Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 254 (2009) 45–63
www.elsevier.com/locate/entcs

Parallel and Distributed Invariant Checking of Microcontroller Software

Jörg Brauer Bastian Schlich Stefan Kowalewski

Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
Email: lastname@embedded.rwth-aachen.de

Abstract

Formal verification techniques are recognized as promising tools for the development of embedded systems. One such technique is invariant checking, which can be applied intuitively by developers as it does not require knowledge of temporal logics. State spaces for invariant checking are built using the same methods as used for model checking. They can become large due to the state-explosion problem. In [MC]SQUARE, which is a model checker for microcontroller programs, most of the time is spent building state spaces when checking programs. To improve the performance of [MC]SQUARE, we have implemented four parallel and one distributed algorithm for invariant checking. Parallel algorithms are especially helpful as they allow to fully utilize multi-core CPUs. This paper describes several parallel and distributed algorithms and presents a case study that compares these algorithms and shows the performance improvements achieved.

Keywords: Embedded Software, Verification, Assembly Code, Invariant Checking, Parallel and Distributed Algorithms

1 Introduction

Embedded systems are frequently used in safety-critical systems. Full testing of these systems is often not possible due to fast time-to-market, uncertain environments, and the complexity of the systems. Formal verification techniques such as model checking [7] or invariant checking [2] have been recognized as promising tools for the analysis of such systems. An advantage of invariant checking is that invariants are easy to specify by developers as they do not require knowledge of temporal logics. Furthermore, counterexamples generated by invariant checking are easy to understand because they are single successor

doi:10.1016/j.entcs.2009.09.059

chains that lead to the error state. These traces can be easily followed by the developer. Checking invariants is linear in the size of the state space, which makes the analysis applicable even to larger programs.

We have developed a model checker for microcontroller assembly code called [MC]SQUARE [21], which can also verify invariants. [MC]SQUARE works directly on the assembly code of the program and automatically applies abstraction techniques such as delayed nondeterminism [20] to tackle the state-explosion problem [6]. In contrast to other model checkers, tailored simulators are used to build state spaces. Despite the application of numerous abstraction techniques, state spaces for real-life programs easily grow vastly and consist of billions of states. It turns out that during the verification process, building state spaces is the bottleneck in terms of computation time and memory requirements. Multi-core processors have become a de-facto standard for computers, and high-bandwidth network connections are available, which can be utilized to make state space building more efficient.

This paper describes our experiences with implementing a number of different parallel and distributed algorithms for building state spaces for invariant checking. With minor modifications, these algorithms can also be used for model checking. Using parallel state space building allows to fully utilize multi-core processors, while distributed state space building reduces the requirements in terms of computational power and memory for state space generation on a single system by spreading the workload among a larger number of systems. We describe how a sequential algorithm for state space building can be extended for parallel and distributed settings using small modifications. A number of algorithms for parallel and distributed state space generation, which differ in the data structures used for storing states and communication, are described and evaluated with respect to their performance.

After an introduction of [MC]SQUARE in Sect. 2, which includes a description of the sequential algorithm on which the presented extensions are based, Sect. 3 presents four parallel algorithms. Section 4 presents an algorithm for distributed state space building. The efficiency of these algorithms is demonstrated in a case study described in Sect. 5. Related work is presented in Sect. 6.

2 [mc]square

[MC]SQUARE is a CTL model checker for microcontroller assembly code. It can verify code for four different microcontrollers, namely the ATMEL ATmega16, the ATMEL ATmega128, the Infineon XC167, and the Intel MCS-51. In contrast to other model checkers, [MC]SQUARE works directly on the assembly

program and automatically applies abstraction techniques such as delayed nondeterminism [20].

2.1 Overview

[MC]SQUARE uses explicit-state model checking algorithms, but the states – called lazy states – are partly symbolic. Lazy states, which are induced by symbolic representations of the microcontroller memory, do not represent single states but sets of states. In [MC]SQUARE, we have modelled different memory abstractions that vary in the degree of abstraction. Figure 1 shows the verification process applied by [MC]SQUARE. First, the binary code in ELF format, the C code if available, and the CTL formula are parsed and transformed into internal representations. Model checking is applied on the assembly code only, but debug information in the ELF file allows to use propositions about C variables in the specification. Then, static analyses are executed, which are used to annotate the program using information from the assembly code, the debug information, and the CTL formula. These annotations are used to reduce state spaces during model checking.

In the next step, [MC]SQUARE conducts invariant checking or model checking. Apart from the extensions presented in Sect. 3 and Sect. 4, we have implemented three different sequential algorithms: An invariant checking algorithm, an on-the-fly CTL model checking algorithm described by Heljanko [13], and a global CTL model checking algorithm based on the algorithm of Clarke et al. [7]. If successors of a state are not yet created, the state space uses a tailored simulator to create the needed successors on-the-fly. The simulator natively handles nondeterminism and creates an over-approximation of the behavior shown by the real microcontroller. Building successors is done by means of interpretation. A state is loaded into the model of the microcontroller, and then, all possible successors are created. A state can have more than one successor because interrupts can occur and input can be read from the environment or from devices with nondeterministic behavior such as timers. Section 2.2 details the algorithm for sequential state space generation.

In the last step, a counterexample is generated and also optimized. That is, loops and other unneeded parts are removed to ease its comprehension. The counterexample is presented in the assembly code, the C code, the control flow graph of the assembly code, and as a state space graph.

2.2 Sequential State Space Generation

In [MC]SQUARE, state spaces are built using tailored simulators. The implemented sequential state space building algorithm requires two data structures,

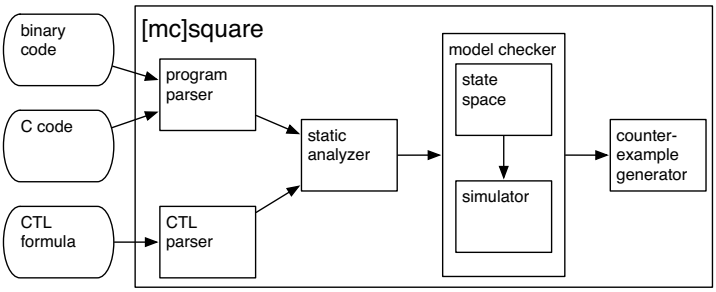

Fig. 1. Model checking process applied in [MC]SQUARE.

namely a state space S implemented by means of a hash map and a container U containing all unprocessed states. Given an initial state s_0, we initially set $S = U = \{s_0\}$. The generation of the state space is executed in a loop, where the following steps are performed while U is non-empty:

(i) A state $s \in U$ is removed from U.

(ii) All successors of s are generated by the simulator.

(iii) For each successor s' of s, an transition (s, s') is added to the state space. If s' is not already contained in S, s' is stored in U and in the state space S.

The search strategy depends on the implementation of the data structure U. If U implements a stack, then a depth-first search is performed, while a queue corresponds to a breadth-first search.

3 Parallel Invariant Checking

We have developed a template for parallel algorithms that extends the sequential algorithm. The template is used for the implementation of four parallel algorithms described. Parallel algorithms for state space building generally differ in two aspects, namely *load balancing* and *partitioning*:

- Partitioning describes both the structure of the state space and the mapping between a state and a state space in presence of multiple state spaces.
- Load balancing distributes the processing of states among the set of threads. Using static load balancing, the assignment of a state to a thread only depends on the state itself, while dynamic load balancing takes the workload of the threads at runtime into account.

First, this section describes the parallel template and different approaches to loading balancing and partitioning. Then, detection of termination in parallel environments is described, before four parallel algorithms implemented in [MC]SQUARE are detailed. Finally, the generation of counterexamples in

the parallel case is explained.

3.1 Parallel Template

This section describes an algorithm template used to implement the different algorithms for generating state spaces of programs in parallel. In the sequential case, one invariant checker directly accesses the state space and the simulator. In the parallel case, however, this is different.

Building a state space in parallel using `n` threads, numbered from 1 to `n`, works as follows. A globally accessible state space is used, but in contrast to the sequential case, each thread `i` possesses its own queue of unprocessed states. Initially, the state space contains only the initial state $\mathtt{s}_0$ of the program. The iteration is started by thread 1, which processes $\mathtt{s}_0$. The following steps are executed in a loop in each thread `i`:

(i) Thread `i` obtains a new state `s` from the queue of unprocessed states. Then, all successors s' of `s` are created using a simulator. Finally, for each successor `s'` of `s`, a transition `(s,s')` is added to the state space. If `s'` is not yet contained in the state space, it is stored and added to the queue of unprocessed states.

(ii) A test for termination is executed as described in Sect. 3.1.3. If no thread has a state to be processed in its local data structures, all threads terminate.

The algorithms described in the remainder of this section differ in the implementation of the functions used to access the state space. In our algorithms, two different approaches to load balancing and one partitioning function are used. Moreover, communication between threads is implemented using different data structures and synchronization primitives.

3.1.1 Partitioning of the State Space

In shared memory architectures, either a single state space or multiple state spaces are used to store states. A state space, however, is not necessarily assigned to a specific thread in general. In case of multiple state spaces, each state has to be mapped to a state space. This is performed using partitioning functions based on hash values. We have implemented a static partitioning function $part : S \to \mathbb{N}$ defined as $part(s) \mapsto f(s) \mod m$, which computes the index of the state space in dependency of the hash value $f(s)$ of a state s and the total number of state spaces m. If the hash function is uniformly distributed, states are uniformly distributed.

We have only implemented static partitioning because when using dynamic

partitioning some problems arise. If a thread performs a lookup for a state s, which is stored in state space i using dynamic partitioning, in order to detect revisits, all state spaces have to be visited in the worst case because the configuration that lead to the assignment of s to state space i is unknown.

3.1.2 Load Balancing

Load balancing describes the distribution of states among threads. Efficient load balancing algorithms minimize synchronization efforts between threads and evenly distribute the workload among all threads.

Static Load Balancing

For static load balancing, only the structure of the processed state has an influence on the assignment of a state to a thread. Stern and Dill [22] propose using an evenly distributed hash function. The static load balancing function implemented in [MC]SQUARE is the static partitioning function described in Sect. 3.1.1.

A different approach to static load balancing is described by Lerda and Sisto [19]. In their approach, the load balancing function is not based on the complete state but only on small parts. The idea behind this approach is that only small parts of a successor state are changed in a single transition. This approach increases the probability that successors of a state `s` are processed by the same thread as `s`. Developing an evenly distributed balancing function for this approach is challenging and sometimes impossible.

Dynamic Load Balancing

Dynamic load balancing decides at runtime which states are assigned to which threads. Here, we distinguish two approaches to dynamic load balancing:

- Synchronous: All threads proceed in two steps. First, all states stored in their local data structures are processed, and then, communication between threads is performed.
- Asynchronous: There is no strict separation between processing and communication. Threads that have finished processing their states do not wait for other threads to finish their computations. In general, this approach can be divided into sender-based and receiver-based algorithms. In the first kind of algorithms, highly utilized threads pass states to threads with lower workload. In contrast, threads with a low utilization request states from threads with high utilizations in receiver-based algorithms.

3.1.3 Termination

During parallel state space generation, the termination of all involved threads has to be detected. Two cases for termination are possible: Either no thread contains a state to be processed in its local data structures, or state space generation is aborted early, for instance, through user interaction or if no more memory is available.

We have implemented two algorithms for termination detection in [MC]-SQUARE. The first algorithm is based on Dijkstra's termination algorithm [8], which is also used by Inggs and Barringer [17] as well as by Holzmann and Bosnacki [14]. The second algorithm uses cyclic barriers available in the JAVA programming language. In practice, the implementation using cyclic barriers turned out to be faster than our implementation of Dijkstra's algorithm.

3.2 Implemented Algorithms

This section describes the implementation of four different parallel algorithms. For the implementation, we have evaluated the performance of the following JAVA containers for storing states: `HashMap` and `TreeMap`, accessed using explicit synchronization, as well as `ConcurrentHashMap` and `ConcurrentSkipListMap`. We observed results similar to the experiments of Goetz et al. [11], that is, `ConcurrentHashMap` being the fastest solution for parallel access.

3.2.1 Dynamic Load Balancing and Access to State Space using Locks

We implemented the receiver-based algorithm by Inggs and Barringer [17] for dynamic load balancing. In this algorithm, threads are logically ordered as a ring. A thread that does not have any states to proceed requests states from its successors in the ring. Synchronization efforts can be minimized by storing self-created states up to a certain size in a local stack in case the shared data structures are full. The states in this local data structure are processed with priority, and shared data structures, which require synchronization, are processed only if the local stack is empty. Each thread requires the following data structures: a partitioned state space, a queue of unprocessed states, and a local stack for local states.

While the state space and the unprocessed states can be accessed by all threads and hence require synchronization, the local stack is used to reduce synchronization overhead. In our implementation, mutual exclusion for the shared data structures is ensured using locks. States are generated similarly to the sequential algorithm described in Sect. 2.2, but states are first taken

from the local stack and then from the shared queue. The communication structure for two parallel threads is depicted in Fig. 2.

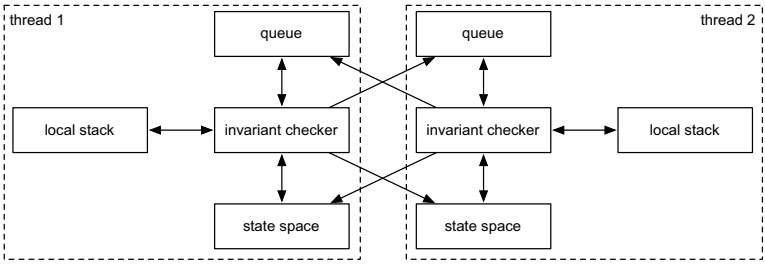

Fig. 2. Structure of inter-thread communication for dynamic load balancing using locks.

The thread that processes a state is determined using the static partitioning function $part : S \to \mathbb{N}$. If a successor state s' of a state s needs to be processed, thread $part(s)$ first attempts to store s' in the queue of unprocessed states of the respective thread $part(s')$. If this queue is full, however, then s' is pushed in the local stack of the current thread. The state, however, is stored in the state space of thread $part(s')$ even though it is processed by thread $part(s)$.

The size n of the shared queue strongly influences the performance of load balancing. With small n, many threads compete with one another for states to be processed, which limits parallel computations, while with large n, more computation time is required to fill the queues and the synchronization overhead increases. During experiments, we found out that the number of parallel threads turned out to be a good choice for n.

3.2.2 Static Load Balancing and Access to State Space using Locks

The second algorithm implements parallel state space building similar to the algorithm described before. The main difference is that static load balancing is used, and hence, no local stack is required to adjust the utilization of a thread. That means, a successor state s' is always stored in the state space of thread $part(s')$ and is then processed by this thread. Operations on the shared queue and the state space are synchronized using locks. Apart from the absence of a local stack in this approach, the overall architecture is similar to the one presented in Fig. 2.

3.2.3 Static Load Balancing and Access to State Space using Master Thread

In this approach, one master thread is exclusively used to access the state space in addition to the regular invariant checker threads. That is, the master thread is used to store states in the state space and to perform static load

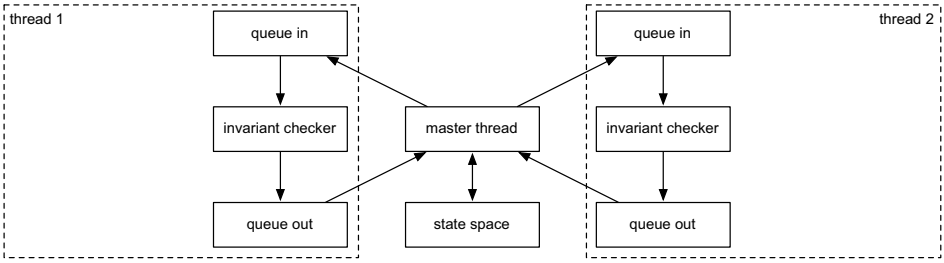

Fig. 3. Structure of inter-thread communication for static load balancing using a master thread.

balancing. This structure corresponds to a bidirectional producer-consumer pattern. Successor states are generated by n threads, which are consumed by the master thread and stored in the state space. The master thread uses the partitioning function $part(s)$ to determine the thread that processes state s. That is, the master thread fills the queues of the invariant checkers.

The overall structure is depicted in Fig. 3. Communication between the master thread and the invariant checker threads is implemented using queues, which are synchronized using locks. Here, n queues are used for each direction and the master thread polls the incoming queues using a round-robin strategy. The invariant checker threads only obtain states through their incoming queues, which are filled by the master thread.

In contrast to the previous algorithms, no synchronization of the state space itself is required. The advantage of this approach is that the workload of processing a state is spread among different threads. While invariant checker threads build new states, the master thread only stores states in the state space and implements load balancing. That is, the state space is accessed sequentially. The performance of this algorithm compared to other approaches strongly depends on the number of threads used. This approach is especially suitable if synchronization of the state space is more costly than the actual operation on the state space, which is the case if a large number of threads tries to access the shared state space at the same time.

3.2.4 Static Load Balancing and Local Access to State Space

Using the previously discussed algorithms, either each thread or only a master thread accesses the state space. Using the algorithm described in this section, each thread is assigned a partition of the state space, which it manages exclusively. The structure of this approach is depicted in Fig. 4. To allow each thread to insert states into each part of the state space, shared queues for sending states to be processed and stored are used. The queue is protected using a lock. All threads can insert a successor state s' into the queue of thread $part(s')$. The thread $part(s')$ then obtains the state from its queue, stores it

in its local state space, and eventually processes the state. This approach performed best of all four parallel algorithms as detailed in Sect. 5.

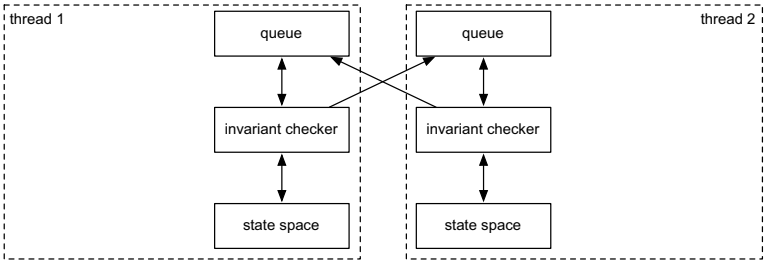

Fig. 4. Structure of inter-thread communication for static load balancing using queues.

3.3 Generation of Counterexamples

In the sequential case, counterexample generation is performed by means of a depth-first search in the state space. If new states are created, the corresponding transitions are stored in the order of creation. The counterexample generator then traverses the state space starting from the initial state visiting the last created states first. That is, the state space is searched diametrically to the sequence of creation because the last created state violates the invariant. In the parallel case, however, this procedure is not applicable because the insertion of states is not ordered due to thread interleavings. The lack of an order requires a complete depth-first search, which is currently performed by a single thread, but could be parallelized in the future.

4 Distributed Invariant Checking

This section describes an algorithm for distributed state space generation, which is based on the approaches of Stern and Dill [22], Lerda and Sisto [19], and Holzmann and Bosnacki [14]. We use the term *node* for each process in the distributed network. One master node is used, which starts the other nodes and detects termination of the distributed algorithm. It performs a different task than the master thread for parallel invariant checking.

4.1 Distributed Algorithm

In the distributed algorithm, each node runs three threads. The main thread executes invariant checking, that is, it has exclusive access to the state space, and performs load balancing. In our distributed approach, static load balancing and static partitioning are used as in the parallel case. Two threads, a sender and a receiver, are used for communication of states between nodes,

which is implemented using the JGroups library [1]. In [mc]square, nodes communicate using multicast-communication based on UDP. The communication threads and the invariant checking thread exchange states using queues, which are synchronized by locks.

4.2 Termination

Detection of termination is performed using the approach of Stern and Dill [22]. Each node i executes a test for termination and periodically sends a status message to the master node. This status message consists of the number of sent states s_i, the number of received states r_i, the number of states S_i in the communication queue of i, and the number of transitions T_i in the communication queue of i. Given the number of nodes n, then Φ is defined as:

$$\Phi = \sum_{i=1}^{n} s_i - r_i + S_i + T_i$$

If $\Phi = 0$, the master node sends a message to all nodes, which requires the regular nodes to send a reply. Once the master node has received all reply message, it recomputes Φ. If it is $\Phi = 0$, each node has to terminate, and otherwise, they are sent a message to continue their computations. This mechanism is used because of asynchronous communication between nodes. It allows to detect situations where all nodes have finished their computations, but messages are currently being sent from one node to another.

4.3 Generation of Counterexamples

Lerda and Sisto [19] propose using a stack for distributed counterexample generation similar to the sequential approach. This requires each node to know all visited states. This means, the complete trace from the initial state to the current state is sent to the node, which holds the successor state. In [mc]square, we abstain from generating counterexamples using a distributed algorithm. In contrast, the master node is responsible for counterexample generation and request states to be sent from their respective node similar to the parallel case.

5 Case Study

This section details a case study to evaluate the performance of all algorithms for parallel and distributed state space building described in the previous sec-

[1] http://www.jgroups.org/

tions. First, the evaluation principles used in this case study are described in Sect. 5.1. Then, Sect. 5.2 explains the influence of the state space structure on the performance of parallel and distributed algorithms. The analyzed programs are detailed in Sect. 5.3. Then, we present results for parallel algorithms in Sect. 5.4 and for the distributed algorithm in Sect. 5.5. All analyzed programs were written by students in lab courses, exercises, or diploma theses, and were previously used to evaluate the performance of [MC]SQUARE. None of the programs was written intentionally for use in this case study.

5.1 Evaluation Principles

The evaluation principles described in this section are based on Jones et al. [18]. The performance is evaluated with respect to the total speedup $S_a : \mathbb{N} \to \mathbb{R}$ and the efficiency $E : \mathbb{N} \to \mathbb{R}$. The total speedup is the ratio between the total runtime T_s using the sequential algorithm and the total runtime $T_p(n)$ of the parallel or distributed algorithm using n threads. The efficiency E is a measure for the utilization of the hardware used, defined as the ratio between the total speedup $S_a(n)$ and the number of threads n. It is always $E(n) \leq 1$, and an algorithm is optimal if $E(n) = 1$. The formulas are shown in Fig. 5. We have evaluated the runtime for verifying the invariant AG `true` as it leads to the creation of the complete state space. The computations required for evaluating the atomic proposition `true` are negligible.

$$S_a(n) = \frac{T_s}{T_p(n)} \qquad\qquad E(n) = \frac{S_a(n)}{n}$$

Fig. 5. Formulas for the total speedup $S_a(n)$ and the efficiency $E(n)$.

5.2 Structure of State Spaces

Ezekiel und Lüttgen [9] showed that the structure of state spaces has a significant influence on the performance of parallel algorithms, which is also valid for distributed algorithms. Three different situations are depicted in Fig. 6. The left-hand side depicts a state space that can barely be parallelized because in each stage, successors are generated only through exactly one state. The middle picture shows a situation that requires frequent communication between different threads. On the right-hand side, a state space is presented that can well be parallelized. Here, each thread can processes a large number of states and only few synchronization is required.

State spaces can be generated that allow optimal utilization of parallel and distributed algorithms. We believe, however, that such state spaces do not reflect the structure of state spaces of real programs and lead to overly optimistic results.

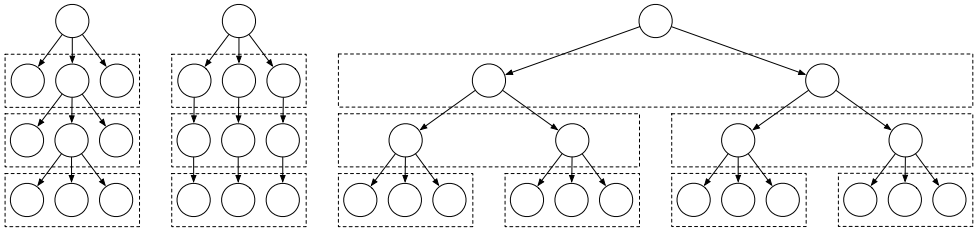

Fig. 6. Different structures of state spaces based on Ezekiel and Lüttgen [9].

5.3 Description of Analyzed Programs

We used two programs for the ATMEL ATmega16 microcontroller to evaluate the performance of the presented algorithms. The program `adc.elf` implements a distance measurement using an infrared controller. It consists of 467 lines of assembly code, which result in 434,756,686 states with all optimizations such as delayed nondetermism or dead variable reduction enabled in [MC]SQUARE. The program `window_lift.elf` implements a controller for a powered window lift used in a car and consists of 288 lines of assembly code. Without any optimizations, this program leads to creation of 2,589,681 states. We have chosen different abstraction techniques to uncover a potential influence on the performance of the parallel and distributed algorithms.

5.4 Evaluation of Parallel Algorithms

The system used was a SUN Fire X4600 M2 server with 8 dual-core AMD Opteron 8833 processors and 256 GB main memory. For all programs, an additional thread was used to perform garbage collection. Runtimes for the analyzed programs using 2, 4, 8, and 15 threads are presented in Tab. 1. Verifying the program `adc.elf` required a runtime of 5949.81 seconds using sequential state space generation. Using parallel algorithms and 8 threads, this was reduced to runtimes between 1682.70 seconds with static load balancing and a master thread and 1727.44 with static load balancing and local access. The speedup ranges from 3.444 to 3.536. That is, the overall results are very close. For `window_lift.elf`, the algorithm using dynamic load balancing performed worst. Verification of AG `true` using sequential state space building required 37.31 seconds. An absolute speedup of 2.596 in the best case and using static load balancing and local access was observed, while the algorithm using dynamic load balancing resulted in a speedup of 1.971.

More tests on different programs showed that the algorithm using static load balancing using queues described in Sect. 3.2.4 performs best of the implemented parallel algorithms on average. Comparing dynamic and static load balancing using locks, the static algorithm turned out to be faster be-

Table 1
Performance results for all parallel algorithms.

(a) Results of `adc.elf`.

	n	$T_p(n)$	$S_a(n)$	$E(n)$
dynamic+locks	2	3219.18	1.808	0.904
	4	2002.67	2.971	0.743
	8	1727.44	3.444	0.431
	15	1712.22	3.474	0.232
static+locks	2	3547.65	1.677	0.389
	4	1972.51	3.016	0.754
	8	1698.76	3.502	0.438
	15	1692.17	3.516	0.234
static+master	2	3526.57	1.687	0.843
	4	1940.16	3.067	0.767
	8	1682.70	3.536	0.442
	15	1685.05	3.531	0.235
static+local	2	3556.91	1.673	0.836
	4	1958.35	3.038	0.759
	8	1694.05	3.512	0.439
	15	1696.18	3.508	0.239

(b) Results of `window_lift.elf`.

	n	$T_p(n)$	$S_a(n)$	$E(n)$
dynamic+locks	2	26.33	1.411	0.706
	4	19.12	1.951	0.488
	8	18.93	1.971	0.246
	15	17.89	2.085	0.139
static+locks	2	22.49	1.411	0.829
	4	15.03	2.482	0.621
	8	14.71	2.536	0.317
	15	15.37	2.247	0.162
static+master	2	22.28	1.674	0.837
	4	15.13	2.466	0.616
	8	14.42	2.587	0.323
	15	15.09	2.472	0.165
static+local	2	23.34	1.598	0.799
	4	15.22	2.451	0.613
	8	14.37	2.596	0.324
	15	15.26	2.445	0.163

cause of the almost evenly distributed static hash function, which requires less runtime overhead and leads to almost uniform utilization of the available processors. Even though an absolute speedup of up to 7 was observed for certains programs, the overall improvement using parallel algorithms is disappointing. Using more than 5 threads did not pay off in most situations and results sometimes worsened using more than 10 threads.

We identified two reasons for these results. First of all, we believe it stems from the inefficient synchronization primitives in Java as it can be observed in all implemented parallel algorithms. Similar results were observed by Inggs [15] and Goetz et al. [11]. Another problem is the structure of our multi-core system. Although the server has a shared-memory architecture,

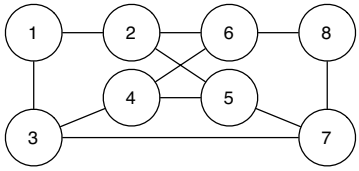

Fig. 7. Memory architecture of the Sun Fire X4600 M2.

each processor is directly connected to only 32 GB main memory. Other memory regions are accessed indirectly by tranferring data through other processors as depicted in Fig. 7. Our implementation does not take this memory topology into account in order to fully utilize the processing power. In order to optimize performance, the architecture of the system used has to be incorporated in the design of the algorithm and the implementation.

5.5 Evaluation of Distributed Algorithms

For this case study, a cluster of 9 Sun Blade servers was used, each of which is equipped with 2.33 Ghz Intel dual-core processors and 16 GB main memory. The results are depicted in Tab. 2. We observed an almost linear speedup for the program `adc.elf`. The verification time was reduced from 5422.32 seconds using the sequential algorithm to 671.82 seconds using the complete cluster. Similarly, the runtime for verifying `window_lift.elf` was reduced from 290.42 seconds to 45.79 seconds. Overall, the performance gained through distributed algorithms is much larger than in the parallel case. Due to the limitations of the available cluster, we could not evaluate the performance gained using a larger number of nodes.

Table 2
Performance results for distributed algorithm.

(a) Results of `adc.elf`

n	$T_p(n)$	$S_a(n)$	$E(n)$
2	2766.19	1.960	0.980
4	1448.01	3.744	0.936
6	981.25	5.526	0.691
9	671.82	8.071	0.538

(b) Results of `window_lift.elf`

n	$T_p(n)$	$S_a(n)$	$E(n)$
2	163.41	1.777	0.889
4	93.34	3.111	0.778
6	68.24	4.256	0.532
9	45.79	6.342	0.423

6 Related Work

Apart from the approaches described in this section, we have pinpointed other approaches in the respective sections. Stern and Dill [22] extended the model checker Murphi by a distributed algorithm for the verification of safety properties. In this approach, every process stores its assigned part of the state space and a queue of unprocessed states, where the assignment is based on a static partitioning function. Lerda and Sisto [19] developed an extension of SPIN for distributed LTL model checking. This approach is based on the work by Stern and Dill but uses a different static load balancing. While their approach does not speed up the verification process in SPIN, it allows the verification of larger models. The distributed approach of Garavel et al. [10] is only used for state space generation but not for verification of properties. The state space is built separately on different processes and then unified for verification by a single process. Inggs and Barringer [16,17] described a parallel algorithm, which uses dynamic load balancing and minimizes synchronization efforts using data structures similar to the ones we used (cp. Sect. 3). Other approaches, such as the ones by Barnat et al. [4,3], who extended the algorithm of Lerda and Sisto [19] by using data structures that allow a distributed depth-first search, are specifically suited for proving liveness properties in LTL.

Other techniques focus on state spaces for Petri nets and Markov processes. Caselli et al. [5] proposed a distributed algorithm for Petri nets. A parallel algorithm for stochastic Petri nets was described by Allmaier and Horten [1], while a distributed algorithm was developed by Haverkorth et al. [12].

7 Conclusion & Future Work

To evaluate the efficiency of parallel and distributed algorithms for state space building, we have implemented different algorithms in [mc]square. The extension of [mc]square with respect to the parallel algorithms required only small modifications of the architecture. The extensions needed for the distributed algorithm were more involved. We had to add a command line version of [mc]square to be able to deploy it to other nodes and used some existing libraries. In the end, we only kept the implementation of the best parallel algorithm, namely static load balancing and local access to the state space, and the distributed algorithm. Both algorithms can be tuned by users by adjusting the number of threads or nodes used.

In a case study that comprises some typical microcontroller programs, we have evaluated the performance of different parallel algorithms. The absolute speedup observed varies between a factor of 2 and 4 most of the time. Using

more than 5 processors barely paid off and sometimes even caused a slowdown due to the synchronization overhead. Comparable numbers were also observed by others such as Inggs [15] or Goetz et al. [11] when dealing with parallel Java programs. It could be the case that this observation is caused by the communication between the data structures used in Java. There are two possible solutions to this problem. First, we could implement the important methods in C or C++ and then use the Java Native Interface to use these methods. Another solution could be to use the new Java 7 version as it will include several performance improvements.

The algorithm now included in [mc]square for parallel invariant checking uses static load balancing. It performs best because the hash function distributes states almost evenly over all processors and the corresponding state spaces and requires few synchronization. Therefore, a dynamic distribution only required additional computations and did not show any improvements. An unexpected result of the distributed algorithm was that the performance improved up to a factor of 8 and the relative speedup was closely linear. In all situations, the speedup is larger than in the parallel case. We believe that the performance can be further improved by using other frameworks such as MPI or OpenMP for communication. This would, however, affect the portability of [mc]square. The distributed algorithm uses static load balancing as the best algorithm in the parallel case. In the future, we want to investigate combining parallel and distributed algorithms and evaluate the performance in high-performance clusters consisting of more than 9 nodes.

State spaces for invariant checking can also be used for model checking. For model checking, we plan to extend our global CTL model checking algorithm. We expect that this allows to use more than a single search front for state space building. In the local model checking algorithm, this is not efficient as it contradicts the local character of the algorithm.

Acknowledgement

This work has been supported partly by the UMIC Research Centre, RWTH Aachen University. Moreover, we thank Stefan Mau for many fruitful discussions and his efforts on the implementation of the algorithms described in this paper.

References

[1] Allmaier, S. C. and G. Horton, *Parallel shared-memory state-space exploration in stochastic modelling*, in: *4th International Symposium on Solving Irregularly Structured Problems in*

Parallel (IRREGULAR 1997), Paderborn, Germany, Lecture Notes in Computer Science **1253** (1997), pp. 207–218.

[2] Baier, C. and J.-P. Katoen, "Principles of Model Checking," The MIT Press, 2008, 936 pp.

[3] Barnat, J., L. Brim and P. Rockai, *Scalable multi-core LTL model-checking*, in: *14th International SPIN Workshop, Berlin, Germany*, Lecture Notes in Computer Science **4594** (2007), pp. 187–203.

[4] Barnat, J., L. Brim and J. Stríbrná, *Distributed LTL model-checking in SPIN*, in: *8th International SPIN Workshop, Toronto, Canada*, Lecture Notes in Computer Science **2057** (2001), pp. 200–216.

[5] Caselli, S., G. Conte and P. Marenzoni, *Parallel state space exploration for GPSN models*, in: *16th International Conference on Application and Theory of Petri Nets* (1995), pp. 181–200.

[6] Clarke, E. M., O. Grumberg, S. Jha, Y. Lu and H. Veith, *Progress on the state explosion problem in model checking*, in: *Informatics - 10 Years Back. 10 Years Ahead*, Lecture Notes in Computer Science **2000** (2001), pp. 176–194.

[7] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," The MIT Press, 1999.

[8] Dijkstra, E. W., W. H. J. Feijen and A. J. M. van Gasteren, *Derivation of a termination detection algorithm for distributed computations*, Information Processing Letters **16** (1983), pp. 217–219.

[9] Ezekiel, J. and G. Lüttgen, *Measuring and evaluating parallel state-space exploration algorithms*, in: *6th International Workshop on Parallel and Distributed Methods in Verification (PDMC 2007), Berlin, Germany*, Electronic Notes in Theoretical Computer Science **198** (2008), pp. 47–61.

[10] Garavel, H., R. Mateescu and I. Smarandache, *Parallel state space construction for model-checking*, in: *8th International SPIN Workshop on Model Checking of Software* (2001), pp. 217–234.

[11] Goetz, B., T. Peierls, J. Bloch, J. Bowbeer, D. Holmes and D. Lea, "Java Concurrency in Practice," Addison-Wesley, 2006.

[12] Haverkort, B., A. Bell and H. Bohnenkamp, *On the efficient sequential and distributed generation of very large markov chains from stochastic petri nets*, in: *8th International Workshop on Petri Nets and Performance Models (PNPM 1999), Zaragoza, Spain* (1999), pp. 12–21.

[13] Heljanko, K., *Model checking the branching time temporal logic CTL*, Research Report A45, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland (1997).

[14] Holzmann, G. and D. Bosnacki, *Multi-core model checking with SPIN*, in: *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, USA* (2007), pp. 1–8.

[15] Inggs, C. P., "Parallel Model Checking on Shared-Memory Multiprocessors," Dissertation, Department of Computer Science, University of Manchester (2004).

[16] Inggs, C. P. and H. Barringer, *Effective state exploration for model checking on a shared memory architecture*, in: *Workshop on Parallel and Distributed Model Checking (PDMC 2002)*, Electronic Notes in Theoretical Computer Science **68** (2002), pp. 605–620.

[17] Inggs, C. P. and H. Barringer, *CTL* model checking on a shared-memory architecture*, Electronic Notes in Theoretical Computer Science **128** (2005), pp. 127–123.

[18] Jones, M., E. Mercer, T. Bao, R. Kumar and P. Lamborn, *Benchmarking explicit state parallel model checkers*, in: *2nd International Workshop on Parallel and Distributed Model Checking (PDMC 2003), Boulder, USA*, Electronic Notes in Theoretical Computer Science **89** (2003), pp. 84–89.

[19] Lerda, F. and R. Sisto, *Distributed-memory model checking with SPIN*, in: *Proc. of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking* (1999), pp. 22–39.

[20] Noll, T. and B. Schlich, *Delayed nondeterminism in model checking embedded systems assembly code*, in: *Hardware and Software: Verification and Testing (HVC 2007), Haifa, Israel*, Lecture Notes in Computer Science **4899** (2008), pp. 185–201.

[21] Schlich, B., "Model Checking of Software for Microcontrollers," Dissertation, RWTH Aachen University, Aachen, Germany (2008).
URL http://aib.informatik.rwth-aachen.de/2008/2008-14.pdf

[22] Stern, U. and D. L. Dill, *Parallelizing the Murphi verifier*, in: *9th International Conference on Computer Aided Verification (CAV 1997)* (1997), pp. 256–278.