-core verifications with the SPIN model checker [5]. A guiding principle in the design of these new algorithms has been to interfere as little as possible with the existing algorithms for the verification of safety and liveness properties. The extensions are designed to preserve most of the existing verification modes and optimization choices, including, for example, partial order reduction, bitstate hashing, and hashcompact state storage. The basic computational complexity of the verification procedure also remains unchanged. This means that the verification of all correctness properties remains linear in the size of the state graph, when parts of the search are done in parallel. The SPIN algorithms

[1] The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, as part of NASA's ESAS Project 6G on Reliable Software Engineering.
[2] Email: gerard@spinroot.com

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

are known to be efficient, and we would like to preserve this advantage. As many have found, it can be hard to outperform a single-core run of Spin with standard optimizations enabled, even when using large numbers of processing cores with multi-core algorithms.

In this paper we explore one of the new algorithms we are considering in a little more detail. We focus on the algorithm for the verification of safety properties, which is based on a stack slicing method that has some unexpected benefits beyond the intended purpose of scaling performance up to linearly with the number of available CPU cores. We will first summarize this stack slicing algorithm, show some performance results, and then discuss some of the more surprising features of this algorithm.

2 The Stack Slicing Algorithm

A distributed algorithm tries to achieve a number of different objectives. Some of the more important ones are to achieve:

- an even distribution of the work across the available CPUs, so that all CPUs do roughly the same amount of work (load balancing),
- maximal independence between the work done on the different CPUs, so that most of the work can be done concurrently, and
- minimal communication overhead.

Some amount of overhead is inevitable, for instance to allow for the transfer of work from one CPU to another, but clearly any time lost to the maintenance of the multi-core infrastructure must be regained through the performance of work done in parallel. Less overhead means less pressure to makeup for the lost time. The reverse of this is that at some point the overhead can become so large that we are better off doing a single-core instead of a multi-core search.

Figure 1 shows the pseudo-code for a standard depth-first search process, as it is used for the verification of safety properties in the Spin model checker. The search starts by pushing the initial system state onto the search stack, and entering the state in the global state table. It then proceeds by recursively exploring successor states until all reachable states have been visited. Within the `Add_Statespace` routine, basic safety checks on newly reached system states can be performed, and correctness violations can be reported. A point in favor of the depth-first search procedure is that when an error is found, a complete step-by-step counter-example of all actions that lead up to the violation is easily generated by reading off the execution steps stored on the depth-first search stack `D`.

A modified depth-first search for the verification of safety properties, as implemented in Spin version 5.0, is illustrated in Figure 2. The CPUs are connected in a logical ring, where each CPU can hand off work only to its right neighbor (rn), counting modulo the number of available CPU cores (NCORE), as illustrated in Figure 3. To connect the CPUs, we introduce one work queue

```
 1 Stack D = {}
 2 Statespace V = {}
 3
 4 Start()
 5 {
 6      Add_Statespace(V, s0)
 7      Push_Stack(D, s0)
 8      Search()
 9 }
10
11 Search()
12 {
13      s = Top_Stack(D)
14      for each (s,l,s') in T
15      {   if (In_Statespace(V, s') == false)
16          {   Add_Statespace(V, s')
17              Push_Stack(D, s')
18              Search()
19      }   }
20      Pop_Stack(D)
21 }
```

Fig. 1. Standard Depth-First Search.

per CPU, in shared memory, to store the handoff states. By using a logical ring, we can ensure that each work queue has only one reader and one writer, which means that we can implement the associated data structures without any locks for maximal efficiency. In the modified algorithm we also added two integer variables, one to count the number of execution steps from the local root of the search (called `Depth`) and one to set a default depth at which a state transfer to another CPU core will be attempted (called `Handoff`).

The `Start()` routine is initiated on each CPU, with a different `core_id` number in the range 0..`(NCORE-1)` passed to each one. The CPU with `core_id` 0 starts the search in the usual way by pushing the initial system state onto its search stack and calling its recursive search procedure. The main difference in the depth-first search procedure itself can be found on lines 28-30, where we check if the preset `Handoff` depth has been exceeded. If it has, we check if the target work queue has slots available and if so we hand off the state to that CPU by copying it (in shared memory) into the target queue. The search now immediately backtracks and starts exploring other reachable system states, without waiting for the subtree below the handoff state to be fully explored. Note that the handoff is suppressed if the target work queue is full, in which case the neighbor CPU already has a sufficient amount of pending work so nothing more can be gained from passing it still more work to do. In this case the CPU considered will continue the search locally, while remaining prepared

```
1 Stack D = {}                  /* in local memory */
2 Statespace V = {}             /* in shared memory */
3 Queue wq[NCORE]               /* in shared memory */
4 int Depth = 0, Handoff = 20   /* in local memory */
5
6 Start(int core_id)
7 {
8       if (core_id == 0)
9       {   Add_Statespace(V, s0)
10          Push_Stack(D, s0)
11          Search(core_id)
12      }
13      while (NotTerminated)
14      {   if (NotEmpty(wq[core_id]))
15          {   s = First_State(wq[core_id])
16              Push_Stack(D, s)
17              Search(core_id)
18      }   }
19 }
20
21 Search(int core_id)  /* rn: right neighbor in logical ring */
22 {    int rn = (core_id + 1) % NCORE;
23       Depth++
24       s = Top_Stack(D[core_id])
25       for each (s,l,s') in T
26       {    if (In_Statespace(V, s') == false)
27            {    Add_Statespace(V, s')
28                 if (Depth > HandOff && NotFull(wq[rn]))
29                 {    Handoff_State(wq[rn], s')
30                 } else
31                 {    Push_Stack(D, s')
32                      Search(core_id)
33        }    }   }
34        Pop_Stack(D)
35        Depth--
36 }
```

Fig. 2. Modified Depth-First Search: Stack-Slicing Algorithm.

to hand off any future successor to this state at a later point in the search as soon as slots open up in the target work queue.

Once the search process has been completed, the search returns to the `Start()` routine and the next step is to check in the work queue for the CPU to see if any states were handed off to it, which are then explored in the

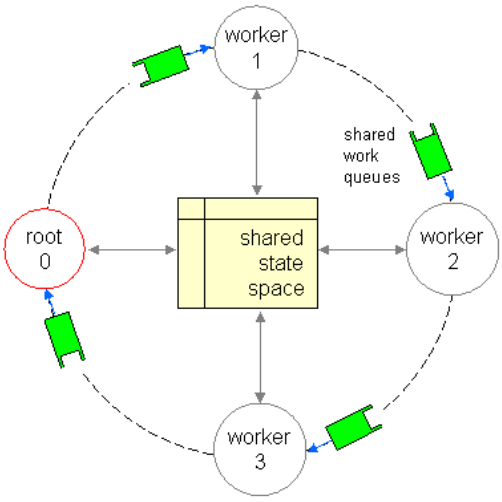

Fig. 3. Logical Ring Structure for the CPUs in a Multi-Core System.

same manner. Of course, the input work queue being empty is not a sufficient condition to terminate the search, so a distributed termination detection algorithm must be added to this basic design to make sure that the search process can terminate correctly. Termination detection can be done with any of the standard algorithms that have been developed for this purpose, so we will not discuss this further here. The implementation in Spin 5.0 is based on a variant of Dijkstra's treatment of Safra's algorithm [1]. A verification model of this algorithm can be proven correct with Spin itself, providing a curious example of a case where a verification tool can be used to prove the correctness of part of its own implementation.

Although we focus on shared memory systems here, the stack slicing algorithm and the logical ring structure used, can easily be extended further for the use on cluster computers. The modification to the ring structure that makes this possible, supported as an option in Spin 5.0, is illustrated in Figure 4. To form a logical ring that spans more than one PC in a cluster arrangement, we replace one node in the ring on each PC with a proxy. The proxies on neighboring machines collaborate by mirroring the contents of the work queues they are connected to between the PCs. The remaining "worker" nodes in each PC remain unchanged, and can be completely unaware that part of the computation is done on a distant system. Clearly, it would be inefficient to force all PCs to update states in a single shared statespace, so in this case each PC will maintain a separate state space that is only shared among the workers that execute on the same PC. This can lead to some redundancy, but in most cases the overhead of additional traffic across the network that would be necessary to maintain a single state space would introduce greater inefficiencies.

Startup and termination of the search process works as before, without any change. To facilitate the transfer of states without separate encoding and decoding of state information, the simplest method is to use binary compatible

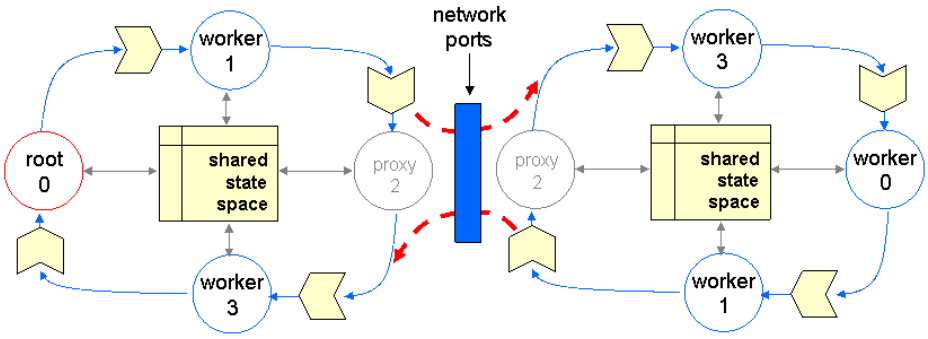

Fig. 4. Extended Ring Structure for Distributed Model Checking on a Compute Cluster.

systems that can execute precisely the same model checking code, and use the same memory layout for states. In our implementation, therefore, the binary of the model checker itself is send from the PC that starts the search to each participating PC as part of the search initialization. Another consideration in this setup is the speed of the network that connects the PCs. The best guarantee for adequate performance is to use a fast network with 1 to 10 Gbps bandwidth, to minimize the time lost to transfer states between PCs. The time required should be close to the time required for in-memory state transfers. Needless to say, on a standard 100 Mbps network optimal performance is not easily realized.

3 Generating Counter-Examples

One feature of the classic depth-first search procedure, illustrated in Figure 1, needs extra attention for the modified search procedure. This is the ability to generate counter-examples when an error state is found. Since each CPU only retains a small portion of the stack, it can no longer trace a path back to the original initial system state by reading off the steps contained in its local search stack. By default, the new algorithm therefore only retains the ability to recreate part of the counter-example, and in particular the final few steps leading to the error state, which fortunately is often sufficient for the diagnosis of errors.

It is possible to recreate the ability to generate also *full* counter-examples by adding an extra data structure, at the price of increasing memory use and the average runtime. If the search is performed in this mode, each CPU maintains a pointer into a data structure, which is best described as a *stack tree*. The stack tree is maintained in shared memory. When a state is handed off to another CPU, the corresponding pointer into the stack tree is passed as well. Each CPU adds a frame to the stack tree when executing a forward step in the depth-first search. When the search backtracks, the frames are not necessarily removed, though, but only a pointer is updated to keep track of the frame that

Table 1
Performance of Stack Slicing Algorithm (runtimes in seconds)

#Cores:	1	2	3	4	5	6	7	8
Leader	364.0	222.0	158.0	129.0	112.0	102.0	103.0	102.0
Tpc	99.4	73.6	58.0	50.4	45.3	41.2	39.4	36.1
RefModel	376.0	189.0	128.0	96.7	77.0	64.2	56.1	50.4

corresponds to the current point in the search. When an error state is reached, a path through the global stack tree back to the original initial system state can now be found to produce a full counter-example. Note that a stack frame can only safely be removed if none of the CPUs could need the frame anymore to generate a counter-example at any point during the search, including CPUs to which successor states were transferred either directly or indirectly. Each stack frame in the tree needs to contain only minimal information about the search path: the *id* of the process performing an execution step (one byte) and the *id* of the transition that was executed (typically a short integer). On a cluster system, the construction of full counterexamples requires a few more steps, to handle the case where the error trail crosses PC boundaries, but the basic procedure remains the same.

Stack frames that become redundant as the search progresses can be recycled with a garbage collection process, e.g., by maintaining a reference count in each frame that records how many successors may still be relying on it. When the count drops to zero, the frame can be recycled. Garbage collection introduces the need for locking, though, which can negatively impact overall runtime performance.

4 Properties of the Stack Slicing Algorithm

The performance of the slice stack algorithm is often surprisingly good (surprising for a relatively simple load balancing method and its minimal intrusion on the existing depth-first search process implemented in Spin). The handoff depth simultaneously provides locality and independence between cores, and trivial load balancing across cores. There are interesting engineering tradeoffs to be made. Note for instance that larger values for the handoff depth can give more independence in the search, and lower the overhead of state transfers between CPU cores, while shorter values can provide better load balancing.

Scaling with Available Cores: A representative result for the performance of the stack slicing algorithm in the verification of safety properties is shown in Figure 5 and Table 1. Figure 5 shows the percentage of time of a single-core run that is used when the number of cores is increased to 2, 4, and 8, for three different models. The top curve (solid) is for a small model of a phone switch (tpc), with 32.9 million reachable system states. The next curve (dashed)

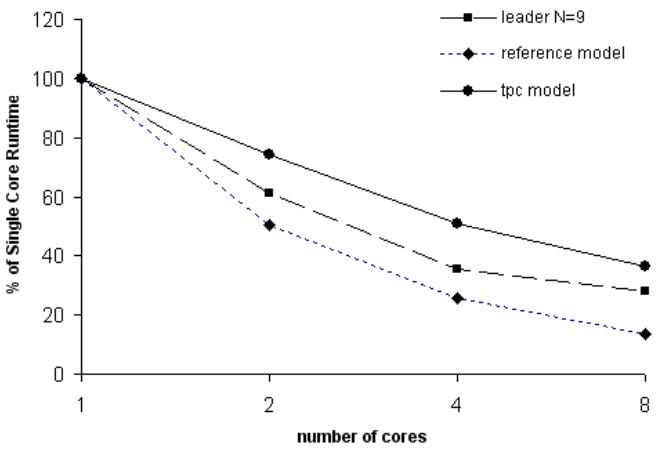

Fig. 5. Multi-core safety verification with SPIN 5.0 stack slicing algorithm. The curves show the percentage of runtime used by the stack slicing algorithm on multiple cores, compared with a single-core run (i.e., of a standard SPIN verification) for three different models.

is for the standard leader election model from the SPIN distribution, with 9 processes, which (without partial order reduction) generates a state space of 33.6 million reachable system states. The bottom curve is for a reference model [8] that allows us to control key structural parameters of the state space generated, such as the size of a state, the transition delay, and the average number of successors per state. The parameters for this reference model were chosen to produce near optimal performance of the multi-core algorithm. The state size chosen was 200 bytes, the average number of successor states was 8, and the transition delay was approximately 16 μseconds. The number of states generated by the reference model is 500,0000. The raw performance numbers for these tests, in seconds of runtime, are given in Table 1. Note that for the reference model, scaling is close to linear. On 4 cores, the optimal runtime would be $376/4 = 94$ seconds, and we measure 96.7 seconds; on 8 cores, the optimum would be 47 seconds, and we measure 50.4 seconds.

Stern-Dill Approximation: The minimal handoff depth is 1, and can be used to reproduce a search strategy similar to the original Stern-Dill algorithm [10]. This type of immediate handoff strategy, though, gives the least amount of locality and induces the greatest overhead and is therefore unattractive in this setting. There is also a maximal handoff depth. If we have N processing cores and the depth of the reachability graph of all system states is D execution steps, then we cannot hope to achieve proper load balancing if the handoff depth is set larger than D/N. Typically, especially for larger verification problems where a distributed verification algorithm can be most beneficial, D is in the order of 10^5 to 10^7 steps. This means that for a number of processing cores in the range of 10^2 to 10^3 cores, a handoff depth in the range of $10^1..10^2$ will be effective. Experiments show that the performance of the stack slicing

algorithm [8] is not very sensitive to the precise value chosen, which means that in most cases a fixed default value (the Spin implementation uses the value 20) will suffice to realize a performance speedup. In more exceptional cases, the user can provide a different value for the handoff depth to optimize performance.

Short Counter-Examples: The stack slicing algorithm can be understood as an interesting combination of a depth-first and a breadth-first search procedure. Note that when a CPU hands off a state to another CPU, it immediately backtracks and starts the exploration of other states that lie within the handoff depth limit of the local stack. This means that the search of all states reachable within H steps from the initial system states can be completed before all states have been explored that lie deeper in the search tree. An error state reachable within H steps, therefore, can be found faster than in a regular depth-first search, leading to shorter counter-examples being generated.

Short Stacks: Another unexpected benefit of the stack slicing method is that local stacks that must be maintained within each processing core can be quite small, and become independent of the depth of the global state graph itself. This decoupling can mean the difference between a tractable verification and one that is intractable. In large verification models, especially those with embedded C code with large amounts of matched and unmatched external state data [6], the regular depth-first search stack can contain tens of kilobytes of data. For deeper search trees, the amount of memory necessary to perform the basic search process can quickly exceed any amount of memory necessary to build the global state graph, especially when using aggressive state compression techniques such as bitstate hashing or hashcompact compression. Note that the data on the search stack cannot easily be compressed (and in no case with lossy techniques). This means that a single-core search for some of these models will quickly exhaust all available memory and fail to complete, while a multi-core search completes easily, using only a fraction of the amount of memory. Despite the fact of using only short local stacks, the multi-core search can retain the ability to construct full counter-examples, as described earlier. The data that needs to be preserved in the frames of the stack tree is comparatively small and typically restricted to just 2 words of memory, one word to store the process and transition*id*s, and one pointer to record the immediate predecessor stackframe along the current search path.

This extra capability of the stack slicing algorithm to handle verification models with large amounts of embedded code and data is unexpected, but it leads to the idea that the implementation of this type of algorithm could also be of interest even on a single core system. Nothing in the search algorithm needs changing to run the algorithm in this mode. The logical "ring" of CPU cores in this case contains just a single CPU, and the CPU is its own right neighbor. When the CPU "hands off" a state, it merely places it in its own work queue for later exploration. In effect, this means that the algorithm now maintains both a depth-first stack and a breadth-first queue that are used

jointly to perform the search, and that combine some of the benefits of each search mode.

5 Liveness Verification

So far, we have only discussed the verification of safety properties, for which the stack slicing algorithm was designed. In [8] we outlined a very similar dual-core algorithm that can be used for liveness verification. The method is simply to perform the first and nested part of Spin's depth first search procedure [4] in parallel on two separate cores. The performance of this algorithm is unavoidably application dependent, but it typically offers a speedup over the single-core algorithm. Optimally, of course, it could cut the verification time for large verification problems in half. As has correctly been pointed out in [2], this liveness verification algorithm does not have the desired property of scaling with the number of available processing cores beyond two. Both the multi-core safety and liveness algorithm were designed to satisfy two important design criteria:

- First, we require that the algorithm, like all other algorithms in Spin, can work on-the-fly. If the amount of available memory on our system is insufficient, we still want to be able to complete the best possible verification within the available resource limits. This eliminates any algorithm that first requires a global reachability graph to be generated in memory before an analysis phase can be initiated.
- Second, we require that the computational complexity of the verification problem is not increased by a non-linear factor. Clearly, any overhead introduced due to a switch to an algorithm with higher computational complexity will have to be regained elsewhere if we want to realize an overall performance improvement. With larger numbers of available processing cores, it may be acceptable to increase the verification cost by a small constant factor (say 2 or 3), in the knowledge that the available parallelism will be able to make up for the loss. It would, however, be a significant setback if the verification cost could increase by a non-linear factor, e.g. quadratically. In performance measurements, we should of course also always compare results with the best available single-core version of an algorithm, not with single-core runs of the algorithm with higher complexity.

Several algorithms have been studied that do incur a higher verification cost than the nested depth-first search. In [2] an implementation of a few of the more promising candidates is discussed, and detailed performance results are presented, which makes it possible to compare the performance of the current Spin multi-core LTL verification algorithm with that of these alternatives. The research group in Brno has made a significant effort to build a large database of model checking problems that can be used as benchmark problems to compare the performance of different model checking algorithms.

Table 2
Performance Comparison Liveness (runtimes in seconds)

	DiVinE				Spin 5.0	
Nr Cores Used:	1	4	8	16	1	2
elevator2.3a.prop4	98.70	66.10	35.20	26.80	19.37	19.20
leader-filters.5.prop2	26.60	13.90	9.70	7.90	0.51	0.58
peterson.4.prop4	42.50	22.10	12.30	9.20	6.82	6.82
rether.5.prop5	90.00	52.70	37.50	27.20	2.61	2.55

The collection contains 57 separately models. [3] Each model is parameterized to give between 3 and 8 different problem instances, which brings the total number of models in the database to 298. For each instance a wealth of information is provided. For most models, a translation from the native DVE format to Promela (Spin's specification language) is also provided.

A difficulty in performing unbiased comparisons between model checkers has always been that different model checkers use different specification languages. Although it is often possible to convert a specification from one format to another, such translations almost always benefit the model checker for which the original specification was written. Each model checker supports constructs that it can exploit to optimize its search process. It is very hard for a translator to produce models for each target tool that use the same optimizations. Instead, the translated model is typically inefficient. This effects holds for the models in the BEEM database, in the sense that for each model provided in Promela , it is readily possible to rewrite that model by hand, without any change to the model semantics, to achieve very significant performance improvements. If we are interested in demonstrating the capability of a model checker to solve a given verification problem, then we would have to do so to achieve a fair comparison. In this case, though, the situation is different. As long as we can show that each model checker explores roughly the same number of reachable states, we can achieve a fair comparison of the performance of the multi-core algorithms, irrespective of which verification problem is being represented. The models in a sense merely serve to define a reachable statespace, and all we need to do is to explore this same statespace.

6 Comparison

Figure 6 and Table 2 show results reported in [2] for the best reported alternative algorithm for LTL verification, reporting significantly improved results over an earlier implementation of the same algorithm (the OWCTY algo-

[3] See `http://anna.fi.muni.cz/models/` and `http://spinroot.com/spin/beem.html`

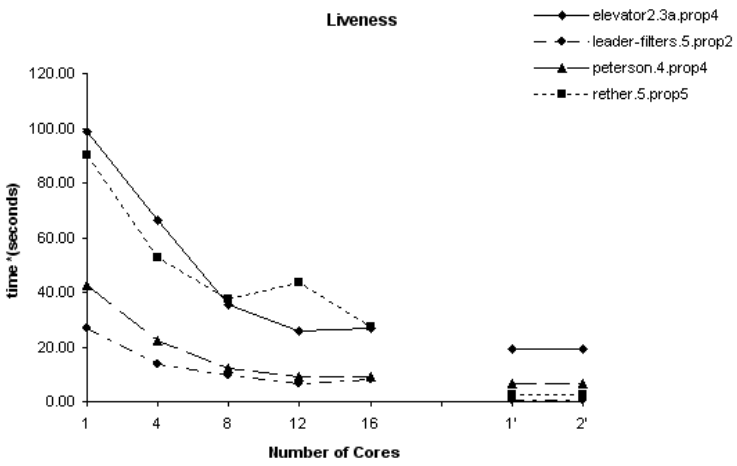

Fig. 6. Performance results for multi-core LTL verification of four verification problems from the BEEM database reported in Fig. 7 of [2] (1 to 16 cores, curves on the left-hand side), compared with performance results for the same problems with SPIN 5.0 on 1 and 2 cores (curves on the right-hand side).

rithm). The performance results for four separate models, for up to 16 processing cores, are shown in the curves on the left-hand side in Figure 6 and Table 2. On the right-hand side in Figure 6, we have plotted the performance results for the conservatively designed liveness algorithm in SPIN 5.0, proving the same properties for the same statespaces. The runtimes themselves are shown in the two right-most columns in Table 2. The measurements on the left were made on a 2.6 GHz Linux system (Red Hat 4.1.1-1), and compiled with gcc version 4.1.2, using -O3 optimization in 32-bit mode. Our measurements were made on a 2.3 GHz Linux system (Ubuntu 7.0.4, 64-bits) with 32 GB of memory and using the same version of gcc, also compiling in 32-bit mode. The results were normalized by multiplying our performance numbers with 2.3/2.6 to match the clockspeed of the computer used for the Brno results. All verifications were performed in the same way that they were done in [2], which means that we disabled statement merging (using spin -o3 to generate the verifiers), and we disabled partial order reduction (adding the compile-time directive `-DNOREDUCE` to the compilations).

In all four cases, the performance of the liveness algorithm from SPIN 5.0 using two processing cores is better than the performance reported in [2] for runs using twelve or sixteen processing cores. Curiously, the performance of SPIN running on one single core also outperforms the performance of the alternative algorithm running on sixteen cores. The largest difference is seen for the BEEM leader election model, where SPIN performs the liveness verification 15 times faster on one processing cores than the alternative algorithm on 16 cores. We believe that the explanation for this phenomenon is the increased verification complexity that is incurred by the alternative algorithms, only some of which can be made up with the use of larger numbers of cores.

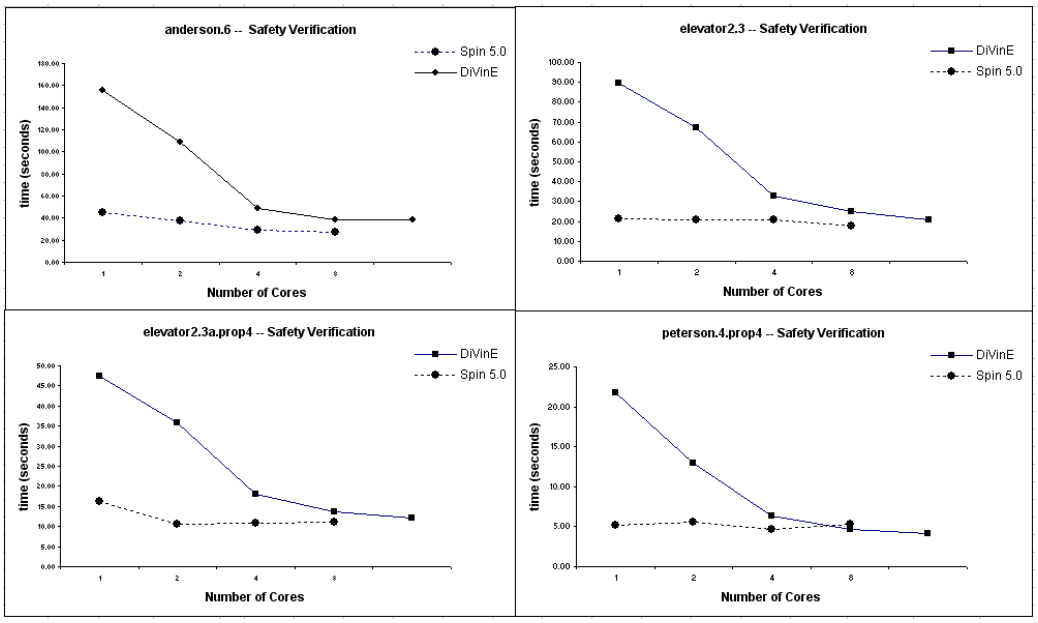

Fig. 7. Performance results for multi-core safety verification of four verification problems from the BEEM database reported in Fig. 8 of [2] (solid lines) compared with performance results for the same problems with SPIN 5.0 on 1 through 8 cores (dotted lines).

The improvements seen in the alternative algorithms are in these cases limited to roughly 12 cores, and no further improvement is seen by adding more.

The results are not exclusively positive for SPIN though. As it turns out, for these particular problem instances, the dual-core liveness verification algorithm in SPIN 5.0 does not succeed in delivering further improvements, as shown in Table 2. It would be tempting to say that the search is already optimal and cannot be improved further in these cases, but that would be far from the truth. There are several factors that can explain the effect better, as explored more fully in [8]. One potential explanation is that the state sizes for these problem instances are all relatively small (ranging from 56 to 272 bytes), where our measurements indicate that SPIN's algorithms perform best for larger state sizes (corresponding to larger verification problems). Another reason is that for relatively short runtimes, the overhead of setting up shared memory segments and work queues, and for performing termination detection, becomes more noticeable and starts to reduce overall performance; that is, some of the problem instances are too small to see a benefit of multi-core algorithms.

Also reported in [2] are results for safety verification using a different multi-core verification algorithm named MT-BFS. Figure 7 shows the MT-BFS results (solid lines) together with the results for the same models using SPIN's stack slicing algorithm (dotted lines). The system available to us for these measurements was limited to 8 cores, so we could not repeat the measurements in [2] with 12 or 16 cores. The same effects as observed in Figure 6 are

visible. In this case we cannot explain the differences in performance based on the computational complexity of the algorithms that are used (they should match). The difference could merely be that the Spin implementation is more efficient. For the models used in Figure 7, the stack slicing algorithms shows little improvement with increasing numbers of cores, which is certainly within the range of possible behaviors, but not typical. (Cf. Figure 5).

Reflecting on the graphs in Figure 7 we can also observe that many implementation inefficiencies, which hide in all verification tools, often add only a linear cost to the verification process, which can be overcome with the use of multi-core processing. We may be seeing this effect in the comparisons in Figure 7, where the performance of the alternative algorithms converges near 8 cores. Still for these models, a single-core run with Spin already seems to produce a verification process that the best currently available algorithms cannot seem to improve upon. This is of course not the result we were after. It merely means that there is much work that remains to be done in this domain of application.

7 Conclusion

It has been argued that the classic depth-first search procedure is inherently sequential and therefore cannot be parallelized [9]. The stack slicing algorithm shows that this is not necessarily the case. At least in the domain of logic model checking we have found an application where we can parallelize the depth-first search procedure and can in some cases achieve even near linear speedups in the verification of safety properties on multi-core systems. Much more work remains to be done in this domain to more fully explore the options that are available to use to improve the search process further. Not explored here, but equally important, are the impact of partial order reduction strategies and of compiler optimization techniques on search performance. More details on these aspects can be found in [8].

It is as yet an open problem how a liveness verification algorithm could be generalized to the use of more than two processing cores while retaining a low search complexity. It would be easy to conclude that no such generalization is possible, but as we have seen there often are special cases where significant improvements can be achieved. In retrospect such findings often seem obvious. Finding a simple extension of the liveness algorithm, however, will for the time being have to remain non-obvious.

Acknowledgement

The author is grateful to Dragan Bosnacki from Eindhoven University and to Rajeev Joshi and the other members of the JPL Laboratory for Reliable Software for many comments and key insights provided on the work that is presented here.

References

[1] Dijkstra, E.W., *Shmuel Safra's version of termination detection*, EWD998, 15 Jan. 1987.

[2] Barnat, J., L. Brim, and P. Rockai, *Scalable multi-core LTL model-checking*, Proc. 14th SPIN Workshop 2007, Berlin, Germany, Springer Verlag, LNCS.

[3] Geldenhuys J., *State caching reconsidered. In Susanne Graf and Laurent Mounier*, Proc. 11th SPIN Workshop 2004, Barcelona, Spain, Springer Verlag, LNCS 2989.

[4] Holzmann, G.J., D. Peled, and M. Yannakakis, *On Nested Depth-First Search*, The SPIN Verification System, American Mathematical Society, (1996), 23–32.

[5] Holzmann, G.J., "The SPIN Model Checker - Primer and Reference Manual," Addison-Wesley, 2004.

[6] Holzmann, G.J., R. Joshi, *Model-driven software verification*, Proc. 11th SPIN Workshop 2004, Barcelona, Spain, April 2004, Springer Verlag, LNCS 2989, 77–92.

[7] Holzmann, G.J., *The design of a distributed model checking algorithm for SPIN*, Conf. on Formal Methods in Computer Aided Design (FMCAD), San Jose, CA, USA, (November 2006), invited talk.

[8] Holzmann, G.J., and D. Bosnacki, *The design of a multi-core extension of the SPIN model checker*, IEEE Trans. On Software Engineering, to appear.

[9] Reif, J.H., *Depth First Search is inherently sequential*, Information Processing Letters, Vol. 20, Nr. 5, (1985), 229–234.

[10] Stern, U., and D. Dill. *Parallelizing the Murphi verifier*, Proc. 9th Int. Conf. on Computer Aided Verification, Haifa, Israel, Springer Verlag, LNCS 1254, (June 1997), 256–278.