Squeeze All the Power Out of Your Hardware to Verify Your Software!

Jiří Barnat and Lubos Brim*

Faculty of Informatics, Masaryk University
Brno, Czech Republic

Abstract. The computer industry is undergoing a paradigm shift. Chip manufacturers are shifting development resources away from single-processor chips to a new generation of multi-processor chips, huge clusters of multi-core workstations are easily accessible everywhere, external memory devices, such as hard disks or solid state disks, are getting more powerful both in terms of capacity and access speed. This fundamental technological shift in core computing architecture will require a fundamental change in how we ensure the quality of software. The key issue is that verification techniques need to undergo a similarly deep technological transition to catch up with the complexity of software designed for the new hardware. In this position paper we would like to advocate the necessity of fully exploiting the power offered by the new computer hardware to make the verification techniques capable of handling next-generation software.

1 Introduction

The computing power of computers has increased by a factor of a million over the past couple of decades. As a matter of fact, the development effort, both in industry and in academia, has gone into developing bigger, more powerful and more complex applications. In the next few decades we may still expect a similar rate of growth, due to various factors such as continuing miniaturization, parallel and distributed computing.

With the increase in complexity of computer systems, it becomes even more important to develop formal methods for ensuring their quality and reliability. Various techniques for automated and semi-automated analysis and verification have been successfully applied to real-life computer systems. However, these techniques are computationally demanding and memory-intensive in general and their applicability to extremely large and complex systems routinely seen in practice these days is limited. The major hampering factor is the state space explosion problem due to which large industrial models cannot be efficiently handled unless we use more sophisticated and scalable methods and a balance of the usual trade-off between run-time, memory requirements, and precision of a method.

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/I338 and the Academy of Sciences grant No. 1ET408050903.
A lot of attention has been paid to the development of approaches to battle the state space explosion problem. Many techniques, such as abstraction, state compression, state space reduction, symbolic state representation, etc., are used to reduce the size of the problem to be handled allowing thus a single old-fashioned single-processor computer to process larger systems. All these methods can be therefore characterized as “reduction” techniques.

Verification and analysis methods that are tailored to exploit the capabilities of the new hardware architectures are slowly appearing as well. These “platform-dependent” techniques focus on increasing the amount of available computational power. These are, for example, techniques to fight memory limits with efficient utilisation of external I/O devices, techniques that introduce cluster-based algorithms to employ aggregate power of network-interconnected computers, or techniques to speed-up the verification on multi-core processors.

The idea of exploiting hard disks or parallel computers in verification already appeared in the very early years of the formal verification era. However, inaccessibility of cheap parallel computers with sufficiently fast external memory together with negative theoretical complexity results excluded these approaches from the main stream in formal verification. The situation changed dramatically during the past several years. The computer progress over the past two decades has measured several orders of magnitude with respect to various physical parameters such as computing power, memory size at all hierarchy levels from caches to disk, power consumption, physical size and cost. In particular, the focus of novel computer architectures in parallel and distributed computing has shifted away from unique massively parallel systems competing for world records towards smaller and more cost effective systems built from personal computer parts. In addition, recent shift in the emphasis of research on parallel algorithms to pragmatic issues has provided practically efficient algorithms for solving computationally hard problems. As a matter of fact, interest in platform-dependent verification has been revived.

2 Running Example: Enumerative LTL Model-Checking

Model checking is one of the major techniques used in the formal verification [17]. It builds on an automatic procedure that takes a model of the system and decides whether it satisfies a given property. In case the property is not satisfied, the procedure gives a counterexample, i.e. a particular behaviour of the model that violates the verified property.

In order to demonstrate, overview and advocate the advantages we can gain by exploiting the new hardware possibilities in more technical setting, we consider one particular verification problem, namely enumerated LTL model checking. Similar conclusions can be drawn for other verification problems as well.

An efficient procedure to decide LTL model checking problem is based on automata and was introduced in [42]. The approach exploits the fact that every set of executions expressible by an LTL formula is an ω-regular set and can be described by a Büchi automaton. In particular, the approach suggests to express
all system executions by a system automaton and all executions not satisfying the formula by a property or negative claim automaton. These automata are combined into their synchronous product in order to check for the presence of system executions that violate the property expressed by the formula. The language recognised by the product automaton is empty if and only if no system execution is invalid.

The language emptiness problem for Büchi automata can be expressed as an accepting cycle detection problem in a graph. Each Büchi automaton can be naturally identified with an automaton graph which is a directed graph \( G = (V, E, s, A) \) where \( V \) is the set of vertices \( (n = |V|) \), \( E \) is a set of edges \( (m = |E|) \), \( s \) is an initial vertex, and \( A \subseteq V \) is a set of accepting vertices. We say that a cycle in \( G \) is accepting if it contains an accepting vertex. Let \( A \) be a Büchi automaton and \( G_A \) the corresponding automaton graph. Then \( A \) recognises a nonempty language if \( G_A \) contains an accepting cycle reachable from \( s \). The LTL model-checking problem is thus reduced to the accepting cycle detection problem in the automaton graph.

The optimal sequential algorithms for accepting cycle detection use depth-first search strategies to detect accepting cycles. The individual algorithms differ in their space requirements, length of the counter example produced, and other aspects. For a recent survey we refer to [41]. The well-known Nested DFS algorithm is used in many model checkers and is considered to be the best suitable algorithm for enumerative sequential LTL model checking. The algorithm was proposed by Courcoubetis et al. [18] and its main idea is to use two interleaved searches to detect reachable accepting cycles. The first search discovers accepting states while the second one, the nested one, checks for self-reachability. Several modifications of the algorithm have been suggested to remedy some of its disadvantages [23]. Another group of optimal algorithms are SCC-based algorithms originating in Tarjan’s algorithm for the decomposition of the graph into strongly connected components (SCCs) [40]. While Nested DFS is more space efficient, SCC-based algorithms produce shorter counterexamples in general. For a survey we refer to [21]. The time complexity of these algorithms is linear in the size of the graph, i.e. \( O(m + n) \), where \( m \) is the number of edges and \( n \) is the number of vertices.

The effectiveness of the Nested DFS algorithm is achieved due to the particular order in which the graph is explored and which guarantees that vertices are not re-visited more than twice. In fact, all the best-known algorithms rely on the same exploring principle, namely the postorder as computed by the DFS. It is a well-known fact that the postorder problem is P-complete and, consequently, e.g. a scalable parallel algorithm which would be directly based on DFS postorder is unlikely to exist.

An additional important criterion for a model checking algorithm is whether it works on-the-fly. On-the-fly algorithms generate the automaton graph gradually as they explore vertices of the graph. An accepting cycle can thus be detected before the complete set of vertices is generated and stored in memory. On-the-fly algorithms usually assume the graph to be given implicitly by the function
$F_{\text{init}}$ giving the initial vertex and by the function $F_{\text{succ}}$ which returns immediate successors of a given vertex.

3 Get New Algorithms!

<table>
<thead>
<tr>
<th>Position Statement:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recent architectural shift means that it is no longer possible to benefit from hardware progress, without introducing algorithmic changes to our tools. New algorithms have to be designed.</td>
</tr>
</tbody>
</table>

In many cases the algorithms as used traditionally are not appropriate to be adopted to the new hardware architectures. This can be demonstrated by LTL model checking algorithms as mentioned above. All the efficient algorithms build on depth-first search exploration of the state space. However, there is no known way to compute DFS postorder when using hard disks or parallel architectures efficiently. New algorithms, often radically different, have to be invented to replace the classical ones.

We will support this argument by presenting two different algorithms for accepting cycle detection (LTL model checking). Sequential complexity of these algorithms is worse than those based on DFS, but both allow to solve the LTL model-checking problem on new hardware architectures much more efficiently as will be exemplified later in the paper. Here we consider only two such algorithms. One is the MAP algorithm [13, 14], the other one is the enumerative OWCTY algorithm [15]. For a survey on these and other algorithms we refer to [2].

Algorithm Based on Topological Sort

The main idea behind the OWCTY algorithm comes out from the fact that a directed graph can be topologically sorted if and only if it is acyclic. The core of the accepting cycle detection algorithm is thus in application of the standard linear topological sort algorithm to the input graph. Failure in topologically sorting the graph means the graph contains a cycle.

The algorithm performs a cycle detection that is based on the recursive elimination of vertices with zero predecessors. At first, the algorithm computes reachability to remove vertices from which no accepting state is reachable (these cannot belong to any accepting cycle) and computes the number of immediate predecessors for every reachable vertex. Then the algorithm eliminates vertices whose predecessor count drops to zero. During vertex elimination, the predecessor count is decreased for all immediate successors of the eliminated vertex.

The algorithm does not work on-the-fly and the entire automaton graph has to be generated first. Also, the algorithm does not immediately give the accepting cycle, it only checks for its presence in the graph. However, the counter-example is easily generated using two additional linear graph traversal, like BFS.

Time complexity of the algorithm is $O(h \cdot m)$ where $h$ is the height of the corresponding quotient graph (the graph of strongly connected components). Here
the factor \( m \) comes from the computation of \textit{Reachability} and \textit{Elimination} functions and the factor \( h \) relates to the number of external iterations. In practice, the number of external iterations is very small (up to 40-50) even for very large graphs. This observation is supported by experiments in [22]. Similar results are communicated in [36] where heights of quotient graphs were measured for several models. As reported, 70% of the models have heights smaller than 50.

A positive aspect of the algorithm is its extreme effectiveness for \textit{weak automaton graphs}. A graph is weak if in each strongly connected component all the states are accepting or none of them is. For weak graphs only one iteration of the algorithm is necessary to decide accepting cycles, the algorithm works in linear time and is thus optimal. The studies of temporal properties [19, 16] reveal that verification of up to 90% of LTL properties leads to weak automaton graphs.

**Maximal Accepting Predecessors Algorithm**

The main idea behind the MAP algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own predecessor. The algorithm that would be directly derived from such an idea requires expensive storing of all proper accepting predecessors for each (accepting) vertex. To remedy this, the algorithm stores only a single representative accepting predecessor for each vertex. We presuppose a linear ordering \( \prec \) of vertices (given e.g. by their memory representation) and choose the \textit{maximal accepting predecessor}. For a vertex \( u \) we denote its maximal accepting predecessor in the graph \( G \) by \( \text{map}_G(u) \). Clearly, if an accepting vertex is its own maximal accepting predecessor \( (\text{map}_G(u) = u) \), it lies on an accepting cycle. Unfortunately, the opposite does not hold in general. It can happen that the maximal accepting predecessor for an accepting vertex on a cycle does not lie on the cycle. This is exemplified on the graph given in Fig. 1. The accepting cycle \((2, 1, 3, 2)\) is not revealed due to the greater accepting vertex 4 outside the cycle. However, as vertex 4 does not lie on any cycle, it can be safely deleted (marked as non-accepting) from the set of accepting vertices and the accepting cycle still remains in the resulting graph. This idea is formalised as a \textit{deleting transformation}. Whenever the deleting transformation is applied to the automaton graph \( G \) with \( \text{map}_G(v) \neq v \) for all \( v \in V \), it shrinks the set of accepting vertices by those vertices that do not lie on any cycle. As the set of accepting vertices can change after the deleting transformation has been applied, maximal accepting predecessors must be recomputed. It can happen that even in the graph \( \text{del}(G) \) the maximal accepting predecessor function is still not sufficient for cycle detection. However, after a finite number of iterations consisting of computing maximal accepting predecessors followed by application of the deleting transformation an accepting cycle is certified. For an automaton graph without accepting cycles the repetitive application of the deleting transformation results in an automaton graph with an empty set of accepting vertices.

![Fig. 1. Undiscovered cycle](image-url)
Time complexity of the algorithm is $O(a^2 \cdot m)$, where $a$ is the number of accepting vertices. Here the factor $a \cdot m$ comes from the computation of the map function and the factor $a$ relates to the number of iterations. Unlike the OWCTY algorithm, the MAP algorithm works on-the-fly.

Experimental evaluation of this algorithm demonstrated that accepting cycles were typically detected in a very small number of iterations. On the other hand, if there is no accepting cycle in the graph, the number of iterations is typically very small compared to the size of the graph (up to 40-50). Thus, the algorithm exhibits nearly linear performance in practice.

4 Squeeze the Juice Out of Your Hard Disk!

POSITION STATEMENT:

*External memory devices provide a viable computational alternative in analysing and verifying very large systems. With special external memory efficient techniques we can touch verification problems that are far beyond the capabilities of pure RAM approaches.*

Hard disk has traditionally been regarded as a hopelessly slow cousin to RAM. However, the bandwidth of commodity disks today is on the order of 100MB/s. Is this enough to consider disk-based computation a way to increase working memory and achieve results that are not otherwise economical?

For external memory devices, the goal is to develop algorithms that minimize the number of I/O operations an algorithm has to perform to complete its task. This is because the access to information stored on an external device is orders of magnitude slower than the access to information stored in the main memory. Thus the complexity of I/O efficient algorithms is measured in the number of I/O operations only.

A distinguished technique that allows for an I/O efficient implementation of a graph traversal procedures is the so called *delayed duplicate detection* [32, 35, 39]. A traversal procedure has to maintain a set of visited vertices to prevent their re-exploration. Since the graphs are large, the set cannot be completely kept in the main memory and must be stored on the external memory device. When a new vertex is generated it is checked against the set to avoid its re-exploration. The idea of the delayed duplicate detection technique is to postpone the individual checks and perform them together in a group for the price of a single scan operation.

Unfortunately, the delayed duplicate detection technique is incompatible with the depth-first search (DFS) of a graph [20]. Therefore, the first approaches to I/O efficient LTL model checking have focused on the state space generation and verification of safety properties only. The very first I/O efficient algorithm for state space generation has been implemented in Munz [39]. Later on, several heuristics for the state space generation were suggested and implemented in various verification tools [26, 30, 33]. The first attempt to verify more than safety properties has been described in [31]. The approach uses the random search to
find a counterexample to a given property, it is thus incomplete in the sense that it is not able to prove validity of the property.

A complete I/O efficient LTL model checker was suggested in [20] (we refer to it as IO-EJ) where the problematic DFS-based algorithm was avoided by the reduction of the accepting cycle detection problem to the reachability problem [10, 38] whose I/O efficient solution was further improved by using the directed (A*) search and parallelism. The algorithm works in the on-the-fly manner meaning that only the part of the state space is constructed, which is needed in order to check the desired property. The reduction transforms the graph so that the size of the graph after the transformation is asymptotically quadratic with respect to the original one. As the external memory algorithms are meant to be applied to large graphs, the quadratic increase in the size of the graph is significant. This is especially the case when the model is valid and the entire graph has to be explored to prove the absence of an accepting cycle. The approach is thus mainly useful for finding counterexamples (falsification) in the case a standard verification tool fails due to the lack of memory.

Another complete I/O efficient LTL model checking algorithm (IO-OWCTY) has been proposed in [5]. The algorithm builds on the topological sort algorithm described in Section 3. Remember that this algorithm does not rely on the DFS postorder, hence is compatible with the delayed duplicate detection technique.

The algorithm uses BFS to traverse the graph and basically maintains three data structures: a set of vertices that await processing (open set), a set of vertices that have been processed already (closed set), and a set of candidates, i.e., vertices for which the corresponding check against the closed set has been postponed. The way in which vertices are manipulated is depicted in Fig. 2(a). A vertex from the open set is selected and its immediate successors are generated. The newly generated vertices are checked against the candidate set, to ensure that information stored in the candidate set is properly updated. Also, if there is need for further processing of some vertices, they are inserted back into the open set along with all necessary information for the processing. As the check is not done directly against the closed set, this is the point where duplicates might appear.

Candidates are flushed to disk to resolve duplicates using a merge operation under two different circumstances: either the open set runs empty and the algorithm has to perform a merge to get new vertices into it, or the candidate set is too large and cannot be kept in memory anymore. The merge operation performs the duplicate check of candidate vertices against closed vertices, and inserts those vertices that require further processing into the open set.

A weak point of the IO-OWCTY algorithm is that the merge operation is performed every time the algorithm empties the set of open vertices, which happens at least after every BFS level. Often a single BFS level contains a relatively small number of vertices, in comparison to the full graph. Processing them means that the merge operation has to traverse a large disk file, which is costly.

In [6] a different algorithm (IO-MAP) to fight this inefficiency was suggested. This algorithm builds on the maximal accepting predecessors algorithm as de-
scribed in Section 3. The distinguished feature of the algorithm is that it allows more BFS levels to be explored at once without destroying the correctness of the algorithm (the algorithm is termed revisiting resistant). The substantial modification in the vertex work flow of an I/O efficient algorithm is depicted in Fig. 2(b). A vertex, when generated, is inserted not only into the set of candidates, but also into the open set. This causes some of the vertices stored in the candidate set to be revisited. I.e., the “visit” procedure is performed repeatedly for a vertex without properly updating its associated information in the closed set residing in external memory. Note that some graph algorithms, like topological sort, may exhibit incorrect behavior in this case. There is another very important difference between IO-OWCTY and IO-MAP, namely the latter works on-the-fly.

To demonstrate how I/O efficient LTL model checking algorithms compare, we conclude with some experimental measurements. All the models and their LTL properties are taken from the BEEM project [37]. The results are listed in Table 1. We noticed that just before an unsuccessful termination of IO-EJ due to exhausting the disk space, the BFS level size still tended to grow. This suggests that the computation would last substantially longer if sufficient disk space would have been available. For the same input graphs, the algorithms IO-OWCTY and IO-MAP manage to perform the verification using a few Gigabytes of disk space only.

Evaluation on models with valid properties demonstrates that IO-MAP is able to successfully prove their validity, while IO-EJ fails. The IO-MAP with its revisiting resistant techniques is able to outperform IO-OWCTY in many cases. We observed that specifically in cases with high $h_{BFS}$, e.g., Rether(16,8,4),P2, time savings are substantial. A notable weakness of IO-OWCTY is its slowness on models with invalid properties. It does not work on-the-fly, and is consequently outperformed by IO-EJ in the aforementioned class of inputs. The algorithm IO-MAP does not share IO-OWCTY’s drawbacks, and in fact it outperforms both, IO-OWCTY and IO-EJ on those inputs. This can be attributed to their on-the-fly nature (on all our inputs, a counter example, if existing, has been found during the first iteration).
Table 1. Run times in hh:mm:ss format and memory consumption on a single work-
station for I/O LTL algorithms. “OOS” means “out of space”.

<table>
<thead>
<tr>
<th>Valid Properties</th>
<th>IO-EI</th>
<th>IO-OWCTY</th>
<th>IO-MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lamport(5),P4</td>
<td>(OOS)</td>
<td>02:37:17</td>
<td>5.5 GB</td>
</tr>
<tr>
<td>MCS(5),P4</td>
<td>(OOS)</td>
<td>03:27:05</td>
<td>9.8 GB</td>
</tr>
<tr>
<td>Peterson(5),P4</td>
<td>(OOS)</td>
<td>18:20:03</td>
<td>26 GB</td>
</tr>
<tr>
<td>Philk(16,1),P3</td>
<td>(OOS)</td>
<td>01:49:41</td>
<td>6.2 GB</td>
</tr>
<tr>
<td>Rether(16,8,4),P2</td>
<td>53:06:44</td>
<td>12 GB</td>
<td>07:22:05</td>
</tr>
<tr>
<td>Szymanski(5),P4</td>
<td>(OOS)</td>
<td>45:52:25</td>
<td>38 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invalid Properties</th>
<th>IO-EI</th>
<th>IO-OWCTY</th>
<th>IO-MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson(5),P3</td>
<td>00:00:17</td>
<td>50 MB</td>
<td>07:14:23</td>
</tr>
<tr>
<td>Bakery(5,5),P3</td>
<td>00:25:59</td>
<td>5.4 GB</td>
<td>68:23:34</td>
</tr>
<tr>
<td>Szymanski(4),P2</td>
<td>00:00:50</td>
<td>203 MB</td>
<td>00:20:07</td>
</tr>
<tr>
<td>Elevator2(7),P5</td>
<td>00:01:02</td>
<td>130 MB</td>
<td>00:00:25</td>
</tr>
</tbody>
</table>

5 Squeeze the Juice Out of Your Parallel Computer!

Parallel computers in principle come in two flavours. As shared-memory, in which multiple computing units share a single global large piece of memory, and as distributed-memory, in which every computing unit is equipped with its own local (not necessarily small) amount of memory.

Until recently, improvements in hardware architecture have been providing verification tools with performance increases mostly for free – without any need for implementational or algorithmic changes in the tools. However, this trend appears to be diminishing in favour of increasing parallelism in the system – which is nowadays much cheaper and easier to implement than it is to build computers with even faster sequential operation. However, this architectural shift means that it is no longer possible to benefit from hardware progress, without introducing algorithmic changes to our tools. This is what we are striving for – providing algorithms able to exploit such parallel architectures and offering an implementation that can be deployed in practical situations.

To gain the maximum of a parallel computer we have to use suitable parallel algorithms. Despite rare situations, where the problem is embarrassingly parallel, we need to do more than just adopt the best sequential algorithm to a parallel machine. As for LTL model checking, both the Nested DFS and the Tarjan’s algorithm are inconvenient for parallel systems as they essentially rely on a depth first search postorder of vertices. Unfortunately, no optimal scalable technique
is known that would allow to compute the postorder while not eliminating the parallel processing at the same time. As a result, theoretically unoptimal, but parallel and scalable algorithms are used in practice. This is because despite the purely theoretical asymptotic worst case complexity, there are many other, often more practical, aspects that render an algorithm suitable for solving the LTL model checking problem. For example, both the algorithms presented in Section 3 can be of use.

While these parallel algorithms perform single parallel computation on a bunch of computation units that communicate intensively to achieve their common goal, a different approach might be to decompose the problem into subproblems and solve these subproblems independently combining just the individual results at the end of parallel computation.

This type of parallel processing was considered also for enumerative LTL model checking [1]. In the automata-based approach to LTL model checking the product automaton originates from synchronous product of the property and system automata. Hence, vertices are ordered pairs. An interesting observation is that every cycle in a product automaton graph emerges from cycles in the system and the property automaton graphs. As the property automaton is typically quite small, it can be pre-analysed. In particular, it is possible to identify all strongly connected components of the property automaton graph. The decomposition of the property automaton can be then used to identify independent subgraphs of the graph to be searched for the presence of an accepting cycle. If a part of the product automaton graph respects the decomposition of the property automaton graph into strongly connected components, no cycle can cross the boundaries of the part. If every part is processed by a single computation node only, multiple Nested DFS algorithms localized to individual parts of the graph may be employed to detect the presence of an accepting cycle in the whole graph.

Another interesting information can be drawn from the property automaton graph decomposition. Maximal strongly connected components can be classified into three categories [34]:

**Type F:** *(Fully Accepting)* Any cycle within the component contains at least one accepting vertex. (There is no non-accepting cycle within the component.)

**Type P:** *(Partially Accepting)* There is at least one accepting cycle and one non-accepting cycle within the component.

**Type N:** *(Non-Accepting)* There is no accepting cycle within the component.

Realising that a vertex of a product automaton graph is accepting only if the corresponding vertex in the property automaton graph is accepting it is possible to characterise types of strongly connected components of product automaton graph according to types of components in the property automaton graph. Classification of components into types $N$, $F$, and $P$ is useful for parallel processing as dedicated algorithms may be applied to process parts of the graph of different types. In [15] it was shown that the OWCTY algorithm is in fact optimal in
the case the graph of property automaton contains only components of type \( N \) and \( F \). Moreover, the same authors claimed that most LTL properties that are verified in practice are of this type [16].

A different example of dividing the given task into independent subtasks is nicely demonstrated on the parallel version of the MAP algorithm. It can be easily seen that an accepting cycle can be formed from vertices with the same maximal accepting predecessor only. A graph induced by the set of vertices having the same maximal accepting predecessor is called predecessor subgraph. It is clear that every strongly connected component (hence every cycle) in the graph is completely included in one of the predecessor subgraphs. Therefore, after applying the deleting transformation, the new map function can be computed separately and independently for every predecessor subgraph. This allows for speeding up the computation (values are not propagated to vertices in different subgraphs) and for an efficient parallelisation of the computation.

**Shared-Memory LTL Model Checking**

Both platforms have been considered in the context of model checking. Much of the extensive research on the parallelization of model checking algorithms for followed initially the distributed-memory programming model and the algorithms were parallelized for networks of workstations, largely due to easy access to networks of workstations. However, it is the shared-memory environment that is paid more attention in the last several years. This is because multicore CPUs made it easily available to general public.

It is in principle possible to take a distributed-memory parallel algorithm and simply run it on a multi-core machine as it is (which is possible, e.g. due to multi-core implementation of MPI). The result will be disappointing however. In Fig. 3 this is demonstrated on the bad scaling of the OWCTY algorithm. Adaptations taking into account specifics of the shared-memory architecture must be taken into account. Of course the very first attempt is to fully exploit the shared-memory for sharing data among individual threads. It is a little bit surprising that using shared hash table to avoid contention does not necessarily lead to better performance. For LTL model-checking on multi-core machines it seems to be best to use partitioned hash-tables (similarly to distributed-memory) which give better cache locality and to tune the parallel algorithm using techniques like efficient concurrent memory allocation and deallocation or lock-free and wait-free data structures for interactions [7].

The pioneering work is [28], where Holzmann and Bosnacki proposed an extension of the SPIN model checker for dual-core machines. Proposed algorithms keep their linear time complexity and the liveness checking algorithm supports full LTL. The algorithm for checking safety properties is capable of running on an arbitrary number of CPU cores, whereas the liveness checking algorithm, which is based on the original SPIN’s nested DFS, is limited to dual-core systems.

The paper [4] was the first one on multi-core LTL model-checking, focused on bringing distributed-memory algorithms to shared memory. It has discussed implementation techniques that have allowed to improve scalability of those
Fig. 3. Scalability of “MPI” OWCTY running on a multi-core machine and the tuned MT-OWCTY version.

A different approach to shared-memory model checking has been presented in [29], based on CTL* translation to Hesitant Alternating Automata. The proposed algorithm uses so-called non-emptiness game for deciding validity of the original formula and is therefore largely unrelated to the algorithms based on fair-cycle detection.

Distributed Memory LTL Model Checking

The standard parallel platform for distributed-memory computing is a network of workstations, a cluster for short. Cluster-based algorithms perform their com-
putation simultaneously on multiple computation nodes that are allowed to communicate and synchronise themselves by means of message passing. The advantage of this environment is not only the aggregate computation speed achieved by parallel processing, but also the aggregate amount of memory the platform can provide. Cluster-based algorithms proved their usefulness in verification of large-scale systems in many studies. They have been successfully applied to symbolic model checking [24, 25], analysis of stochastic [27] and timed [8] systems, equivalence checking [11] and other related problems [9, 12].

6 Conclusion

Platform-dependent verification is a newly emerging field. Extending the techniques as they are known from the sequential world adds significant complications and often requires entirely new approaches. We need to change our attitude in designing practical solutions for verification on the new hardware architectures. The key steps for their effective deployment in industry and real applications is to design appropriate algorithms, use algorithm engineering techniques and effects of the memory hierarchy, as well as implications of communication complexity, and heuristics. The new demand for platform-dependent verification algorithms that are of practical utility has raised the need to replace the traditional sequential approach.

Despite significant progress in platform-dependent verification that we have encountered during the last several years, practically useful platform-dependent verification tools are still to be developed. There are only a few of them, mostly of academic nature. In the area of LTL model-checking, the parallel distributed-memory verification tool DiVinE [3] has recently been released in its multi-core version – slightly ahead of the release of the multi-core version of SPIN. There are also many open questions and problems that naturally arise when we consider the new technological platform. An example is the following open problem: Is there a scalable parallel algorithm for accepting cycle detection whose sequential complexity is linear and the algorithm works on-the-fly?

References