# Multi-Core Model Checking with SPIN

Gerard J. Holzmann and Dragan Bošnački

***Abstract*—We present the first experimental results on the implementation of a multi-core model checking algorithm for the SPIN model checker. These algorithms specifically target shared-memory systems, and are initially restricted to dual-core systems. The extensions we have made require only small changes in the SPIN source code, and preserve virtually all existing verification modes and optimization techniques supported by SPIN, including the verification of both safety and liveness properties and the verification of SPIN models with embedded C code fragments.**

***Index Terms*—logic model checking, distributed algorithms, verification, multi-core, shared memory, safety, liveness, linear temporal logic.**

## I. Introduction

Model checking can be used to verify the correctness of distributed algorithms and asynchronous system designs, both for hardware and software. Thanks to a series of improvements over the last few decades and significantly helped by the steadily increasing power of CPUs, the range of problems that can be solved with model checking tools continues to expand. Model checkers such as SPIN today can analyze models with millions of reachable system states in a matter of seconds – which is more than adequate to support the verification of abstract design models of asynchronous software systems. As a result, logic model checking tools have become a standard part of safety critical systems development. The SPIN verifier [14] is a public-domain, open-source software tool, first introduced in 1989, and designed for the verification of correctness properties of asynchronous software systems. It is currently one of the most widely used verification tools in this domain.

The effectiveness of any verification method, whether it is applied manually or with computer support, is ultimately limited by problem complexity. Yet, the larger the range of problems we can analyze today, the stronger our desire to tackle still larger problems tomorrow. Sadly, the effect of Moore's curve [21] to drive a continuing increase in the performance of CPUs appears to be diminishing somewhat sooner than anticipated. In mid 2002, for instance, the fastest desktop PC ran at a clock-speed of 2.5 GHz. At the time of writing, late 2006, the fastest PC available ran at 3.8 GHz, where a continuation of Moore's curve would have predicted a clock-speed of 6.6 GHz. Chipmakers are currently focusing their attention on the further development of *multi-core* CPU systems. Dual-core and quad-core CPU systems are already widely available, with larger number of processing cores on the horizon. This means that to increase the problem solving capabilities of logic model checking tools in the foreseeable future we must develop strategies that can exploit the capabilities of *multi-core* CPU systems.

Multi-core systems provide all CPUs with access to fast shared memory, making inter-CPU data transfer much more efficient than it can be on a cluster-computer (or *multi-CPU* systems, as we shall call them here to distinguish them from *multi-core* systems). Given a word size of 64 bits there is no real limit to the amount of shared memory that could be addressed, making this the ideal context for the use of distributed model checking algorithms. The objective is of a multi-core extension of logic model checking algorithms is then to achieve reductions in the *runtime* requirements of a verification run, not to seek an increase in the amount of memory that can be addressed by all CPUs jointly. The improvement will be greatest if we can achieve maximal independence between the verification work that is done on different CPU cores. This means that the load balancing method is a critical factor in the design of a new algorithm.

We will discuss this issue in Section 2. In Section 3 we discuss an extension of SPIN's partial order reduction algorithm for multi-core algorithms. Section 4 provides metrics on the performance of the method we have implemented and presents an analysis of model characteristics that can enhance or degrade performance of multi-core algorithms. Section 5 reviews earlier work in this area and Section 6 concludes the paper.

## II. Load Balancing

A multi-core model checking task can be performed most efficiently if work can be distributed approximately evenly between the CPU cores. As little communication as possible should be required between the CPUs. This ideal is trivially realized if we can perform completely independent verification runs for distinct properties, e.g., specified as a

Part of this research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, as part of the 'Reliable Software Engineering' project, ESAS Project 6G**.**

G. J. Holzmann is with the Laboratory for Reliable Software, Jet Propulsion Laboratory, Pasadena, CA 91109 USA (phone: 818-393-5937; e-mail: firstname.lastname@ jpl.nasa.gov).

D. Bošnački., is with Eindhoven University, 5612 MB Eindhoven, The Netherlands (e-mail: dragan@win. tue.nl).

set of mutually independent LTL formulae. This was the method used for achieving even load balancing of hundreds of verification runs on a 16-CPU compute cluster in the Bell Labs FeaVer system [13]. For the verification of a single correctness property, though, this much 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 have the property that when a successor state *s* is generated by CPU *n*, most of the successors of *s* will also be explored by CPU *n*. If the CPUs use a shared data structure to store all states, there is no danger that the CPUs will start exploring the same parts of the state graph redundantly. The price to pay for this sharing of data is the enforcement of fine-grained mutual exclusion locks on access to the (relevant part) of the state tables (e.g., SPIN's hashtable), to prevent race conditions.

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 can be defined 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, that is: no sequence of transitions leads from the target state of an irreversible transition back to its source state.

Irreversible transitions divide the state graph into disjoint sub-graphs. They can trivially be identified statically in a SPIN model. Although *any* irreversible transition will divide the set of global system states into disjoint sub-sets, these sets are not always of similar size. The identification of irreversible transitions therefore is by itself not sufficient for defining a load balancing strategy. In SPIN there is one case though where we *can* identify an irreversible transition that divides the states space into two approximately equal and disjoint subsets. This is the transition that separates the first and the second search in the nested depth-first search algorithm [12,14]. Since the nested depth-first search enables the verification of *liveness*, this gives us a simple method for a *dual-core* extension of the model checking algorithm for liveness properties. To handoff a state from one CPU to another, it is simply copied into a work-queue in shared memory.

### *A. Liveness*

The 1st CPU adds each accepting state, in post-order [7], to the work queue of the 2nd CPU. The 2nd CPU retrieves state from this queue, and performs the nested part of the search to determine for each accepting state if is reachable from itself. The 2nd CPU records all new states it generates into a separate (non-shared) part of the state space, since there can be no overlap between the states generated in the 1st and the 2nd depth-first search. In this case, therefore, no locking is needed on access to the state tables. The basic complexity of the search remains unchanged compared to the single-core algorithm. Thus, for *dual*-core systems the speedup for the verification of liveness properties can be close to twofold (cf. Section 4).

### *B. Safety*

For the verification of safety properties, we adopt a similarly simple load balancing method that extends naturally also to multi-core systems with more than two cores. States are still transferred from one CPU to another via shared work-queues. These queues are always bounded. In liveness mode, there can only be state transfers in one direction: from the 1st to the 2nd CPU. When the work-queue of the 2nd CPU fills up, the 1st CPU will wait for a slot to become available. (A timeout allows it to recover from a possible crash of the 2nd CPU.) When checking safety properties though, state transfers can happen in any direction, and waiting on a full queue now runs the risk of deadlock. In this case, the sending CPU will always defer the handoff when a target queue is full and will explore the state locally instead. Note that the objective of load balancing is still achieved in this case, since the receiving CPU already has its maximal work load.

The metric for state handoffs for safety properties is based on the distance of a state from the root of the state space graph. Let ***d*** be the depth in the state graph at which a state is generated. Each CPU can handoff a newly generated successor state to another CPU when ***d*** exceeds a preset bound ***L*** within its local stack. When a state is transferred to another CPU, the target CPU will explore that state starting with an empty local stack and a search depth ***d*** of zero. This means that at every ***d%L*** steps from the original root of the global state graph, a state sequence can be transferred to another CPU. For ***L*** steps in the search, the CPUs can perform independent work (this time recording states in a shared state table), which means that we can control the degree of independence between CPUs by selecting an appropriate value for the handoff threshold ***L***. We will report on the performance of this method in Section IV.

The multi-core extension of SPIN can be achieved with minimal intrusion on the existing code by carefully selecting the points in the search where state handoffs can occur. Fig. 1 illustrates the nested depth-first search algorithm that is used in SPIN [14, p.180] and indicates two points in the search where a state can be handed off. These points are the natural recursion points in the depth-first search. By performing the extension in this way, the change to the existing system can be limited to a few hundred lines of new code. (In Fig. 1, `A.s0`, `A.T`, `A.F` are, respectively, *the initial state, the transition set, and the set of acceptance states* of the automaton that is obtained by combining the model and the property.)

```
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)
      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()    # start nested search
    Pop_Stack(D)
    seed = nil
  }

  Pop_Stack(D)
}
```

Fig. 1 – Handoff points for the dual-core nested depth first search algorithm.

The point marked [L] in Fig. 1 corresponds the start of the nested part of the search, which is the handoff point for the verification of liveness properties. The point marked [S] is the handoff point for the verification of safety properties, based on the depth metric. The two points interfere only minimally with the existing algorithm and preserve all other SPIN options. Since we are targeting only *dual*-core systems here, the two modes are never mixed. For liveness verification only handoff point [L] is used, and for safety verification only handoff point [S] is used.

A few supporting algorithms are used to complete the implementation. Peterson's algorithm for enforcing mutual exclusion in a platform independent way [22] was considered, but turns out to require CPU-specific adaptations to work correctly on modern CPUs.[1] The simpler solution in this case was to adopt small platform specific test-and-set instructions in assembly code instead. A distributed termination detection algorithm is also needed. The algorithm used for multi-core SPIN is based on Dijkstra's discussion of Safra's solution [8], which was verified with standard SPIN.

## III. Partial Order Reduction

The standard implementation of SPIN can achieve a considerable speedup from the use of a partial order reduction method that was introduced in [11] and revised in [12]. 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 guarantees that when a transition is deferred for later execution its continued executability is unaffected. 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 can close an infinite deferral cycle and lead to an incompleteness of the search process. 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 [11]. This pre-condition on the application of partial order reduction is known as the *cycle proviso* (or for depth-first search 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.[2] Clearly, in the case of a breadth-first search there is no depth-first stack, and thus an alternative method must be adopted to prevent the ignoring problem. The variants of the proviso now require that:

- Depth-first: *at least one* successor state appears outside the stack [10].
- Breadth-first: *all* successor states are previously unvisited [3].

The condition for breadth-first searches is independent of the stack contents, but generally achieves smaller reductions of the state space size. (An improvement, described in [3], and implemented in SPIN, is to require that at least one successor state appears within the breadth-first search queue.) In a multi-core search, similarly, 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. To achieve this, our implementation forces the exploration of *all* successor states in two extra places in the search (i.e., in addition to the case where a successor state is found on the local stack of the executing CPU).

- The *first* additional expansion is made for so-called "border states," that is states whose successors fall below the handoff depth of the current CPU, and therefore might have appeared on the search stack.

[1] The adaptation is needed to defeat out-of-order execution with so-called memory barrier functions.

[2] No efficient algorithm is known for the verification of liveness properties with a breadth-first search algorithm [13], so that combination is currently not supported.

Table 1 -- Performance improvement for verification of safety properties
(all runs exhaustive, generating the same number of states, without partial order reduction).

| Runtime for Verification of Safety Properties | Single-core (seconds) | Dual-core (seconds) | Ratio Dual/Single (%) |
|---|---|---|---|
| Sliding window protocol (W=5) | 39.73 | 23.17 | 58.3 |
| Leader election protocol (N=7) | 172.85 | 88.19 | 51.0 |
| Dining Philosophers (N=9) | 26.22 | 18.26 | 69.6 |
| Peterson's Algorithm (N=4) | 65.47 | 46.42 | 70.9 |

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.

The second case above can be optimized further by restricting it to cases where the previously visited state was generated by a CPU with a *higher* process number (pid) than the executing CPU, to ensure that the full expansion only occurs in one CPU, and not in both, as was also noted in [4].

*Proof Sketch for the correctness of the partial order reduction:* For the states generated by the CPU with the lowest pid*,* the correctness of the proviso follows from the proof for the standard depth-first search [10, 3], given that handoff states are treated as stack states, as well as the states generated by all other CPUs. All states are 'fully explored,' which means that no action enabled at any state can be deferred indefinitely. For the CPU with the next higher pid number we can rely on this fact, whenever it reaches a state generated by the first CPU. The proof generalizes in a straightforward way to any number of CPUs. Each CPU can trust that the successors of all states generated by CPUs with lower pid, are always fully explored.

Because of the full expansion of all border states, this version of the reduction method will work poorly for short handoff depths. As the data presented in the next section confirms though, for a handoff depth of 10 steps or more, this effect largely disappears.

## IV. Measurements

### A. Basic Performance

Table 1 shows a comparison of the runtime requirements of exhaustive verification runs with the dual-core extension of SPIN, for four different verification models taken from the SPIN distribution: a sliding window protocol for a window size of 5 messages, the leader election algorithm for a ring network of 7 processes, the dining philosophers problem with 9 nodes, and Peterson's generalized mutual exclusion algorithm for 4 concurrent processes. In all these measurements, the verification was performed for safety properties only without use of partial order reduction, to ensure that both the dual-core and single-core verification runs explore precisely the same number of reachable states. The runtime reductions, compared with the single core runs, vary from a near optimal ratio of 51% for the leader election example, to a less impressive ratio of just under 71% for Peterson's algorithm. We will study the reason for the smaller improvement for some models in Section IV.B.

#### 1) Liveness Verification

The performance of liveness verification gives similar results. For example, for the verification of the LTL property that eventually *one* node will be elected as the leader of the ring (in LTL: <>[]p), again for 7 nodes in the ring, we measured a runtime of 69.765 seconds for the standard single-core run, and 38.53 seconds for a dual-core run, giving a reduction to 55% of the runtime requirements.

#### 2) Influence of Compiler Optimization

Another interesting data point is obtained if we compare the performance of optimized verification runs with un-optimized runs. Table 2 shows the results, in this case for the sliding window protocol example with a window size of

Table 2 – Comparison of optimized and un-optimized performance.

| Runtime for Verification of Safety Properties | Single-core (seconds) | Dual-core (seconds) | Ratio Dual/Single (%) |
|---|---|---|---|
| Standard compilation | 39.73 | 23.17 | 58.3 |
| Optimized compilation (-O2) | 23.18 | 15.14 | 65.3 |

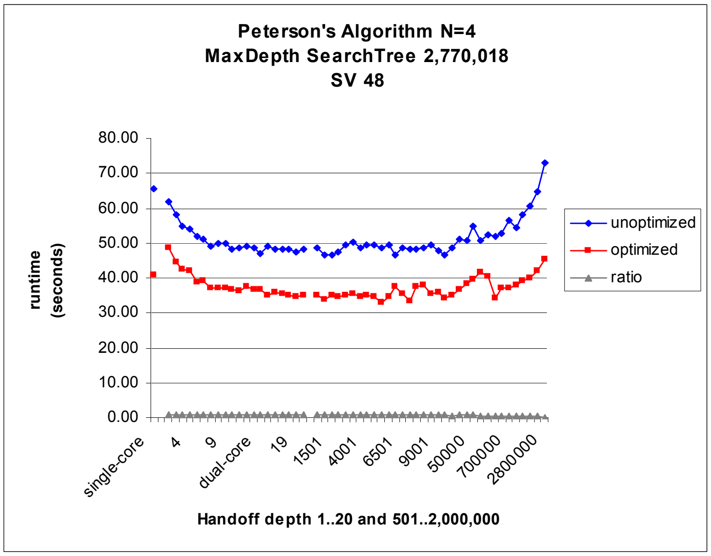


Fig. 2 – Measurement of the influence of the handoff depth chosen in dual core runs for safety properties. The two leftmost points indicate the performance of optimized (red, bottom) and unoptimized (blue, top) single core verification runs. The state vector (SV) size is 48 bytes for this model. The bottom ratio curve shows a load balance ratio close to one.

5 messages. For the un-optimized runs, the reduction in runtime achieved is 58%, but for the optimized run it is only 65%. Clearly, the overhead of inter-CPU state transfers becomes more noticeable for optimized code than it is for un-optimized code. The state handoff code itself consists of a simple memcpy call, which is hard to optimize further. Code optimization reduces time spent on independent computations. We will study this effect in more detail in Section 4.2. As an aside, it is also noteworthy to observe that merely enabling compiler optimization at level `-O2` suffices to achieve performance similar to an un-optimized dual-core verification run.

### 3) *Influence of Handoff Depth*

We measured the influence of handoff depth on the performance of dual-core verification. A representative result is shown in Fig. 2, in this case for a verification of safety properties for the leader election protocol with 4 nodes. The maximum depth of the search tree is 2.7 million steps. The state graph is acyclic, since all executions necessarily terminate with the election of a leader of the ring. The statevector is 48 bytes in this case, which is relatively small for a verification model.

Fig. 2 shows a characteristic "bath-tub" curve for the performance of the dual-core runs, with a conveniently long flat bottom where any handoff depth selected in this interval will give comparable performance. The left-hand sides of the curves, corresponding to the smaller handoff depth values, show relatively poor performance, since the load balance ratios are small for these values. In these measurements partial order reduction was disabled, to ensure that the single and dual core verification runs always explore the same numbers of states.

The right-hand sides of the curves also reveal growing performance degradation. This time the degradation is caused by approximating the maximum search depth of the state space itself – hence handoffs near and beyond this limit will not be able to achieve adequate load balancing anymore. If too few states are transferred from one CPU to the other, the first CPU will end up doing most of the work and the performance degrades to that of a single-core run, or worse (e.g., due to the overhead of the dual-core infrastructure needed).

### 4) *Influence of Partial Order Reduction*

In most cases, the use of partial order reduction preserves the benefits of dual-core verification, as expected. We show in Fig. 3 what seems at first to be a strongly anomalous result. In this case, for the verification of Peterson's mutual exclusion algorithm, there is not only no benefit from dual-core verification, the performance can actually degrade, despite perfect load balancing, especially for short handoff intervals. We can hypothesize that this is caused by the change that the partial order reduction makes in the state graph structure, reducing the average number of successor nodes of a state. This effect is not visible in most applications, but clearly important. We therefore investigate this phenomenon more fully in the next section.

## B. *Reference Model*

To verify how dual-core verification depends on various structural model characteristics, we construct a reference model that allows us to vary specific structural parameters over a range of values. This reference model is shown in Fig. 4. The model has three parameters: 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, and the time it takes to generate a state, which is

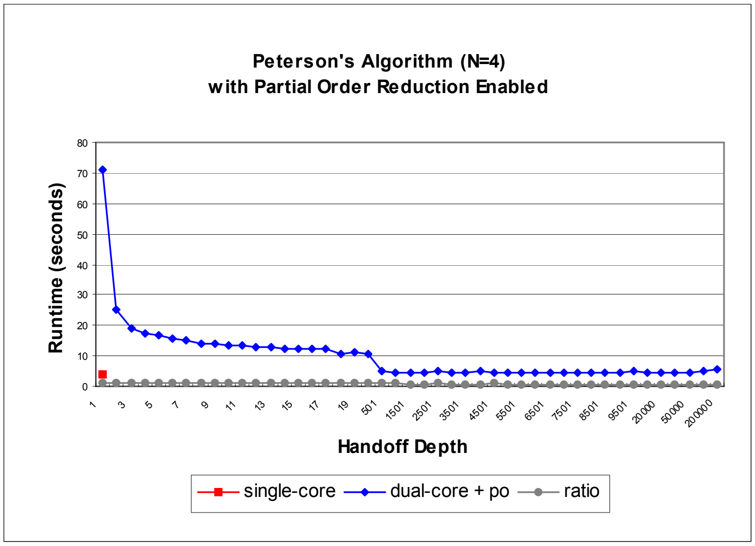


Fig. 3 – Anomalous performance of dual-core runs for Peterson's algorithm, when partial order reduction is enabled.

captured as the time to execute a state transition. We use embedded C code to control this parameter by the number of times the code executes a dummy computation.

The model always generates 500,000 reachable system states, independent of the parameters settings – to allow us to compare the relative runtimes across runs. The generated state space graph is a tree whose nodes have a predefined number of successors.

We first measure how the performance of dual-core verification depends on transition delay. Fig. 5 shows, in the top left graph, the ratio of dual-core runtime versus single-core run time, for transition delays that range from $2^1$ to $2^{18}$ time units. Three curves are plotted, the top curve (blue) is for a branch factor of one (every state reached always has a single successor), the middle curve (green) corresponds to a branch factor of two, and the bottom curve (black) corresponds to a branch factor of eight.

For models with small transition delays and/or small branch factors, a dual-core run can take up to 2½ times as long as a single-core run. For models with an average out-degree of eight though, the dual-core runs are never slower than single-core runs. The same effect is observed for larger transition delays and branch factors above one. Since partial order reduction reduces the branch factor, the slowdown of dual-core verification runs when partial order reduction is used is now explained.

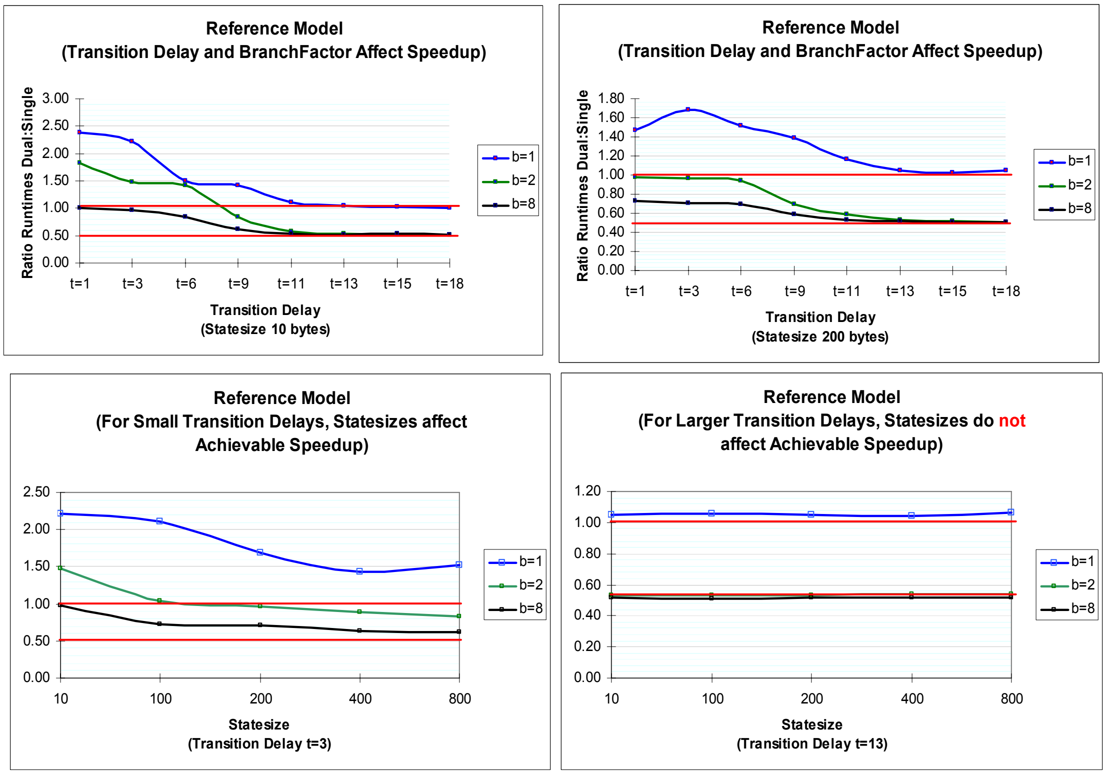
The graph on the upper right in Fig. 5 repeats the measurements for a larger state size. We see the same

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

int count;
byte filler[StateSize];

active [BranchSize] proctype test()
{
end: do
     :: d_step {    /* define one single transition step */
          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 */
            }
            /* make sure no extra states are created */
            memset(now.filler, 0, StateSize*sizeof(char));
          };
         count++
        }
     od
}
```

*Fig. 4—Reference model for measuring the influence of structural model parameters on the performance of multi-core model checking algorithms.*


Reference Model (Transition Delay and BranchFactor Affect Speedup)

Reference Model (Transition Delay and BranchFactor Affect Speedup)

Reference Model (For Small Transition Delays, Statesizes affect Achievable Speedup)

Reference Model (For Larger Transition Delays, Statesizes do not affect Achievable Speedup)

Fig. 5 – Measurements on the reference model, to study the effect of structural model properties on the performance of multi-core algorithms. Varied are the out-degree of states (the branch factor), the state size, and the transition delay.

effect, but for branch-factors above one (which correspond to deterministic models) the performance degradation disappears and optimal performance can be realized. The graph on the lower left side in Fig. 5 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 independent of state size, because it is now dominated by transition delay.

A few observations can be made about these results. First, the performance of dual-core algorithms should be expected to be smaller when partial order reduction is used then when it is not used, although the effect can be mitigated by a number of other factors. In cases where partial order reduction is not an option, for instance in the verification on non-stutter-invariant liveness properties, multi-core algorithms can prove especially valuable. Note that for deterministic models, no multi-core state partitioning method is likely to be effective. In these cases one processor will inevitably always be waiting for other processors to complete its work, as shown in Fig. 5.

Another observation is that the multi-core algorithms can be expected to perform especially well for models with large state sizes and/or large transition delays. This nicely fits an important application domain of verification models with embedded software and model-driven verification techniques where the model checker must control and track potentially large amounts of implementation level code.

The importance of this type of extension should therefore be expected to increase over time as we start tackling larger problem sizes. Alas, it also means that multi-core systems cannot really show their full potential on small class-room size examples, so some tutorial value of this important new class of algorithms is lost.

## V. Earlier Work

Most work in distributed model checking to date has been focused on algorithms for the verification of *safety* properties on *physically distributed* computers. An early objective was also to increase the amount of *memory* that could be dedicated to verification. As noted, many of these early assumptions are no longer valid with the switch to multi-core systems with a 64-bit address space.

The Stern-Dill algorithm [23] uses a hash function to assign states to nodes in a compute cluster. With a good hash-function, this method should achieve near-optimal load balancing, but it suffers from the overhead of frequent state transfers. A different cluster algorithm for Spin appears in [20], but also restricted to safety properties. This algorithm performed load balancing with a partitioning method that is based on the structure of a SPIN model. The issue of load balancing was also addressed in [2] and [18].

An algorithm for distributed model checking of liveness properties is given in [1]. The algorithm maintains a global structure on one CPU to ensure that accepting states are expanded in the correct order. The memory requirements for this structure can be prohibitive though. Another algorithm [19] was designed to partition the state space in such a way that accepting cycles always appear within the same CPU. Load balancing is difficult to generalize with this method. In [5] and [6] algorithms are described that require multiple passes over the state space, increasing the computational complexity of the search process, and potentially defeating the benefit of a parallelized verification process.

An algorithm for safety properties is given in [17], and for liveness properties in [9], both using disk memory. The liveness algorithm stores a copy of an accepting state inside the state vector and stops with a counter-example if that copy can be matched. The seed state of the nested search then is duplicated into every new state encountered during the second search, which can significantly increase the memory requirements for that part of the search.

A model checking algorithm for CTL*, using shared memory is given in [16]. To achieve load balancing, idle CPUs can “steal” states from the work-queues of other CPUs. Performance results are given, but only for runs that stop at the generation of a first counter-example, which makes it difficult to compare results between single- and multi-core runs, or for different versions of the algorithms.

## VI. Conclusion

This paper describes how the SPIN model checker can be extended for multi-core systems with shared memory. The extension supports the verification of both safety and liveness properties, with a relatively small change. We have shown that the effect of compiler optimization and search optimization techniques, such as partial order reduction, diminish the benefit of multi-core processing. For applications of interest though, applications with embedded code [13,15], the benefits can be significant.

Our extension preserves most of the existing verification modes of SPIN, including the capability to verify liveness properties, the use of search optimization techniques such as partial order reduction, and also bitstate storage [14]. The capability to generate counter-examples with the multi-core version of the model checking algorithm is also preserved through the use of backward pointers in the state graph.

The method we have described for the verification of safety properties scales without change to the use of larger numbers of CPU cores. The extension for liveness for more than two CPUs remains an open research problem and is expected to be non-trivial.

## Acknowledgment

The authors are grateful to Rajeev Joshi for discussions and to Matt Dwyer, Michael Jones, and Eric G. Mercer for an in-depth overview of the earlier work in this area.

## References

[1] J. Barnat, L. Brim, J. Strıbrná. Distributed LTL model-checking in SPIN. *Proc. 8th SPIN Workshop on Model Checking of Software*, LNCS 2057, May 2001.

[2] G. Behrmann, T. Hune, F. Vaandrager, Distributing timed model checking—How the search order matters, *Proc. CAV 2000*, LNCS 1855, pp. 216-231.

[3] D. Bosnacki, G.J. Holzmann, Improving SPIN's Partial-Order Reduction for Breadth-First Search, *Proc. 12th SPIN Workshop*, LNCS 3639, pp.91-105, 2005.

[4] L.Brim, I.Cerna, P. Moravec, J. Simsa, Distributed partial order reduction of state spaces, Electronic Notes in Theoretical Computer Science, 128 (2005), pp. 63-74.

[5] L. Brim, I. Cerna, P. Moravec, J. Simsa, How to order vertices for distributed LTL model-checking. *Electronic Notes in Theoretical Computer Science*, Vol. 135, pp. 3-18, 2006.

[6] I. Cerna, R. Pelanek. Distributed explicit fair cycle detection. *Proc. SPIN workshop*, LNCS 2648, pp. 49–73, 2003.

[7] C. Courcoubetis, M.Y. Vardi, P. Wolper, M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. *Proc. CAV*, 1990, pp. 233-242.

[8] E. W. Dijkstra. *Shmuel Safra's version of termination detection, EWD998.*

[9] S. Edelkamp and S. Jabar, Large-Scale Directed Model Checking LTL, *13th SPIN Workshop on Model Checking of Software*, pp. 1-18, LNCS 3925, 2006.

[10] G.J. Holzmann, P. Godefroid, and D. Pirottin, Coverage preserving reduction strategies for reachability analysis. *Proc. 12th Conf on Protocol Specification Testing and Verification*, Orlando, FL., 1992.

[11] G.J. Holzmann and D. Peled, An Improvement in Formal Verification, *Proc. Conf. on Formal Description Techniques*, 1994.

[12] G.J. Holzmann, D. Peled, M. Yannakakis, On Nested Depth-First Search, *The SPIN Verification System*, AMS, 1996, pp. 23-32.

[13] G.J. Holzmann, M.H. Smith, Automating software feature verification, *Bell Labs Techn. Journal*, Vol. 5, No. 2, pp. 72-87, 2000.

[14] G.J. Holzmann, *The SPIN Model Checker*, Addison-Wesley, 2004.

[15] G.J. Holzmann, R. Joshi, Model-driven software verification, *Proc. 11th SPIN Workshop*, Barcelona, 2004, LNCS 2989, pp. 77-92.

[16] C.P. Inggs, H. Barringer, CTL* Model Checking on a Shared-Memory Architecture, *Electronic Notes in Theoretical Computer Science*, Vol. 128, 2005, pp. 107-123.

[17] S. Jabbar, S. Edelkamp, Parallel External Directed Model Checker with Linear I/O, *Proc. Conf. on Verification, Model Checking and Abstract Interpretation.*, LNCS 3855, 2006, pp. 237-251.

[18] R. Kumar, E.G. Mercer, Load Balancing Parallel Explicit State Model Checking, *Proc. 3rd Workshop on Parallel and Distributed Model Checking* (PDMC), August 2004.

[19] A.L. Lafuente. Simplified distributed LTL model checking by localizing cycles. *Tech. Rep. 00176*, Univ. Freiburg, Germany, 2002.

[20] F. Lerda, R. Sisto, Distributed-memory model checking in SPIN, *Proc. SPIN Workshop*, LNCS 1680, 1999.

[21] G.E. Moore, Cramming more components onto integrated circuits, *Electronics*, Vol. 38, pp. 114-117, 1965.

[22] G.L. Peterson, Myths about the Mutual Exclusion Problem, *Inf. Processing Letters*, Vol. 12, No. 3, pp. 115-116, June 1981.

[23] U. Stern, D. Dill. Parallelizing the Murφ verifier. *Proc. 9th Conf. on Computer Aided Verification* (CAV’97), Haifa, Israel, June 1997, Springer–Verlag, LNCS 1254, pp 256–278.