EINDHOVEN UNIVERSITY OF TECHNOLOGY

The design of a multi-core extension of the Spin Model Checker

Citation for published version (APA):
Holzmann, G. J., & Bosnacki, D. (2007). The design of a multi-core extension of the Spin Model Checker. *IEEE Transactions on Software Engineering*, *33*(10), 659-674. https://doi.org/10.1109/TSE.2007.70724

DOI:
10.1109/TSE.2007.70724

Document status and date:
Published: 01/01/2007

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 23. Aug. 2022

The Design of a Multicore Extension of the SPIN Model Checker

Gerard J. Holzmann and Dragan Bošnački

Abstract—We describe an extension of the SPIN model checker for use on multicore shared-memory systems and report on its performance. We show how, with proper load balancing, the time requirements of a verification run can, in some cases, be reduced close to N-fold when N processing cores are used. We also analyze the types of verification problems for which multicore algorithms cannot provide relief. The extensions discussed here require only relatively small changes in the SPIN source code and are compatible with most existing verification modes such as partial order reduction, the verification of temporal logic formulas, bitstate hashing, and hash-compact compression.

Index Terms—Software/program verification, model checking, models of computation, logics and meanings of programs, distributed programming.

✦

1 Introduction

Logic model checking can be used to verify the logical correctness of distributed algorithms and asynchronous system designs, both for hardware and software. Thanks to a series of algorithmic improvements over the last few decades and helped by the increasing power of mainstream desktop CPU systems, the range of problems that can be solved with model-checking tools continues to expand. The best model checkers today can analyze models with millions of reachable system states in seconds—which is more than adequate to support the verification of abstract design models of asynchronous software systems. As a result of the progress made, model-checking tools have become a fairly standard part of safety critical systems development. The SPIN verifier [15], [20], first introduced in 1989, is a public-domain open source software tool that is specialized to the verification of correctness properties of asynchronous software systems. It is currently one of the most widely used verification tools in this domain.[1]

The effectiveness of any verification method, be it manual or mechanical, is ultimately limited by problem complexity. Nevertheless, the larger the range of problems we can analyze today, the stronger our desire to tackle still larger problems tomorrow. Alas, the effect of Moore's curve [31] to drive a continuing increase in the performance of CPUs is slowly disappearing. In mid-2002, for instance, the fastest desktop PC one could buy ran at a clock-speed of 2.5 GHz. At the time of this writing, mid-2007, the fastest PC available runs at 3.8 GHz, where a continuation of Moore's curve would have predicted machines almost twice that fast (6.6 GHz). Instead of further increasing raw CPU speed, chipmakers have changed direction to focus on the development of *multicore* systems. Dual-core and quad-core systems are already widely available, with larger numbers of processing cores soon to follow. This means that, to further increase the problem solving capabilities of logic model-checking tools in the foreseeable future, we must develop strategies that can exploit the capabilities of *multicore* CPU systems.

Where the change from a 1 GHz system with 1 Gbyte of memory to a 2 GHz system with 2 Gbytes of memory could bring an automatic doubling of the raw problem solving capability of any model checker, not requiring any change in the tool itself, the change from single-core to multicore systems does require algorithmic changes. In the past, there has been significant interest in using model checkers on large compute clusters. However, progress made in this area has had only a limited impact on the mainstream use of model checkers. Today, there are no broadly used extensions for the most commonly used model checkers for cluster computers that preserve their full range of capabilities.

Multicore systems differ in many aspects from compute clusters. We will refer to compute clusters as multi-*CPU* systems here. A multicore system can offer all CPU cores fast access to a single shared memory arena, thus avoiding the need for the relatively slow transfer of data from one CPU to another across data links in a cluster arrangement. Another change in CPU designs is significant in this context and that is the change from a default word size of 32 bits to a word size of 64 bits. The default word size determines the maximum amount of memory that can directly be addressed by a CPU. On a 16-bit system, only 65,536 addresses (64K or 2^{16}) can be referenced directly. On a 32-bit system, this range increases to 4 Gbytes ($2^{32} = 4.10^9$ bytes), which is still relatively small. On 64-bit systems, which are quickly becoming the standard, the limit on main memory sizes

1. http://spinroot.com.

- G.J. Holzmann is with the Laboratory for Reliable Software, Jet Propulsion Laboratories, California Institute of Technology, NADA/JPL MS 301-230, 4800 Oak Grove Drive, Pasadena, CA 91109. E-mail: gerard@spinroot.com.
- D. Bošnački is with the Eindhoven University of Technology, PO Box 513, Eindhoven, NL-5600 MB, The Netherlands. E-mail: dragan@win.tue.nl.

Manuscript received 5 Jan. 2007; revised 5 June 2007; accepted 19 June 2007; published online 10 July 2007.
Recommended for acceptance by R. Cleaveland.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0004-0107.
Digital Object Identifier no. 10.1109/TSE.2007.70724.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

has, for all practical purposes, disappeared. (It moves to 10^{18} bytes, a size that is unlikely to be matched by any real hardware for a long time.)

The change from a sequential execution of a model-checking algorithm to a distributed or parallel execution may seem attractive for two separate reasons: increasing *speed* and/or increasing the available amount of *memory*. The speed improvement is realized if we can perform different parts of the verification process in parallel. The memory increase can result from pooling the memory resources of different machines to create a larger address space than available on a single machine. We will discuss the relevance of each of these two potential motivating factors for the development of a distributed model-checking algorithm in Section 2. In Section 3, we discuss one of the central problems in the development of multicore algorithms: load balancing across CPUs. Section 4 discusses an extension of a partial order reduction algorithm that can help us retain the benefits of this technique in the extended model-checking algorithm. Section 5 provides detailed metrics on the performance of the new algorithm for both dual-core and multicore systems and presents an analysis of the structural model characteristics that can either enhance or degrade the performance of multicore solutions. Section 6 discusses two supporting algorithms that we used in the implementation: an algorithm to enforce mutually exclusive access to shared data structures and an algorithm to perform distributed termination detection. Both of these algorithms can be proven correct with SPIN itself, which is one of the rare examples where a verifier can be used to prove the correctness of aspects of its own extension. Section 7 presents a survey of earlier work in this area and Section 8 concludes the paper.

2 Initial Analysis

2.1 Memory

We first observe that, on a modern 64-bit CPU, the addressable memory size can no longer be considered a limiting factor. When the amount of available memory is the primary consideration, it will be more attractive (and more economical) to use a single CPU with a large amount of memory, rather than to divide the same amount of memory over multiple CPUs in a cluster arrangement, with each CPU needing its own power supply, graphics card, and network interfaces. This means that increasing available memory to store information during the model-checking process by linking computers together is no longer the main motivation for exploring distributed model-checking algorithms. There is no practical limit to the amount of memory that can be supported within a 64-bit address space. Today, systems with up to 64 Gbytes of RAM memory are already commercially available and the trend of steadily increasing memory sizes on standard desktop machines is likely to continue.

2.2 Speed

This leaves only the potential increase in *speed* as the longer term motivation for distributed computation. Can it be a performance advantage to use a physical separation between different computers or does the overhead involved in the inevitable data transfers between CPUs outweigh this potential advantage? To put it as simply as possible: Can we achieve greater performance in distributed model checking by using *two single-core* systems, each with 4 Gbytes of nonshared memory and communicating via a data link, or *one dual-core* system with 8 Gbytes of shared memory and communicating via shared memory? We consider this issue first and come to a conclusion about the type of algorithms that can be expected to have the best potential for realizing long-term performance improvements in logic model checking.

TABLE 1
Relative Performance of Operations in RAM, to Disk, and to a 100 Mbps Data Link Connecting Two 3.4 GHz PCs

Time to copy 10,000 bytes 1 million times	Time in seconds
RAM to RAM	3
RAM to disk	15
RAM to network via socket	847

Table 1 shows the time we measured for copying 10,000 bytes of data from one location in RAM to another with the standard *memcpy* routine, compared to the time needed to write the same amount of data either to a hard disk or to another machine via TCP/IP connection across a standard network link.[2]

These measurements show a significant advantage for local operations, even if those operations target a local disk, and a serious penalty for inter-CPU communications. The overhead of inter-CPU data transfer will have to be regained in parallel operations if an overall gain of using physically distributed computation is to be maintained. It should be observed that the network transfer time shown in the table is near the minimum value of 800 sec for a link speed of 100 Mbps. The time increases when smaller amounts of data are transferred (incurring more of the packet framing overhead), but can also be reduced if faster links are used.

For an initial assessment, assume that we have a fixed number of 100 CPUs available and we would like our model-checking algorithm to explore one million distinct reachable system states. Assume further that the relative time to perform the key operations in the model-checking process related to state creation and state storage is, as shown in Table 2, for multi-CPU systems with physically distributed memory and for multicore systems with shared memory. The first two rows in Table 2 roughly capture the performance of the SPIN model checker and the last two rows relate these numbers to the data in Table 1. For inter-CPU transfer times we assume proper caching of states to reduce the overhead associated with the transfer of small amounts of data.

The time to explore one million states sequentially on a single CPU then is

$$\mathbf{ST}: 1,000,000^{*}(\mathrm{T1}+\mathrm{T2}) = 1,100,000 \text{ units of time.}$$

2. The programs used to make these measurements, as well as the data from all of the measurements reported in this paper, are available online at: http://spinroot.com/spin/multicore/.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

TABLE 2
Relative Time to Generate, Check, Transmit, and Record State Descriptors

	Multi-CPU System (distributed memory)	Multi-Core System (shared memory)
T1: Time to generate a new state	1	1
T2: Time to check if a state is previously visited	10	10
T3: Time to transfer a state from one CPU to another	100	1
T4: Time to read/write a state to a disk file	5	5

The time needed by a *distributed* version of the model-checking algorithm will depend on the amount of decoupling we can achieve in the computation. We can express this as an *Independence Ratio (IR)* that measures how many states, on average, can be generated and processed *locally* on each CPU from a single state descriptor that starts a new search effort before a handoff to another CPU is required. Trivially, if $IR = 1$ (the lowest it can be), then all successor states that are generated must be handed off to another CPU immediately and will incur the cost of inter-CPU transmission. The larger the IR is, the better a distributed model-checking algorithm should perform.

In the well-known Stern-Dill algorithm [37], each new state is assigned effectively randomly to one of the active CPUs, which means that, with 100 active CPUs, the probability that a new state has to be handed off to another CPU is 99 percent, which gives $IR = 1.01$, that is, close to a worst-case value.

With perfect load balancing, each of the 100 CPUs would explore 1,000,000/100 reachable states and, on average, $1,000,000/IR$ of the states will have to be transmitted from one CPU to another at the cost of T3 (Table 2). If the search is performed in parallel on all CPUs, the total clock time can be divided by 100 to give the overall time spend in the parallel search (assuming perfect load balancing):

$$\mathbf{PT} : (1,000,000^{*}(\mathrm{T1} + \mathrm{T2})) + ((1,000,000/IR)^{*}\mathrm{T3}))/100.$$

The speedup of the parallel search compared to the sequential (single-core) search can then be expressed as ST/PT.[3] In the best case, this speedup will approach the number of CPUs used, in our example 100, but we can expect it to depend on the relative values of IR and T3.

The actual speedup that is realized can also be influenced by the relative values of T1 and T2. If the values in Table 2 are used, the speedup as a function of IR follows the lower curve in Fig. 1 (marked *optimized*). If, however, with a slower model checker, the time to generate and check states is higher and gets closer to the time it takes to transmit a state from one CPU to another, the speedup function changes. In Fig. 1, this is plotted as the top curve (marked *unoptimized*), which shows what happens if T1 is set to 50, instead of 1. That is, the top curve shows what the relative speedup will be if the time required to generate a state increases to roughly 50 percent of the time needed to transmit a state from one CPU to another. Curiously, this means that, for an unoptimized model-checking system, the effect of parallelization can be more impressive than for an optimized system. We will return to this observation later in the paper and confirm it in the practical application of the algorithms we introduce here.

Fig. 2 repeats the calculation for multi*core* instead of multi-*CPU* systems, where the time to transfer a state descriptor from one compute node to another is much smaller. The difference in speedup for unoptimized and optimized performance shrinks and the advantage of parallel computation can be predicted to become noticeable for smaller IR ratios. For an IR value of 7, for instance, the optimized multi*core* solution already shows 99.99 percent of the maximal speedup. This same ratio is only achieved for an IR value over 600 on a multi-*CPU* system (outside of the

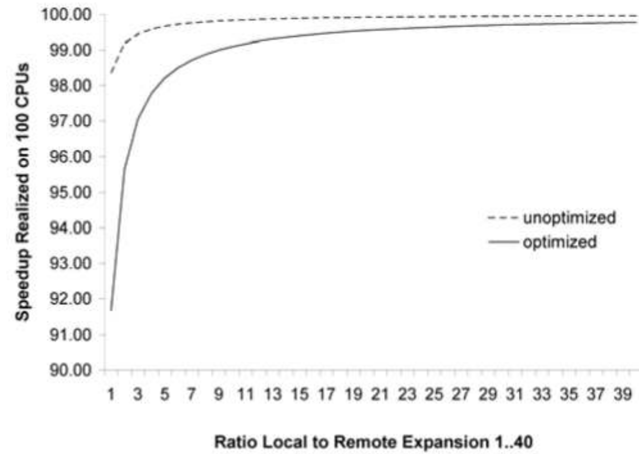

Fig. 1. Effective speedup as a function of IR on a multi-CPU system, using data from Table 2. The relative time to generate a new state is set to 1 for an optimized system and to 50 for an unoptimized system.

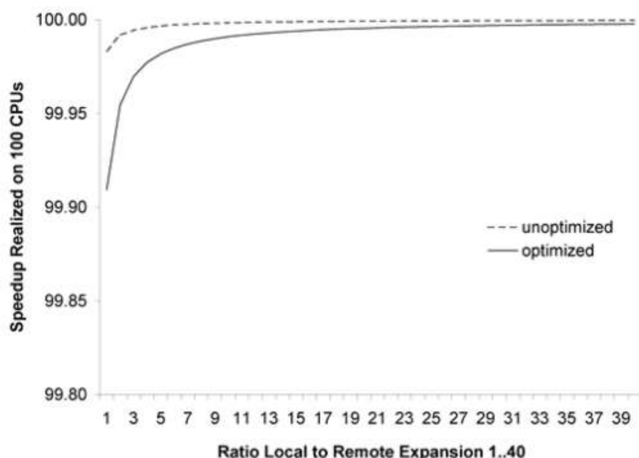

Fig. 2. Effective speedup as a function of IR on a multicore system, using data from Table 2. The relative time to generate a new state is set to 1 for an optimized system and to 50 for an unoptimized system.

3. The improvement obtained with a distributed model-checking algorithm over a sequential one can be expressed as a ratio in two different ways: as a *speedup* ST/PT or as a *reduction* of time needed PT/ST.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

graph in Fig. 1). Note that this level of decoupling requires that, for every state transferred to a CPU, on average, 600 successor states must be explored locally within the target CPU before a state is handed off again.

This initial analysis indicates that it is most attractive to target distributed model-checking algorithms to shared-memory multicore systems. We will therefore focus our attention on that class of systems.

There are several factors that are not considered in this first analysis that can also impact the performance of a multicore solution, such as the effect of partial order reduction techniques and the structure of the verification model itself. We will measure the impact of these additional factors in Section 5.2.

3 Load Balancing

A distributed model-checking task works most efficiently if we can divide the work evenly between CPUs. Since we want to maximize the independence of the CPUs in the system, little communication should be required between the CPUs. This ideal is trivially realized if we can perform separate parallel verification runs for *independent* properties, for example, specified as a set of mutually independent LTL formulas. This was the method used for achieving even load balancing of hundreds of verification runs on a 16-CPU compute cluster used in the Bell Labs FeaVer system [19]. In general, though, this level of decoupling is difficult to achieve.

One method is to define a state space partitioning function that is evaluated on-the-fly by each CPU. This partitioning function determines, for each newly generated state, which CPU should explore it further. The partitioning function should then have the property that, when a state s is generated by CPU x, most of the immediate successors of s will also be explored by CPU x. If the CPUs use a shared data structure to store all states, we can avoid having different CPUs perform redundant work by exploring the same parts of the state graph. The price to pay for this coordination is the enforcement of mutual exclusion locks on access to the (relevant part) of the state tables.

To partition the state graph into disjoint subsets, each of which is explored by a different CPU, we can make use of the notion of an irreversible transition, which we define as follows:

Definition. *An* irreversible state transition *in the global state graph is any transition with the property that its source state is not reachable from its target state.*

This means that there can be no path (a sequence of transitions) that leads from the target state of an irreversible transition back to its source state. Irreversible transitions divide the state graph into disjoint subgraphs. Note, however, that we do not require that the subgraphs be strongly connected, cf., [29].

Irreversible state transitions can be identified at compile time with a static analysis of the `proctypes` in a SPIN model. Although *any* irreversible transition will divide the set of global system states into disjoint subsets, these sets are not necessarily of similar size. The identification of irreversible transitions, therefore, is by itself not sufficient for defining a good load-balancing strategy. In the SPIN system, there is one exception, though, where we *can* identify an irreversible transition that often divides the global states space into two approximately equal and disjoint subsets. This is the transition that separates the first from the second depth-first search in SPIN's nested depth-first search algorithm [20]. Since the nested depth-first search algorithm supports the verification of *liveness* properties, this gives us an immediate candidate strategy for a *dual-core* extension of the model-checking algorithm of linear temporal logic formulas and, in general, for the larger class of ω-regular properties. This strategy does not scale to the use of more than two CPU cores, but it has the important property that it does not alter the fundamental computational complexity of the algorithm used: The complexity remains linear in the size of the number of reachable system states, with the same constant factor as applies to the standard nested depth-first search.

When a state is transferred from one CPU to another, data must be copied between the address spaces of the two CPUs. The data includes the state descriptor, some information from the current stack frame, and the data structures that SPIN uses to interpret state descriptors correctly (for example, given the offsets of process and message channel data inside the global state vector). This information can be appended to a work queue in shared memory, but, in principle (at a performance penalty), it could also be placed in a slower disk memory. Since the shared work queues are accessed in first-in-first-out (and not random) order, their contents may be read from the disk without too much overhead. Standard disk caching methods can leverage the relatively slow disk access over larger numbers of states that are read and cached simultaneously. This method was implemented as an option in the multicore implementation of SPIN, but note that, once the amount of available shared memory is sufficiently large, the option will no longer be needed. For the remainder of this paper, therefore, we will focus on the used shared memory alone.

3.1 Liveness Properties

In the dual-core model checking for liveness properties, the nested depth-first search adds each accepting state in postorder to a shared work queue [18]. This work queue is read by the second CPU, which performs only the nested part of the search to determine, for each accepting state, whether it is reachable from itself. The second CPU can perform this part of the search while recording all new states generated into a separate (nonshared) part of the state space since we already know that there can be no overlap between the states generated in the first and the second depth-first search. In this case, no locking is needed on access to the state tables. The basic complexity of the search remains unchanged compared to a single-core algorithm. Thus, for *dual*-core systems, the speedup for the verification of liveness properties in the best case can be close to twofold (cf., Section 5.1). As noted, an extension of this algorithm for systems with more than two CPU cores is likely to be nontrivial and is not explored here, cf., [36].

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

3.2 Safety Properties

For safety properties, the state partitioning function should achieve two separate objectives: a roughly equal distribution of work on all CPUs and an IR greater than approximately 10 (cf. Fig. 2). In this paper, we explore the performance of a relatively simple metric for load balancing that can achieve these two objectives in many cases of practical interest. The metric is based on a relatively simple stack-slicing algorithm. To implement this algorithm, the CPUs in a multicore system are connected in a logical ring. Each pair of neighboring CPUs shares access to one single work queue, which is maintained in shared memory. CPUs can hand off states to each other only by sending them clockwise around the ring. This means that access to the queues can be implemented with a lock-free data structure since each queue has one unique writer and one unique reader.

Let d be the depth in the state graph at which a state is generated. Each CPU core in the ring can decide to hand off a newly generated successor state to its right neighbor when the search depth d exceeds a preset bound L within its local stack. When a state is transferred, the target CPU will start to explore that state as a local root for its search, that is, with an empty local stack and with the search depth d starting at zero. This means that, with this metric, at every $d\%\mathrm{L}$ steps from the original root of the global state graph, a state sequence can be transferred to a neighboring CPU. The $N\mathrm{th}$ CPU in an N-core system hands off states back to the first CPU, using modulo-N counting. To achieve sufficient independence, the value of d should be larger than 10 and less than D/N, where D is the maximal depth of the depth-first search tree. D is typically on the order of 10^4 to 10^6 steps for the larger applications that are of primary interest for multicore verification. This strategy should be able to achieve reasonably good load balancing for values of N (the number of CPU cores) up to a 10^2 to 10^3. We will report on the actual performance of this metric for the verification of safety properties in Section 5.

Although the stacks are necessarily local (that is, nonshared) in this search mode, we have a choice of placing the state tables in either shared or nonshared memory. Placing the states in the shared memory brings the need for locking on access to the relevant part of the table when new states are inserted, but it eliminates the chance that duplicate work is done. The overhead of locking can be reduced by using fine-grained locking techniques (that is, locking only that part of the hash table that contains the newly generated state), so this will, in almost all cases, be the preferred method. Our implementation uses this method as the default.

One of the goals for the design of the extension of SPIN is to make only minimal changes in the existing code and to preserve as much of the existing capabilities of the system as possible (and the trust we can place in these). This level of minimal intrusion can be achieved by carefully selecting the points in the search where a state handoff can take place. Fig. 3 illustrates the standard nested depth-first search algorithm that is used in SPIN [20, p. 180] and indicates two points in the search, where, in our current implementation, a state can be handed off to another CPU with minimal change to the existing algorithm for logic model checking. These points are the natural recursion points in the nested depth-first search.

```
Stack D = {}
Statespace V = {}
State seed = nil
Boolean toggle = false

Start()
{
   Add_Statespace(V, A.s0, false)
   Push_Stack(D, A.s0, false)
   Search()
}

Search()
{
   (s, toggle) = Top_Stack(D)
   for each (s,l,s') in A.T
   { if (toggle == true
     && (s' == seed || On_Stack(D, s', false)))
     { PrintStack(D)    # accept cycle found
       PopStack(D)
       return           # end nested search
     }

     if (In_Statespace(V, s', toggle) == false)
     { Add_Statespace(V, s', toggle)  # new state
       Push_Stack(D, s', toggle)
[S]    Search()         # dfs recursion
   } }

   if (s in A.F && !toggle)     # in post order
   { seed = s                   # accept state
     Push_Stack(D, s, true)
[L]  Search()                   # nested dfs
     Pop_Stack(D)
     seed = nil
   }

   Pop_Stack(D)
}
```

Fig. 3. Handoff points for the dual-core nested depth-first search.

The first point ([L] in Fig. 3) corresponds to the start of the nested part of the search, which is the handoff point for the verification of liveness properties. The second point (marked [S]) is the handoff point for the verification of safety properties, based on the depth metric we have discussed above. The two points interfere only minimally with the existing algorithm and preserve all other SPIN options. The two metrics are never mixed since SPIN can only do one type of search at a time (either for the verification of safety properties or for the verification of liveness properties). For liveness properties, only handoff point [L] is used; for safety verification, only handoff point [S] is used.

The handoff depth that is used in the verification for safety properties defaults to a value of 20, based on performance measurements with this algorithm (Section 5). The default can be changed by the user with a command line argument.

4 Partial Order Reduction

The standard implementation of SPIN can achieve considerable speedup from the use of the partial order reduction method that was introduced in [17] and revised in [18]. This algorithm reduces the number of successor states that must be generated at each step during the search if it can be guaranteed that any deferred transition will eventually be explored from a later state. The partial order method ensures that, when a transition is deferred for later

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

execution, its continued executability is guaranteed. A key provision in the algorithm is the prevention of infinite deferral of transitions along cyclic paths in the state graph. This cyclic deferral is prevented in the standard SPIN algorithm by making sure that none of the successor states from a reduced set of transitions can appear on the depth-first search stack, above the state being explored. If any successor state appears on the stack, it could otherwise close an infinite deferral cycle and lead to incompleteness of the search process. In the partial order theory, this is known as "the ignoring problem."

In general, for the verification of *liveness* properties, if *at least one* successor state appears *on* the depth-first search stack, *no* reduction is performed from that state [17], [32]. This precondition on the application of partial order reduction is known as the *cycle proviso* (or, in the case of the depth-first search, sometimes also the *stack proviso*).

Two other versions of the cycle proviso are used in SPIN for the verification of *safety* properties with either a depth-first or a breadth-first search (BFS).[4] Clearly, in the case of a BFS, there is no depth-first stack and, thus, an alternative method must be adopted to prevent the ignoring problem.

- Depth first. *At least one* successor state appears outside the stack [16].
- Breadth first. *At least one* successor state is in the BFS search queue [3], [5].

The latter condition is more conservative and independent of stack contents, but, as a result, it achieves smaller reductions of the state space size.

In a multicore search algorithm, like in a BFS, the full depth-first search stack starting from the original root of the state graph is not always available. This means that we must use a different method for solving the ignoring problem. One alternative method that is independent of the stack is to force the full expansion of successor states whenever at least one of the asynchronous processes in the verification model traverses a transition that corresponds to a backward edge in its *local* control structure (a precondition for a *global* cycle being created), cf., [28]. Such transitions can easily be identified statically. An implementation of this method, though, is significantly outperformed by an alternative method that we will describe next.

The method for solving the ignoring problem that we use is to force a full exploration of successor states in two additional places during the search (that is, in addition to the case where successor states are found on the local stack of the executing CPU).

- The *first* additional expansion is made for so-called "border states" (handoff states), that is, states whose successors fall below the handoff depth of the current CPU and, therefore, might have appeared on the search stack. The most conservative approach is to treat them as if they had appeared on the stack.
- The *second* case is for successor states that are previously visited *by another* CPU. In a single-core execution, these states may have appeared on the search stack, but this is no longer verifiable by the executing CPU since it has no access to the full search stack anymore. Again, the most conservative approach is to treat these states as if they appeared on the stack.

4. No efficient algorithm is known for the verification of liveness properties with a BFS algorithm [20], so this combination is currently not supported in SPIN.

The second case can be further optimized by restricting it to cases where the previously visited state was generated by a CPU with a *higher* pid number than the executing CPU [7]. This additional restriction makes sure that the full expansion can occur in only one CPU, not in both.

Proof (Sketch).[5] The proof is by induction. Consider a case where CPU-i generates a successor state s that was previously generated by CPU-0, with *zero* being the lowest pid in the system and $i > 0$. When CPU-0 generated the state, the state was new and, hence, CPU-0 must explore all states reachable from s, guaranteeing that any deferred transitions are eventually explored as well. Since CPU-0 has the lowest pid in the system, it cannot defer any expansion for previously visited states by any other CPU. CPU-0 provides the induction hypothesis, which hinges on the fact that every CPU can trust that the successors of previously visited states generated by CPUs with lower pid numbers are fully explored by at least one of those CPUs. □

Because of the required full expansion of all border states, this version of the partial order reduction method can be expected to disfavor short handoff intervals. As the data presented in Section 5 confirms though, this initial effect quickly disappears.

5 Measurements

5.1 Basic Performance

Table 3 shows a comparison of the runtime requirements of verification runs with the extension of SPIN using one or two CPU cores for the verification for six verification models, most of which are taken from the SPIN distribution: a leader election algorithm for a network of eight processes, Peterson's generalized mutual exclusion algorithm for four processes, a sliding window protocol for a window size of five messages, the dining philosophers problem with nine processes, a model of a phone switch, and a reference model that we will discuss in more detail shortly. For these measurements,[6] we performed verifications for both safety and liveness properties without partial order reduction to ensure that both the dual-core and single-core runs explore precisely the same total number of reachable states. The safety verifications achieve speedups that vary from the near optimal factor of 1.98 for the reference model to 1.36 for Peterson's algorithm.

To assess the performance of liveness verification, we placed an accept label at a local cycle in each model and checked for the presence of global acceptance cycles in the

5. A formal proof is presented in Appendix A.

6. All measurements were made on a system with two quad-core CPUs for a total of eight CPU cores. The CPUs ran at 2.3 GHz, with 32 Gbytes of shared memory, under a 64-bit operating system (Ubuntu version 7.0.4). All compilations were done with gcc version 4.1.2. All data and models are available online at http://spinroot.com/spin/multicore.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

TABLE 3
Measurements for Small Verification Models with Default Compilation

Model	Runtime (seconds)					
	Safety			Liveness		
	Single Core	Dual Core	S:D	Single Core	Dual Core	S:D
Leader (8) 4.9M states	133.0	**82.8**	*1.61*	303.0	**193.0**	*1.57*
Peterson (4) 10.57M states	34.2	**25.1**	*1.36*	72.8	**48.6**	*1.50*
Sliding Window (5) 11.87M states	37.5	**24.8**	*1.51*	74.9	**58.5**	*1.28*
Philosophers (9) 1.64M states	26.2	**17.0**	*1.54*	44.4	**36.2**	*1.23*
Phone Switch 32.89M states	179.0	**110.0**	*1.63*	365.0	**199.0**	*1.83*
Reference B=8, S=200, t=13, 500K states	375.0	**189.0**	*1.98*	756.0	**453.0**	*1.67*

TABLE 4
Measurements for Small Verification Models with Compiler Optimization (-O2)

Model	Runtime (seconds)					
	Safety			Liveness		
	Single Core	Dual Core	S:D	Single Core	Dual Core	S:D
Leader (8) 4.9M states	74.6	**51.3**	*1.45*	167.0	**119.0**	*1.40*
Peterson (4) 10.57M states	18.5	**18.2**	*1.02*	38.5	**29.4**	*1.31*
Sliding Window (5) 11.87M states	19.8	**13.0**	*1.52*	39.5	**36.5**	*1.08*
Philosophers (9), 1.64M states	14.0	**10.0**	*1.40*	23.4	**21.9**	*1.07*
Phone Switch 32.89M states	90.1	**61.0**	*1.48*	189.0	**114.0**	*1.66*
Reference B=8, S=200, t=13, 500K states	124.0	**62.4**	*1.99*	251.0	**144.0**	*1.74*

state space with the nested depth-first search algorithm. Performing the measurement in this way again preserves the number of reachable states in all runs and allows us to compare the performance of safety and liveness, both for single and dual-core runs. Note that performing the same test by adding a separate LTL property would increase the number of reachable state by the addition of the never claim.

The speedup for liveness verifications in these tests ranged from a high 1.83 to a low 1.23. We will study the possible reasons for the lower values in more detail later.

5.1.1 Influence of Compiler Optimization

We have also compared the performance on the same examples when compiler optimization is used. The results are shown in Table 4. In all cases, when compiler optimization is used (compiled with gcc option -O2), the benefits of the multicore runs decrease, which is consistent with our predictions in Figs. 1 and 2. It is also noteworthy that merely enabling compiler optimization can achieve speedups greater than unoptimized dual-core verification.

5.1.2 Influence of Handoff Depth

We measured the influence of the handoff depth heuristic on the performance of dual-core verifications. A representative result of these measurements is shown in Fig. 4, in this case for the verification of safety properties for the model of a phone switch. The maximum depth of the search tree is 292,086 steps in this case and the state vector size is 100 bytes. The graphs (the top curve measured with default compilation and the bottom curve with compiler optimization enabled) have a characteristic "bath-tub" shape when the handoff depths are varied, with a mostly flat bottom, where any handoff depth selected gives comparable performance. The performance for the smallest and largest handoff depth values is often poor since the IRs decrease for these values. As before, in all of these measurements, partial order reduction was disabled to ensure that all verification runs we compare here explore precisely the same numbers of states.

The right-hand sides of the curves in Fig. 4 show performance degradation caused by approaching and exceeding the maximum search depth of the state space. Handoffs near and beyond this limit will trivially not be able to achieve proper load balancing. If too few states are transferred from one CPU to the other, the first CPU will be forced to do most of the work and the performance ultimately degrades to that of a single-core run and worse if the overhead of the dual-core infrastructure becomes noticeable. The effects are the same with or without compiler optimization enabled, although the amount of improvement when dual-core verification is used decreases, as also predicted in our initial analysis.

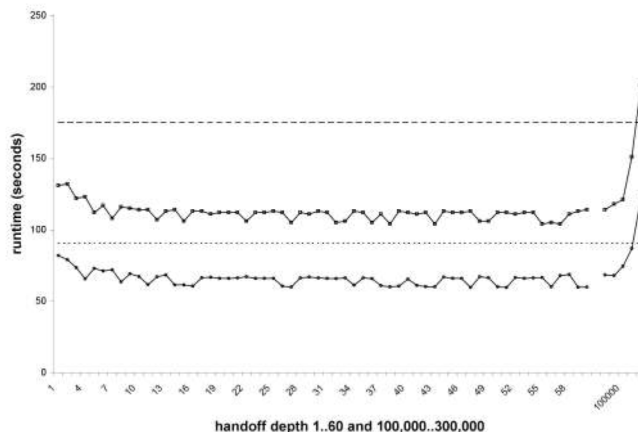

Fig. 4. Measurement of the influence of the handoff depth in the dual core verification of safety properties with (bottom) and without (top) *compile time optimization* enabled. The horizontal lines give the performance of single core runs with (bottom) and without (top) compile time optimization. All runs are without partial order reduction.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

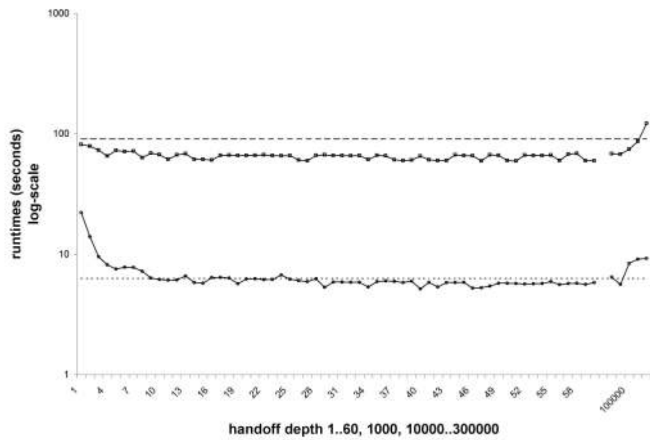

Fig. 5. Dual-core and single-core with (bottom) and without (top) *partial order reduction* enabled for varying handoff depths. All runs were compiled *with* compiler optimization. The performance of the single-core verification runs is indicated by the horizontal lines. Without partial order reduction, there are 32.8 million reachable states; with partial order reduction enabled, the number is 4.7 million states.

5.1.3 Influence of Partial Order Reduction

Fig. 5 gives a representative performance result for the effect of enabling partial order reduction in a multicore verification run for the same verification model as used in Fig. 4, this time with compiler optimization enabled for all runs. With partial order reduction, the maximum depth of the search tree reduces from 292,086 to 73,789 steps and the number of states explored reduces from 32.8 million to 4.7 million. As in Fig. 4, we varied the handoff depth from 1 to 60, in increments of one, with additional samples taken at 1 K, 10 K, 100 K, 200 K, and 300 K steps. For reference, the two horizontal lines show the relative performance of single core runs with partial order reduction enabled (dotted, bottom) and without it (dashed, top).

Clearly, the addition of partial order reduction reduces the relative benefit of multicore verification further. In some cases, when partial order reduction is enabled, the performance does not improve, improves only marginally, or even degrades, especially for the smaller handoff depth values. We can hypothesize that this is in part caused by the changes that the partial order reduction algorithm makes in the state graph structure, effectively reducing the average number of successor nodes of a state. This effect is not visible in all applications, but is clearly important. We therefore investigate this phenomenon more fully in the next section. The potential effect of state graph structure on the behavior of model-checking algorithms was also explored in [35].

5.2 A Reference Model

To verify how multicore verification depends on various structural model characteristics, we construct a reference model that allows us to vary each of a small number of relevant model parameters over a range of possible values that may be encountered in practice. The reference model we use for this analysis is shown in Appendix B.

The reference model has three independently modifiable structural parameters, allowing us to vary the number of successor states per reachable system state (the out-degree or *branch factor* of each state in the state graph), the size of a state in bytes, and the time it takes to generate a successor state, which is captured as the time it takes to execute a state transition in the model. We use an embedded C code to control this last parameter by the number of times we force the code to execute a dummy computation in an idle loop. The model generates a fixed number of 0.5 million reachable system states in each case, independent of the remaining parameters settings—so that we can more easily compare the relative runtimes across all runs.

The first measurement we perform is meant to establish how the performance of a dual-core verification run depends on transition delay. Fig. 6 shows, in the top-left graph, how the ratio of the dual-core runtime versus the single-core runtime for the reference model, for a small state size (10 bytes) and transition delays that are varied from 2^1 time units to 2^{18} time units, increasing by powers of 2. Three curves are plotted, the top curve (dotted) is for a branch factor of 1 (that is, a purely deterministic model where every state reached has at most one successor), the middle curve (dashed) corresponds to a branch factor of 2, and the bottom curve (solid) corresponds to a branch factor of 8.

These measurements show that, for models with small transition delays and/or small branch factors, a dual-core run may take up to 2 1/2 times as long as a single-core run. For models with an average out-degree of 8, though, the dual-core runs are never slower than the single-core runs, not even for the minimal transition delays we measured. The same effect is observed for larger transition delays (corresponding to more costly state transitions) and branch factors above 1. Partial order reduction tends to reduce the out-degree of states in the global system graph, so the observed slowdown of dual-core verification runs can now be understood.

The graph on the upper right in Fig. 6 repeats the same measurements for a larger state size of 200 bytes. We see fundamentally the same effects, but, for branch-factors above unity, the performance degradation effect disappears and optimal performance is quickly reached.

The graph on the lower left side in Fig. 6 separately shows how performance varies with state size for a fixed transition delay of 2^3 time units and the graph on the lower right repeats this experiment for a larger transition delay of 2^{13} time units. Note that, in the latter case, performance becomes almost completely independent of state size, likely because it is now dominated by the transition delay itself.

A few observations can be made about these results. First, the performance of multicore algorithms should be expected to be smaller when partial order reduction is used than when it is not used, although the effect can be mitigated by a number of other factors. The advantage of partial order reduction, though, is important and it is not a real option to disable it in multicore verifications. There are cases, though, where, for unrelated reasons, partial order reduction is not an available verification option, for instance, when an LTL property is not stutter invariant [20]. In those cases, multicore algorithms can prove especially valuable to reduce the complexity of verification. In the remaining cases, the use of multicore methods will provide benefits in all but the relatively rare cases where the verification model defines a near deterministic sequential

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

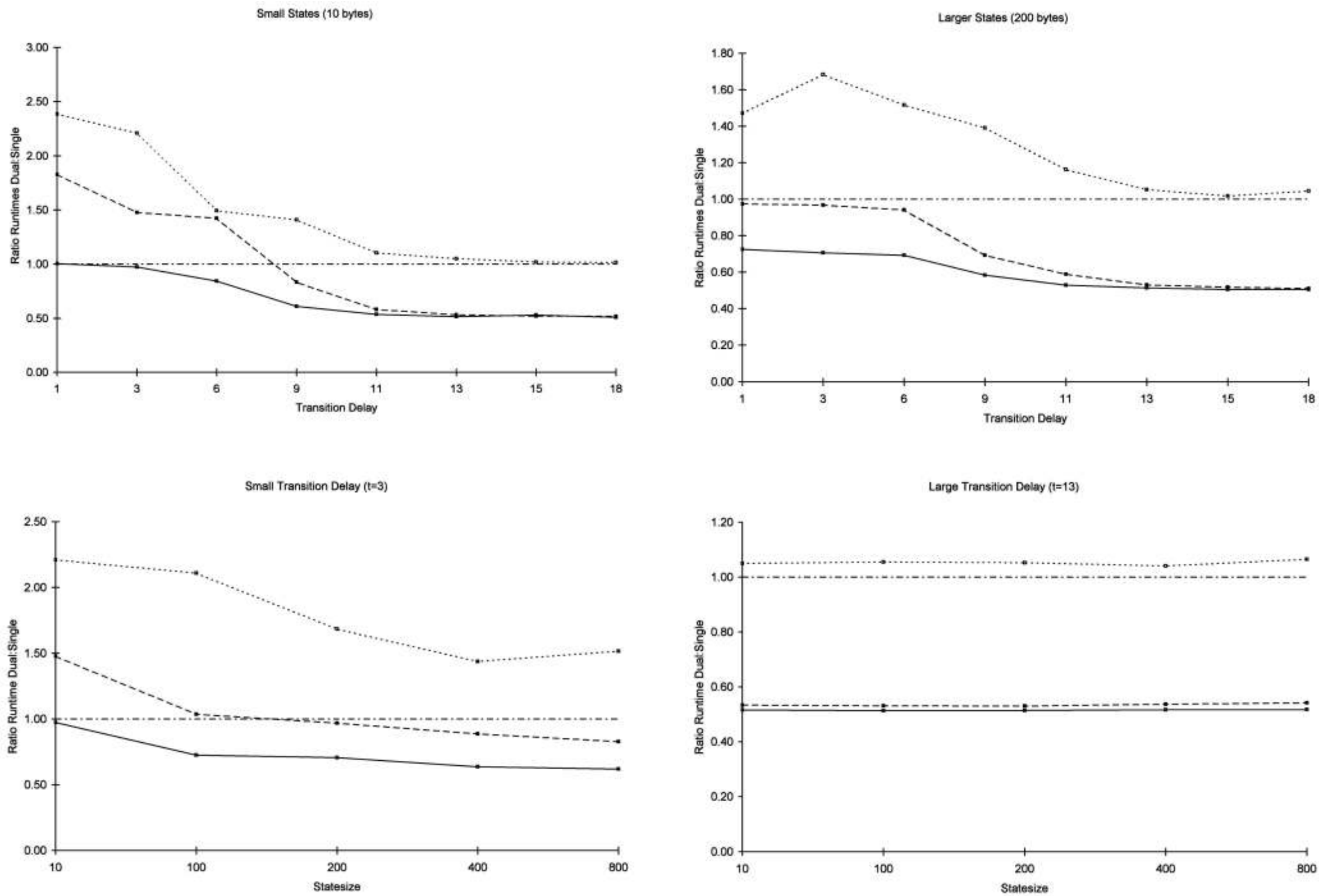

Fig. 6. Measurements on a reference model to study the effect of structural model properties on the performance of multicore verification algorithms. Varied are the out-degree of states (the branch factor), the state size, and the transition delay. The dotted, dashed, and solid lines correspond to branch factors of 1 (that is, a deterministic), 2, and 8, respectively. The horizontal dash-dot line at 1.0 in each graph indicates the value for which the dual-core runtime equals the single-core runtime (that is, there is no speedup or slowdown). The handoff depth for all measurements was 1,000, giving a perfect load balance for this model. The graphs plot the relative runtimes as the ratio of a dual-core runtime divided by the single-core runtime, that is, numbers below 1.0 represent effective speedup and numbers above 1.0 represent performance degradation.

computation after partial order reduction is applied. Note that, in a purely deterministic setting (that is, a branch factor near one), no multicore state partitioning method is likely to be effective. In these cases, one CPU will always be waiting for the other CPUs to complete their work before it can resume its part of the computation. This effect is clearly visible in the graphs in Fig. 6.

Another observation is that the multicore algorithm for safety properties can be expected to perform best for models with large state sizes and/or large transition delays. This matches an important application domain of verification models with large amounts of embedded software and the use of model-driven verification techniques, where the model checker must control and track potentially large amounts of implementation level data [21]. The importance of this type of extension should therefore be expected to increase over time as we start tackling larger and larger problem sizes with logic model checkers and as larger numbers of CPU cores become available to perform the verifications. Alas, it also means that multicore systems cannot easily show their full potential on small classroom size examples, so some tutorial value of this new class of algorithms may be lost.

5.3 Small Verification Models

The results of the performance for safety and liveness verifications for six relatively small verification models were shown in Table 3, where we used default compilations to generate the verifiers, and in Table 4, when we added an optimization phase to all compilations. Partial order reduction was disabled in all of these runs to make sure that both single-core and dual-core runs explore the same number of reachable states and give directly comparable results. For meaningful comparisons, all searches are run to completion and not stopped at errors. (Partial order reduction would also reduce the runtimes for small examples too much to still allow for meaningful measurements with the multicore algorithms.) The best performance in both tables is indicated in bold. Also indicated is the ratio Single/Dual (earlier referred to as the "speedup" factor).

For all verifications, we see effective speedups with dual-core verifications, both with and without compiler optimization. The amount of performance improvement (or degradation for that matter) that is observed in practice can depend on the specifics of a property being checked. If, for instance, we perform a check for the absence of nonprogress cycles without defining any progress labels (creating a near worst-case scenario for dual-core verification where all global

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

TABLE 5
Large Verification Models

Model	Size of one State (number of bytes)	Size of Model (lines of text)	Lines of C Code Linked to Model (lines of code)	Approx. Depth of Search Tree (Thousands of execution steps)	Approx. Number of Reachable States Explored (Millions)
DS1	3,426	487	10,781	179 K	216 M
DEOS	576	6,635	0	24,763 K	22 M
EO1	2,736	3,949	0	0.3 K	10 M
Gurdag	964	1,646	0	597 K	601 M
NVDS	180	1,229	6,068	1.5 K	247 M
CP	344	3,753	0	30,248 K	307 M

TABLE 6
Single Core Processing Rates

Model	Number of States Explored per Second in a Single-Core Run	
	Safety	Liveness
DS1	16,877	Out of memory
DEOS	357,067	185,669
EO1	14,336	11,071
Gurdag	280,833	88,923
NVDS	437,535	305,394
CP	158,116	127,906

states will trigger the start of a nested search and all global cycles constitute nonprogress cycles), performance can be expected to be poor.

5.4 Larger Verification Models

Next, we consider the result of applying the new algorithms to the safety verification of larger verification models, where, based on the experiments with the reference model, we should be able to achieve the best results. Here, we want to verify the applications in the most realistic setting: matching the way one performs large verifications in practice. This means that, in all of the runs reported in this section, we *enable* partial order reduction and we compile the verifiers *with* optimization. As we have seen, these two measures reduce the benefit of multicore verification, but, for larger applications, they should be considered essential.

We consider six large applications. The first (DS1) is a verification of a large model with embedded C code taken from NASA's Deep Space 1 mission, as described in [13]. The second application (DEOS) is a verification model of the DEOS Operating System kernel developed at Honeywell Laboratories, a variant of which is also discussed in [33]. The third application is a verification model of the autonomous planning software used on NASA's EO1 mission [10]. The next application is a verification model of an ad hoc network structure with five nodes, created by a SPIN user. The fifth application is a verification model of flash file system software developed at the Jet Propulsion Laboratory (JPL), using a large amount of embedded C code [26]. The last verification model is a call processing application, mechanically extracted from C source code with a model-extraction technique [19]. Some relevant parameters on the size of each model are shown in Table 5.

Even with partial order reduction, the applications listed here generate state spaces that are, in most cases, too large to explore exhaustively. Only strong compression techniques allow us to perform approximate verifications in these cases. In practice, such verifications normally suffice to identify correctness violations if they exist (they usually do). The numbers of reachable states listed in Table 5 are explored in verification runs with hash-compact compression for the DEOS and Gurdag models and with bitstate hashing for the remaining models. To be able to complete large numbers of tests, we sized the hash table for the bitstate runs such that the single-core verifications could be completed in under 30 minutes of runtime for all models. As one example, the verification for the EO1 model explores 10,059,569 reachable states in 699 sec on one CPU core and 10,056,715 reachable states in 89 sec on eight CPU cores, resulting in a speedup of 7.85. The bitstate verifications can also be done with larger hash arrays and can explore larger numbers of reachable states, but the speedup factors when moving from single-core to multicore remain largely unaffected in such tests. For the EO1 model, for instance, we can set up the bitstate run to explore 2.55 10^9 states in about eight hours, using eight CPU cores, or in 2.6 days on a single core.

Note that, with the use of partial order reduction, the number of states reached during a verification run can vary and will not necessarily be the same when the number of CPU cores is changed. The number will generally also depend on the precise interleaving of the different threads of computation that are now used in the model checker, which means that the number of reachable states reported can also vary from one run to the next. To compare the performance of the multicore algorithms for the different runs, we therefore use a metric that is independent of the size of the state space that is explored and that shows how performance scales with additional CPU cores. The performance metric we use is the number of states that are explored (and stored) per second of runtime, rather than the runtime itself (which depends on the total number of states explored). We should expect the processing rate to increase with the number of CPU cores used. To compare how performance scales with increasing numbers of cores, we use the ratio M:S, where S is the processing rate for a single core verification run (measured in states per second) and M is the processing rate for a multicore run. For N cores, the optimal M:S ratio is N.

Safety. The processing rate depends on the size of the state descriptors, as can easily be seen by correlating the data shown in Table 6 with that in Table 5. Fig. 7 shows how the performance scales with increasing the number of CPU cores for the verification of safety properties for these six large models. For each model, the processing rates differs (being related to the numbers given in Table 6), but the scaling behavior can still be compared. Performance for safety verification can be seen to increase with increasing numbers of CPU cores for each model, but in varying degrees. For the two applications with the largest state descriptor sizes (DS1 and EO1), the scaling is close to optimal: achieving a near N-fold speedup on N cores (7.3 for DS1 and 7.8 for EO1 on eight cores). For three

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

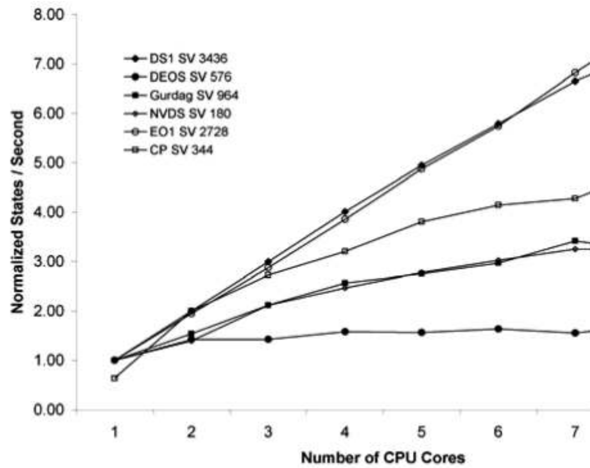

Fig. 7. Scaling with increasing numbers of CPU cores for six large verification models on an 8-core machine (verifying safety properties). In two cases, near optimal scaling is seen.

applications with medium size state descriptors (CP, Gurdag, and NVDS), the speedup is between 5 and 3 on eight cores and, for the remaining application (DEOS), a speedup of 1.7 is measured when eight cores are used. Perhaps not by coincidence, the applications that turn out to scale best are the ones with the largest state descriptors, which is consistent with the predictions we made based on the reference model. For the CP application, the speedup compared to a single core run is 7.7, but the single core run appears to be suboptimal in this case—possibly due to the extremely deep search tree of over 30 million steps. This results in an apparent superlinear speedup when compared directly to runs with larger numbers of CPU cores (where, due to the sliced stack algorithm, the search stack remains small). For this application, we therefore use the dual-core performance as the point of reference for the scaling chart in Fig. 7—reducing the calculated speedup on eight CPU cores from 7.7 to 4.9.

Liveness. Performance results for the verification of liveness properties are shown in Table 7, where we used the same method as before: We checked for the existence of acceptance cycles after adding one or more accept-state labels to each model. Liveness properties could not be verified for the DS1 model within the available amount of memory. All results are again obtained with partial order reduction enabled and with compiler optimization. Visible in the table is that the number of visited states can increase slightly from a single-core to a dual-core run, most likely due to the somewhat weaker partial order reduction rules. Despite searching slightly more states, in four out of five cases, we measured a speedup for the dual-core runs with the liveness algorithm ranging from 1.5 to 1.3. In one case, the call processing application, we measured an increase in runtime. We will return to this effect briefly in Section 8.

In Section 6, we will briefly review some supporting algorithms that were used in the implementation of the dual-core algorithms in SPIN, which, in some cases, has led to surprising conclusions about the usability of some well-known algorithms for mutual exclusion, and favorite examples of distributed model-checking capabilities.

TABLE 7
Liveness Checks for Large Models

Model	States Visited (Millions)		Runtime (seconds)		
	Single Core	Dual Core	Single Core	Dual Core	S:D
DEOS	44.6	44.7	357	266	1.34
Gurdag	178.5	182.8	1930	1470	1.31
NVDS	289.5	292.8	862	571	1.51
EO1	20.1	20.1	2200	1470	1.50
CP	308.7	323.9	1960	2640	0.74

6 Supporting Algorithms

6.1 Mutual Exclusion

The SPIN source code is designed to be platform independent, with the benefit that the standard distribution of the tool can be compiled without changes on any system that supports an ISO/ANSI compatible C compiler. The extension to multicore systems requires a provision for mutually exclusive access to shared data structures. In some cases, for instance, when there is just a single writer and a single reader of a shared data queue, locking can be avoided by adopting standard lock-free data access methods, but this is not always the case. Access to the linked lists in the hash table in SPIN, for instance, must be done under mutual exclusion locks, minimally on that part of the data structure that is accessed.

The simplest method to enforce mutual exclusion locks is to take advantage of machine-specific atomic test-and-set or compare-and-swap instructions. However, these instructions are not platform independent. An alternative is to use one of the well-known algorithms for enforcing mutual exclusion that require only the indivisibility of individual read and write operations. One of the simplest algorithms of this type is Peterson's algorithm, which was first described in [34]. The algorithm can be modeled in SPIN and proven correct (the example is part of the standard SPIN distribution). There are many independent correctness proofs for this and similar types of algorithms and they are used as standard examples of model-checking techniques. An implementation of this algorithm in C is also readily built. The shared variables *flag*[0], *flag*[1], and *turn* must be declared "volatile" to make sure that the compiler knows that their value can be changed by processes other than the executing process, suppressing certain types of compiler optimization. Curiously, though, when compiled for a modern CPU, like the Intel Pentium D or Xeon chips, the platform independent implementation of Peterson's algorithm turns out to allow violations of the mutual exclusion property with low probability.

The culprit in this case is the use of out-of-order execution optimizations by the CPU, which is used to optimize memory access sequences. This is a known problem that is also reported in the Wikipedia entry for Peterson's algorithm.[7] The solution is to include so-called

7. http://en.wikipedia.org/wiki/Peterson's_algorithm.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

memory barriers in the code that force memory loads and stores to happen at known points in the code. The irony is that these memory barrier instructions are again machine dependent, which turns Peterson's platform independent algorithm back into a machine dependent one—losing the advantage over the much simpler machine dependent test-and-set alternative. The SPIN extension, therefore, reluctantly uses the faster test-and-set instruction—with different variants compiled in for some commonly used types of CPUs (such as Intel, AMD, Sparc64, PowerPC, and so forth).

An additional observation on this issue may be of interest. The potential loss, with low probability, of mutual exclusion on access to the hash table is actually less harmful than it would appear at first sight. The code can be written in such a way that the effect of a collision on access to the linked lists in the hash table can result in either the loss of states from the hash table or the duplicate entry of a small number of states into the hash table. Interestingly, neither effect can affect the logical soundness or completeness of a verification run. Duplicate states can cause higher memory consumption than necessary and lost states can cause redundant work to be done (if the same states are revisited later in the search), but neither can cause an error. Errors could be caused if corruption of states in the hash table would be possible, but this is easily prevented in a dual-core implementation. Similar changes in the contents of depth-first *stack* information would be fatal, but, since the stack information is always maintained in nonshared memory, no such effect can occur. This means that, for platforms without test-and-set support, we could, in principle, fall back on Peterson's algorithm, where we would have to accept possible inaccuracies in the reporting of state counts at the end of a verification run yet remain secure in the accuracy of the verification itself.

6.2 Distributed Termination Detection

When a CPU's work queue is empty, the computation is not necessarily complete. It could be that another CPU will generate more states, some of which will eventually be handed off to the now idle CPU. We need a reliable distributed termination detection algorithm to conclude a distributed run of the model checker. Fortunately, this problem is well understood and several solutions are available. Our version of the algorithm is based on Dijkstra's presentation of Safra's solution [11]. The verification model for this algorithm (as it has been implemented in the extended version of SPIN) is available as part of the standard SPIN distribution.

7 Overview of Earlier Work

Most work in distributed model checking to date has been focused on algorithms for cluster computers, instead of shared memory architectures. Much of this earlier work was also focused on the verification of *safety* instead of *liveness* properties. The first contribution of this type was the Stern-Dill algorithm [37]. This algorithm uses a hash function as the criterion for assigning states to nodes in a compute cluster. With a sufficiently good hash-function, this method should achieve near-optimal load balancing, but it suffers from the overhead of frequent state transfers. As noted earlier, with N CPUs, the probability that a state generated on a given node can be further explored on that node is only $1/N$. The original objective of the Stern-Dill algorithm, though, was in part to increase the amount of memory available to store states, not to reduce the overall runtime requirements of verification.

A different algorithm was published in [30], this time targeting the SPIN model checker, but also restricted to safety properties. The objective of this algorithm was to improve the *IR* by using a partitioning method that exploits the structure of a SPIN model. The algorithm places an upper bound on the number of transitions that start at one CPU node and end at another. In a later algorithm [4], load-balancing decisions for state generation are treated separately from load balancing of state storage. Both the hash function in [37] and the partitioning method in [30] are used to achieve these load balancing decisions. The issue of load balancing was also addressed in [27].

7.1 Liveness Verification

The search algorithm that SPIN uses for the verification of liveness properties is based on a nested search, in postorder, from all accepting states reached in an initial depth-first search. The postorder discipline ensures that the verification consumes no more than twice the amount of time of a single (nonnested) search for safety properties. The order of expansion of the accepting states is therefore an important characteristic of the nested depth-first search [18].

An algorithm for distributed model checking of liveness properties on machines without shared memory was described in [2]. This algorithm maintains a separate dependency structure on a central CPU. The dependency structure enforces the required search order by ensuring that accepting states are expanded in the correct order. The memory requirements for the dependency structure can be significant though. Another algorithm [29] was designed to partition the state space in such a way over CPU nodes that accepting cycles always appear within the same CPU node. Load balancing is difficult to generalize with this algorithm, for instance, if the state graph consists of a single strongly connected component that spans all reachable states. In [6], [9], a different algorithm is studied that appears to incur quadratic complexity in the number of accepting states, defeating the benefit of the nested depth-first search. Other studies [8] have focused on determining search orders for the independent expansion of accepting states, though, so far, mostly without success.

7.2 Shared Memory Algorithms

A model-checking algorithm for shared memory systems was described in [25], restricted to safety properties and based on the use of disk storage. A different disk-based algorithm is proposed in [12] for the verification of liveness properties. The algorithm stores a copy of an accepting state inside the state vector and stops with a counterexample if that copy can be revisited. The seed state of the nested search is duplicated into every new state encountered during the second search, which can dramatically increase the memory requirements. In the worst case, the memory requirements are multiplied by the number of accepting states in the global state graph, which, in the worst case,

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

equals the number of reachable states. The states, though, are self-contained and can be transferred to any CPU node for expansion, thus simplifying the load balancing.

In [23], [24], a model-checking algorithm for shared memory systems is described for the logic CTL^* (which includes LTL, the logic used in SPIN). The algorithm uses both a shared and a nonshared work queue for each CPU node, plus a globally shared queue for all CPUs. Visited states are recorded in a shared state table. In this algorithm, each CPU retrieves the states found in its nonshared work queue first; next, it retrieves work from the globally shared queue, and, last, it can retrieve states from the shared queues of other CPUs. Newly generated states are inserted into either the private work queue of the CPU, its shared-work queue, or, if both queues are full, into the global queue. The process continues until all queues are empty, at which point, a distributed termination detection algorithm is initiated. The extension of the algorithm for CTL^* is given in terms of *hesitant alternating automata* and game theory, which makes it harder to compare it with the treatment of liveness properties in SPIN. Inggs and Barringer [24] include results for safety and liveness verification of several models, including a sliding window protocol, in a cluster system with up to 16 CPUs. There appears to be considerable variance in the runtimes reported in this study. In several cases, the performance was observed to degrade instead of improve when more CPUs were added. In at least one case, the reported performance for two CPUs was reported to be six times better than the reported performance for a single CPU ([24, Fig. 8, lower left]); a phenomenon that cannot be easily explained without more detailed information about the way the measurements were made. Possibly, the verification runs were stopped on the generation of a first counterexample, which would make it impossible to judge the overall performance of the algorithm for different numbers of CPUs used.

8 Conclusion

This paper describes the design of an extension of the SPIN model checker for multicore, shared memory systems. The extension supports the verification of both safety and liveness properties with a relatively small change in the SPIN source code itself. We have provided evidence to show that the effect of both compiler optimization techniques and search optimization techniques such as partial order reduction diminish the benefits of multicore processing. For applications of interest though, that is, large applications with embedded C code and relatively costly transition functions or large embedded data structures, the benefits, especially for the verification of safety properties, can be significant [19], [21], [22].

Our extension of SPIN deliberately preserves most of the existing verification modes of the tool, including a basic capability for the verification of liveness properties, but also the use of search optimization techniques such as partial order reduction, hash-compact state storage, or bitstate storage (the *supertrace* algorithm [15]). The capability of generating counterexamples with the multicore version of the model-checking algorithm is also preserved. In a single core version, counterexamples can be read from the search stack. In the new algorithm, this is not readily possible since only part of the stack is available to each CPU. To solve this, an optional tree-like data structure with backward pointers can be maintained so that error trails can also be generated when the trail spans states explored by different CPUs, at the cost of a small memory overhead for the storage of the additional data.

The sliced stack algorithm has several other initially unanticipated benefits, as also discussed in [22]. The multicore search algorithm in this mode combines properties of both breadth-first and depth-first searches. Note, for instance, that the first layer of the stack can explore all states at a depth of maximally d from the initial system state, without waiting for all subtrees of handoff states at level d to be fully explored first. The sliced stack algorithm can therefore find short counterexamples faster than a regular depth-first search while still retaining many of its benefits. Shorter stacks (of size d) can also have the benefit of reducing memory requirements for model-checking runs with large amounts of embedded source code. One example is the DS1 application that we used for some of our benchmark measurements. The stack requirements of this application make it impossible to perform a full verification run in a single core mode, but the verification succeeds in multicore mode.

The sliced stack algorithm should be applicable to any explicit state logic model-checking tool based on depth-first search. An adaptation to symbolic instead of explicit state model checking would likely be less straightforward. For safety properties, the algorithm scales well with increasing numbers of CPU cores in some cases nearly linearly (Fig. 7).

The liveness verification algorithm we have described is limited to the use of just two CPU cores. It has the property that, when most accepting states in the global state space are discovered *early*, then the nested part of the search can proceed largely in parallel with the nonnested part, resulting in a potential speedup. If, however, most accepting states are discovered *late* in the search, the benefit of parallelization is lost and performance can degrade to that of a single-core run with the added overhead of copying state descriptors. This means that the performance of the liveness algorithm depends on the property being verified, which makes its performance less predictable. Finding a liveness verification algorithm that retains the low complexity of the nested depth-first search method used in SPIN yet can scale with increasing numbers of CPU cores is as yet an open problem.

Appendix A

The set of all transitions that are enabled in a given state s is denoted $enabled(s)$. The reduced set of transitions (a subset of $enabled(s)$) for which successor states are generated when partial order reduction is applied is denoted as $r(s)$. The successor state of s when enabled transition t is applied is denoted as $t(s)$. We will also use the notation $\{s, t, s'\}$ to denote an execution step that starts in source state s and leads to destination state s' after the application of transition t such that $s' = t(s)$. Two enabled transitions t_1 and t_2 are *independent* at state s if 1) they do not disable one another when executed, that is, $t_1, t_2 \in enabled(s)$ implies

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

that $t_1 \in enabled(t_2(s))$ and $t_1 \in enabled(t_2(s))$, and 2) they commute, that is, $t_2(t_1(s)) = t_1(t_2(s))$ [17].

We say that a CPU "owns" a state if it has generated the state *and* will generate its successors (that is, the state is not a handoff state that is either sent from or to the CPU considered) [7].

We denote the full, unreduced, state space of a system by S and the reduced state space by S_r. The variant of partial order reduction we consider here satisfies the following two conditions:

C1. (persistence). For any state $s \in S$ and execution sequence $\{s, t_0, s_1\}; \{s_1, t_1, s_2\}; \ldots; \{s_{n-1}, t_{n-1}, s_n\}, \forall i, 0 \leq i < n$, $t_i \notin r(s)$ implies that, at s_{n-1}, transition t_{n-1} is *independent* of all transitions in $r(s)$.

C2. (cycle proviso). For any state $s \in S_r$ *owned* by CPU k, if $r(s) \neq enabled(s)$, then there exists at least one transition $t \in r(s)$ such that $t(s) \in S_r$ is *not*

1. on the DFS stack of CPU k,
2. previously generated by another CPU l with $l > k$, or
3. a handoff state.

The ignoring problem occurs when the application of an enabled transition t at some state s can be postponed indefinitely, that is, t is never included in some $r(s')$ such that there is an execution sequence in the reduced state space graph from s to s'. The following theorem states that this is impossible if conditions C1 and C2 are enforced, which also entails having all safety properties preserved by the POR algorithm [1], [38].

Theorem. *If the partial order reduction conforms to C1 and C2, then no enabled transition is ignored in any state s of S_r.*

Proof. Assume that we have P CPUs. We partition the states in S_r into P disjoint subsets $S_r^0, S_r^1, \ldots, S_r^{P-1}$ in such a way that state s is in S_r^k if it is generated by CPU k.

The proof is in two parts, each by induction. In the first part, we prove that no transition can be ignored in any state $s \in S_r^0$ by induction on the order in which states are removed from the depth-first stack of the search performed by CPU 0. In the second part, we prove that no transition can be ignored in any state $s \in S_r^k$ provided that no transition is ignored in any state $s \in S_r^l$ with $l < k$. In this part, we use combined induction on the CPU rank and the order in which states are removed from the depth-first stack of the search performed by CPU k.

We first deal with S_r^0, which establishes the base case of the first induction. Let s be the *first* state that is removed from the stack of CPU 0. When s is removed from the stack, each of its successors is either on the stack of CPU 0 previously generated by another CPU or is a handoff state. This is the negation of condition C2 for $r(s) \neq enabled(s)$. Hence, no enabled transition can be ignored for this state s.

Now, consider the nth state s that is removed from the stack of CPU 0. Assume as an induction hypothesis for the states in S_r^0 that no transition is ignored in any state that was removed from the stack before s. Assume further, that $r(s) \neq enabled(s)$—or else we are done—and consider transition $t' \notin r(s)$. By the cycle proviso C2, there exists another transition $t \in r(s)$ such that $t(s) \in S_r^0$, that is, it was either added to the state space earlier by CPU 0 or it is a new state. By the properties of the depth-first search, in both cases, $t(s)$ is removed from the stack before s. Because of C1, t' remains enabled in $t(s)$, that is, $t' \in r(t(s))$. By the induction hypothesis, t' cannot be ignored in $t(s)$. So, there exists an execution sequence σ in S_r starting at $t(s)$ and ending in some state s' such that $t' \in r(s')$. Adding the transition from s to $t(s)$ to σ produces an execution sequence that witnesses that t' is not ignored in s. In this way, we proved that no transition can be ignored in any state of S_r^0.

Next, consider a state $s \in S_r^k$, with $0 < k < P$. By assuming that no transition is ignored in (any state of) $S_r^0, S_r^1, \ldots, S_r^{k-1}$, we show that no transition is ignored in s. Analogously to the proof for S_r^0, we use induction on the order in which states are removed from the depth-first search stack of CPU k. Consider state s, which is removed first from the CPU k stack. Each of its successors $t(s)$, $t \in r(s)$, is either on the stack, a handoff state, or has been added to the state space by another CPU. The only difference with S_r^0 is in the last case: $t(s)$ could now have been added to the state space by a CPU whose rank is *lower* than k. As in S_r^0, in all other cases, C2 implies that s is fully explored. So, suppose that $t(s)$ has been generated by CPU l with $l < k$ and $r(s) \neq enabled(s)$ (or else we are done). In this case, $t(s) \in S_r^l$ and, therefore, by the induction hypothesis, no transition is ignored in $t(s)$. So, as shown above, we conclude that, for each $t' \notin r(s)$, there exists an execution sequence σ from $t(s)$ to some state s' and $t' \in enabled(s')$. By appending the transition from s to $t(s)$ to σ, we obtain a witness sequence.

Next, consider the nth state s that is removed from the stack of CPU k and assume that no transition is ignored in any state that is removed from the stack before s. We discuss only the nontrivial case when $r(s) \neq enabled(s)$. The only difference from the analogous case for S_r^0 is that there could exist a transition $t \in r(s)$ such that $t(s) \in S_r^l$ for some $l < k$. However, in this case, we can use the same arguments as that above for the first removed state by CPU k to conclude that no transition can be ignored in s. □

The validity of the theorem is also preserved with the versions of C1 proposed in [14], [38]. The latter imply that no transition $t' \notin r(s)$ is disabled by a transition in $t \in r(s)$, which is the only property entailed by C1 that is needed in the proof.

APPENDIX B

The reference model that was used for measuring the influence of three separate structural model parameters on the performance of multicore verification algorithms (Tables 3 and 4) is defined as follows:

```
#define BranchSize 8   /* successors per state */
#define StateSize 500  /* bytes in statevector */
#define TransTime 9    /* time to perform step */
#define NStates 500000 /* nr reachable states */
```

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

```
int count;
byte filler[StateSize];

active [BranchSize] proctype ref()
{
end:
  do
  :: d_step { /* one transition */
     count < NStates ->
     c_code {
        int xi;
        for (xi = 0; xi < (1 << TransTime); xi++)
        { now.filler[xi%StateSize] += xi%256;
        /* make sure filler is not eliminated */
        }
        /* avoid creating extra states */
        memset(now.filler, 0, StateSize);
     }; /* end of c_code */
      count++
     } /* end of d_step */
  od
} /* end of proctype */
```

ACKNOWLEDGMENTS

The authors are grateful to Matt Dwyer, Michael Jones, and Eric G. Mercer for their help in surveying and analyzing the earlier work in distributed model checking. Rajeev Joshi, Klaus Havelund, and Alex Groce provided significant feedback. Part of the work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the US National Aeronautics and Space Administration (NASA). The work of Gerard J. Holzmann was suppported by NASA's "Reliable Software Engineering" program, ESAS 6G. The work of Dragan Bošnački was supported in part by the UE FP6 project Evolving Signaling Networks in Silico (ESIGNET).

REFERENCES

[1] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Partial-Order Reduction in Symbolic State-Space Exploration," *Formal Methods in System Design*, vol. 18, pp. 97-116, 2001.
[2] J. Barnat, L. Brim, and J. Stribrna, "Distributed LTL Model-Checking in SPIN," *Proc. Eighth Int'l SPIN Workshop Model Checking of Software*, May 2001.
[3] J. Barnat, L. Brim, and J. Chaloupka, "Parallel Breadth-First Search LTL Model Checking," *Proc. Automated Software Eng.*, 2003.
[4] G. Behrmann, T. Hune, and F. Vaandrager, "Distributing Timed Model Checking—How the Search Order Matters," *Proc. 12th Int'l Conf. Computer Aided Verification*, pp. 216-231, 2000.
[5] D. Bosnacki and G.J. Holzmann, "Improving SPIN's Partial-Order Reduction for Breadth-First Search," *Proc. 12th Int'l SPIN Workshop*, pp. 91-105, 2005.
[6] L. Brim, I. Cerna, P. Moravec, and J. Simsa, "Accepting Predecessors Are Better than Back Edges in Distributed LTL Model Checking," *Formal Methods in Computer Aided Design*, pp. 266-352, 2004.
[7] L. Brim, I. Cerna, P. Moravec, and J. Simsa, "Distributed Partial Order Reduction of State Spaces," *Electronic Notes in Theoretical Computer Science*, vol. 128, pp. 63-74, 2005.
[8] L. Brim, I. Cerna, P. Moravec, and J. Simsa, "How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors," *Electronic Notes in Theoretical Computer Science*, vol. 135, pp. 3-18, 2006.
[9] I. Cerna and R. Pelanek, "Distributed Explicit Fair Cycle Detection," *Proc. SPIN Workshop*, pp. 49-73, 2003.
[10] S. Chien et al., "Using Autonomy Flight Software to Improve Science Return on Earth Observing One (EO1)," *J. Aerospace Computing, Information, and Comm.*, Apr. 2005.
[11] E.W. Dijkstra, "Shmuel Safra's Version of Termination Detection," EWD998, Jan. 1987.
[12] S. Edelkamp and S. Jabar, "Large-Scale Directed Model Checking LTL," *Proc. 13th Int'l SPIN Workshop*, pp. 1-18, 2006.
[13] P.R. Gluck and G.J. Holzmann, "Using Spin Model Checking for Flight Software Verification," *Proc. 2002 Aerospace Conf.*, Mar. 2002.
[14] P. Godefroid, *Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State Space Explosion.* Springer Verlag, 1996.
[15] G.J. Holzmann, *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
[16] G.J. Holzmann, P. Godefroid, and D. Pirottin, "Coverage Preserving Reduction Strategies for Reachability Analysis," *Proc. 12th Int'l Conf. Protocol Specification Testing and Verification*, pp. 349-363, June 1992.
[17] G.J. Holzmann and D. Peled, "An Improvement in Formal Verification," *Proc. Conf. Formal Description Techniques*, Oct. 1994.
[18] G.J. Holzmann, D. Peled, and M. Yannakakis, "On Nested Depth-First Search," *The SPIN Verification System*, pp. 23-32, Am. Math. Soc., 1996.
[19] G.J. Holzmann and M.H. Smith, "Automating Software Feature Verification," *Bell Labs Technical J.*, vol. 5, no. 2, pp. 72-87, Apr.-June 2000.
[20] G.J. Holzmann, *The SPIN Model Checker—Primer and Reference Manual.* Addison-Wesley, 2004.
[21] G.J. Holzmann and R. Joshi, "Model-Driven Software Verification," *Proc. 11th SPIN Workshop*, pp. 77-92, Apr. 2004.
[22] G.J. Holzmann, "A Stack-Slicing Algorithm for Multi-Core Model Checking," *Proc. Sixth Int'l Workshop Parallel and Distributed Methods on Verification*, July 2007.
[23] C.P. Inggs and H. Barringer, "Effective State Exploration for Model Checking on a Shared Memory Architecture," *Electronic Notes in Theoretical Computer Science*, vol. 68, no. 4, 2002.
[24] C.P. Inggs and H. Barringer, "CTL* Model Checking on a Shared-Memory Architecture," *Electronic Notes in Theoretical Computer Science*, vol. 128, pp. 107-123, 2005.
[25] S. Jabbar and S. Edelkamp, "Parallel External Directed Model Checker with Linear I/O," *Proc. Seventh Int'l Conf. Verification, Model Checking, and Abstract Interpretation*, pp. 237-251, 2006.
[26] R. Joshi and G.J. Holzmann, "A Mini-Challenge: Build a Verifiable Filesystem," *Formal Aspects of Computing*, vol. 19, 2007.
[27] R. Kumar and E.G. Mercer, "Load Balancing Parallel Explicit State Model Checking," *Proc. Third Int'l Workshop Parallel and Distributed Model Checking*, Aug. 2004.
[28] R.P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün, "Static Partial Order Reduction," *Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, pp. 345-357, 1998.
[29] A.L. Lafuente, "Simplified Distributed LTL Model Checking by Localizing Cycles," Technical Report 00176, Institut fur Informatik, Univ. Freiburg, Germany, July 2002.
[30] F. Lerda and R. Sisto, "Distributed-Memory Model Checking in SPIN," *Proc. SPIN Workshop*, 1999.
[31] G.E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, pp. 114-117, 1965.
[32] D. Peled, "Combining Partial Order Reductions with On-the-Fly Model-Checking," *Proc. Conf. Computer Aided Verification (CAV '94)*, pp. 377-390, 1994.
[33] J. Penix, W. Visser, C. Pasareanu, E. Engstrom, A. Larson, and N. Weininger, "Verifying Time Partitioning in the DEOS Scheduling Kernel," *Formal Methods in Systems Design J.*, vol. 26, no. 2, Mar. 2005.
[34] G.L. Peterson, "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115-116, June 1981.
[35] R. Pelanek, "Typical Properties of State Spaces," *Proc. 11th SPIN Workshop*, pp. 5-22, Apr. 2004.
[36] J.H. Reif, "Depth First Search Is Inherently Sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229-234, June 1985.
[37] U. Stern and D. Dill, "Parallelizing the Murφ Verifier," *Proc. Ninth Int'l Conf. Computer Aided Verification*, pp. 256-278, June 1997.
[38] A. Valmari, "The State Explosion Problem," *Lectures on Petri Nets I: Basic Models*, Springer, pp. 429-528, 1998.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.

Gerard J. Holzmann received the BSc and MSc degrees in electrical engineering and the PhD degree in technical sciences from Delft University, The Netherlands, in 1973, 1976, and 1979, respectively. He joined the Computing Sciences Research Center of Bell Laboratories in Murray Hill, New Jersey, in 1980 as a member of the technical staff. He was promoted to a distinguished member of the technical staff in 1995 and to director of Computing Principles Research in 2001. In 2003, he joined NASA's Jet Propulsion Laboratory, Pasadena, California, as a principal computing scientist and was appointed a JPL Fellow in 2007. He also serves as a faculty associate in the Computing Science Department at the California Institute of Technology, Pasadena. He is a member of the ACM and a member of the US National Academy of Engineering (NAE). He was the recipient of the 2001 ACM Software Systems Award, the 2002 ACM SIGSOFT Outstanding Research Award, and a corecipient of the 2006 ACM Paris Kanellakis Theory and Practice Award.

Dragan Bošnački received the BSc degree in electrical engineering and the MSc degree in computer science from St. Cyril and Methodius University, Skopje, Macedonia, and the PhD degree in computer science from Eindhoven University of Technology, The Netherlands. Currently, he is an assistant professor in the Biomedical Engineering Department at Eindhoven University of Technology. His main research interests include bioinformatics and formal methods and, in particular, in model checking. He is a member of the ACM and of the Dutch Union for Theoretical Computer Science. Earlier, he was a member of the Macedonian Association for Mathematics and Informatics, where he also served as the vice president.

▹ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on October 15, 2009 at 05:16 from IEEE Xplore. Restrictions apply.