Universität Rostock

Traditio et Innovatio

Bachelorarbeit

Parallele Zustandsraumsuche

Vorgelegt von:

Tom Meyer

Matrikel-Nr.: 8200839

Eingereicht am:

02. März 2018

Betreuer:

Prof. Dr. rer. nat. habil. Karsten Wolf

apl. Prof. Dr. rer. nat. habil Van Bang Le

UNIVERSITÄT ROSTOCK | FAKULTÄT FÜR INFORMATIK UND ELEKTROTECHNIK
Lehrstuhl für Theoretische Informatik
Albert-Einstein-Straße 22, 18059 Rostock | Tel: +49 381 498 7641

Parallele Zustandsraumsuche

Mit dem LowLevelAnalyzer (LoLA) ist es durch Auswertung des Zustandsraums möglich, Eigenschaften von Petri-Netzen auszuwerten. Die dafür notwendige Suche wurde dabei bisher durch einen sequentiellen Algorithmus vorgenommen.

Im Hardwarebereich haben sich Prozessoren mit mehreren Kernen durchgesetzt. LoLAs sequentieller Ansatz kann deshalb oft das Rechenpotential moderner Prozessoren nicht ausschöpfen. Auch ein Versuch den vorhandenen Algorithmus zu parallelisieren brachte keine Performanceverbesserung.

In dieser Arbeit wird gezeigt das es möglich ist eine Performanceverbesserung durch Nutzung von mehreren Threads zu erzielen. Der vorherige Versuch wird analysiert und die Implementation so angepasst, dass eine Skalierung der Performance mit der Anzahl der genutzten Threads zu beobachten ist. Die Verbesserung wird an einem verbreiteten Petri-Netz gezeigt.

Parallel state space search

The LowLevelAnalyzer (LoLA) is a useful tool to analyze properties of a Petri net by expanding and evaluating its state space. To search for new states LoLA makes use of a sequential depth-first search algorithm.

However, recent hardware development introduced multi-core processors that cannot be fully exploited with LoLAs implementation. For this reason, the previously single-threaded search algorithm was adapted to use multiple threads. Unfortunately, the sequential algorithm always beats the parallel in the matter of execution time for a yet unknown reason.

In this work, it is shown that it is possible to utilize multiple threads for the state space expansion. The previous parallel implementation of LoLA is analyzed and improved to achieve performance scaling with the number of threads used. The improvement is shown using a common Petri net.

Betreuer:	Prof. Dr. rer. nat. habil. Karsten Wolf apl. Prof. Dr. rer. nat. habil Van Bang Le
Tag der Ausgabe:	13.10.2017
Tag der Abgabe:	02.03.2018

Contents

1 Introduction

The LowLevelAnalyzer (LoLA) is a model checking tool for Petri nets, but until today it utilizes only a single thread in productive use.

Petri net basics A Petri net is a mathematical construction to describe the behavior of distributed systems. Petri nets were first defined by Carl Adam Petri[1]. A Petri net is defined by a five-tuple: (P, T, F, W, m_0). It consists of **places** (P) that can hold infinite **tokens** (or marks). The initial amount of tokens on each place is defined by the initial **marking** (m_0). Places can be connected with transitions (T) via directed **edges** (F for flow). Places cannot be connected to other places and transitions cannot be connected to other transitions.

For example, net D in Figure 1.1 has four places and three transitions. The top place holds two tokens. Its **preset** is empty because there are no incoming edges and its **postset** contains only the top transition because the only outgoing edge points there. The top transition can be **fired** since all places in its preset (the top place) are marked with an equal or greater amount of tokens than the **weight** (W) of the connecting edges. Firing a transition will **consume** tokens from all places that point **to** the transition with a connecting edge. At the same time, it will **produce** tokens on all places where an edge points **from** this transition. This means that firing the top transition will consume one token from the top place and will produce one token on each center place (two tokens in total). After that, all transitions are **activated**, because the places in their preset are sufficiently marked.

Besides activated transitions, there are other properties[2] that can be checked in Petri nets. For example, net B in Figure 1.1 is **dead**. A net is dead if all its transitions are dead. And a transition is dead if there is no reachable marking in which it will be activated again. A Petri net might not be initially dead, but it may become dead if a particular sequence of transitions where fired.

Contrarily Net C is **alive** because all its transitions are alive. Its only transition is alive because it can always be fired in every possibly net state (there are other definitions of liveness[2]). However, a transition might neither be alive nor dead if it is possible to fire it in a reachable net state, but not in all possible net states.

These and other properties can soundly be formulated mathematically which makes Petri nets very useful for model checking.

LoLA LoLAs development started in 1998. It was developed to be used by third party tools to check properties of Petri nets[3]. Since then it was steadily updated to compete

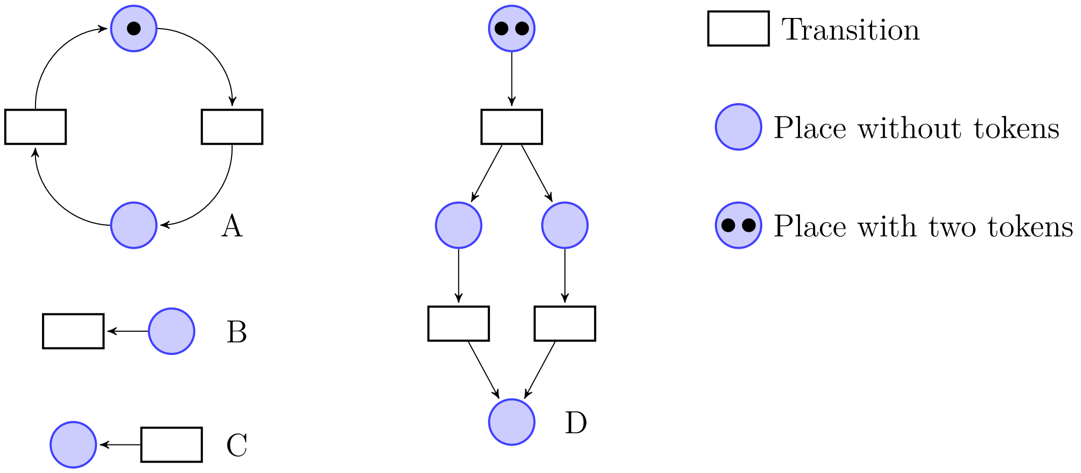


Figure 1.1: Petri net examples
Net A: Net can always fire one transition. One of the places will always have a token. If one has a token, the over will be unmarked.
Net B: The transition cannot be fired because the place has no tokens. If the place would have n tokens, the transition could fire exactly n times. After that, the pictured state would be reached.
Net C: Transition can always be fired since all preceding places (of which there are none) hold tokens. The place can have virtually infinite tokens.
Net D: Firing the top transition will consume one token from the top place and produce two tokens: one on the left place and one on the right place. At the final net state, the bottom place will hold four tokens and the others will hold none.

with other - state of the art - tools. LoLa was repeatedly decorated with prizes in a model checking contest with a focus on Petri nets [4]. The internal property evaluation, however, is still most performant single threaded. To evaluate a property of a Petri net, LoLA searches all necessary net states that can be reached from the initial one. The search is a depth-first search on a directed graph, which is discovered during the search itself.

For the specialized search in LoLA, a previous attempt of a parallel implementation exists. Unfortunately, the performance is usually about the same as the single threaded approach. But it can even get worse the more threads are used.

In this work, we will explore LoLA to find the part which is not performing as expected. The most important task is to find the bottleneck. We will measure the performance of LoLA so that we can compare the base version with an improved one that we will develop. The improved version should hopefully outperform the single-threaded algorithm with a reasonable amount of threads. Additionally, a desirable result would show a linear scaling of the performance with the number of threads.

2 Background and related work

Before we start, we will look at the background of this work. We will see an overview of parallel computing and some challenges that arise with parallel computation. We will investigate some related work in parallel search algorithms, and settle some basic terms of performance measurement. At the end of this chapter, we will look at some relevant implementation details of LoLA, and at the hardware that was used for this work.

2.1 Parallel computing

Until the first decade of the 21st century, the computing power of processors could be scaled with higher clock frequencies. However, this trend was replaced by the tendency to add multiple processing units on recent processors. For this reason, software that wants to exploit all processing capacities of current hardware has to adopt parallel computing paradigms.

Parallel computing can be described as the simultaneous execution of multiple (related) operations at the same time. There are different levels on which these operations can reside. Beside bit-level parallelism - where an arithmetic logical operation is executed on multiple bits (e.g. 32bit and 64bit architectures) and instruction-level parallelism[5] - where different processor actions (e.g loading memory, alu operations, writing memory) can be executed simultaneously, we will focus on the more abstract level of parallel tasks.

A task may be some arbitrary routine that can be processed on a processor. It can be thought of as a thread. Independent tasks can obviously be executed independent and therefore in parallel. But tasks that work as part of the same workflow or on the same data, might conflict with each other in some way. This is typically the case if we try to distribute an existing program to different subtasks that can be executed simultaneously. Some of these subtasks might depend on a result of a previous task or have to change globally shared data. This results in some problems that arise with parallel computing.

2.1.1 Limits in concurrency

There are limits to the potential of parallelizing tasks. Ideally, a program which is executed with n processing units or threads will terminate in $\frac{1}{n}$ of the time. But programs tend to have some parts that have to be executed sequentially. For example: to write to a file it has to be opened previously.

Amdahl already formulated in the 1960's that the speedup of a program is limited by its sequential parts[6]. This implies that thread scaling reaches its limits with relatively achievable amounts of threads. From his work a formula was extracted that is known as Amdahl's law.

Gustafson later reevaluated that formula[7] and points out that Amdahl assumes a fixed problem size. This is often true in academic research, but practical programs scale the size of the problem with the number of processors. He introduces a new formula with a variable problem size and a much more optimistic scaling (Gustafson paper gives a great view on both formulas and their meaning).

In both cases, it should be kept in mind, that a practical parallel program typically needs some kind of overhead to synchronize work that will prevent an optimal scaling.

2.1.2 Data integrity

Another major concern in the design of parallel applications is the data access. Often different parts of a program need to read or write the same information. Doing this step by step is trivial, but if two parallel executed tasks trying to access the same memory regions concurrently, data corruptions quickly strike the careless.

For example, let us assume two threads that need a list of data for a parallel search. The list shall be deleted after the search is finished. Now, if one thread searches the list for some data while the other just finished the search, the list might be freed by the second thread before the first thread is done reading. This will cause a segmentation violation since the memory is no longer accessible.

While concurrent reading is typically unproblematic, a single writing task can interfere with all other tasks. Not only can it cause a reader to return an incomplete written data segment, also accurately written data might be bound to a different program state, which does not fit to what the reader assumed at the time the read was scheduled.

If such behavior is not handled correctly, the corresponding memory segment depends on the first arriving task and are therefore called race conditions. A more detailed analysis of that term was done by Netzer et. al.[8].

To assure data integrity of shared memory, some kind of synchronization has to be done. This is typically achieved by blocking the access to the memory (especially operations involving writing) to one thread at a time. This necessity supports the development of some general synchronization concepts.

2.2 Synchronization concepts

Global memory with access restrictions can introduce bottlenecks which quickly degrades the whole program to sequential (or worse) speeds if done too frequently. One way to avoid that is to allocate thread local memory. This is generally advised for data that depends on a single thread anyway, however often there is some global memory that has to be shared between threads.

To keep this memory upright some rules to access it are required. An often used way is to simply block the access so that it can only be used by one thread at a time. We will look at some of such concepts here before we will come across them later:

A simple lock could be a flag with the two states `locked` and `unlocked`. Memory protected by such a lock would be accessible only if the flag is set to `unlocked`. Keeping the integrity of the states would be the duty of the implementor.

A more advanced version of such a lock was introduced by Dijkstra[9] and is called **semaphore**. Semaphores assume an amount of a resource that can be distributed. It has a value that can be incremented or decremented. If it is decremented to zero, the next task that is trying to access the resource has to wait until another task increments the semaphore again. For example, the set of keys of an apartment can be such a resource where the available count is a semaphore. If someone leaves the apartment he or she will want to take a key from the set to be able to come back again. By taking a key, the left amount is decreased. If there is no key left and somebody wants to leave the apartment (and come back later) this person has to wait until somebody else returns a key.

If there is only one resource available (a semaphore with the maximum value of one) the semaphore is equivalent to the previously described lock. Such a construct is typically referred to as **mutex**.

A locked resource met by a thread can force a thread to delay its work. If the thread cannot be suspended, some sort of waiting has to be implemented. A trivial algorithm of that kind just loops until the resource is freed and blocks the resource again immediately. Such a behavior is called **spinlock** because of its cycling behavior.

This rather trivial algorithm already introduces a quite concealed race condition. A possible chance exists that two threads examine an unlocked mutex and upon that, access the underlying resource before it could be locked by the contender. To resolve such problems, atomic operations were developed. Atomic operations are guaranteed to be executed completely, or not at all, without any possible remote influence. One of them is called **compare and swap**. It will compare a resource with an expected value and will change the resource on equality. Atomic operations are often implemented in state of the art hardware for maximum performance.

2.3 Depth-First search parallelization

LoLA uses a kind of depth-first search over a Petri nets possible (reachable) states, to evaluate a predicate. A depth-first search discovers graphs[10, chapter 1] by traversing the first possible edge of the current vertex. If no unknown edge is left for the current vertex, backtracking[11] is used, to find a previously discovered vertex with untraversed edges. The graph is completely discovered if no traversable edges are left.

As stated before, Petri nets are analyzed by LoLA, by searching for a set of net states that satisfy a given property. This means that one of the primary (possibly the single important) computation of LoLA is the depth-first search (besides some optimizations like net reduction techniques). A search that scales with the number of threads might imply LoLA scales with the number of threads. For this reason, utilizing a parallel depth-first search is much desired.

However, parallelizing depth-first search is not trivial. Freeman summarizes that it, in fact, was long thought to be inherently sequential[12]. He gives an overview of the breakthroughs and history of depth-first search, and directly reviews the history of parallel algorithms. Those often exploit a specific trait of the given problem. For example, Smith gave an algorithm that works for planar graphs[13] and Roa et. al. gave an algorithm for disjoint graphs[14]. Another algorithm form Aggarwal et. al. exists that works on general directed graphs[15], but it assumes that the complete graph is already known before the search is initiated.

LoLAs sequential approach is based on ”strongly connected components (SCC)” introduced by Tarjan[16]. His algorithm has the great benefit that the graph to search - can be discovered - during the search. This not only gives the chance for an early abort once a property is satisfied, without loading the complete graph. It also allows the search on possibly infinite graphs. Losing these qualities would significantly limit LoLAs usability.

Others value the same convenient properties and so Bloemen et. al. recently provided an algorithm based on Tarjan’s SCC’s that preserve them[17].

2.4 Performance measurement

Comparing the performance of different implementations - like we want to do later, can be a bit more complicated as it might appear initially. Just like that, what is called a performant application is highly dependent on its environment.

2.4.1 A distinction between different views on software

A common characteristic is the time it takes to execute a program or a subroutine. But in a lot of scenarios, the time consumption actually does not matter that much. A user might not even notice a factor of 1000 between routines that operate in an order of microseconds. At the same time, he might worry that the program can access enough memory to finish its task. That means before measuring the performance of a program or subroutine, one must define the characteristics that define the performance of exactly that given program or subroutine. Other characteristics besides execution time and memory consumption can be network latency, energy consumption or responsibility. Ultimately it is everything that increases the usability of a program.

After knowing what will be measured, the next concern is how it will be measured.

The act of measuring is typically called benchmark and the approach here is influenced by the scope of the measurement. The java community describes a general distinction between macro-, meso- and micro-benchmarks[18]. Each type addresses a different scope of a program.

Macro-benchmarks give an overview of the entire execution. The data collection is often less complicated since it is done over the whole application runtime and hardly requires any modifications to it. Measuring the runtime of a program is as simple as it gets: taking the time difference between the end and the start of the program. Memory allocation can often also be measured with simple tools that the operating system or other third party software provide. The downside is that a lot of software cannot be measured at all in this scope. So it is a property of event-driven programs that they do not terminate at a fixed point in time and therefore the execution time is of limited use.

Micro-benchmarks address the most narrow aspects of an application. They measure the execution of single instructions or atomic routines. Results of micro benchmarks have to be taken with a grain of salt because compile and hardware optimizations strongly influence them. Assumptions that work for the single code snippet might become false after the compiler integrates that part into the complete program or vice versa. It is a good advise to use a benchmarking tool which is built for this task. Overwise there is a risk to be trapped in false results. Like compilers that are optimizing away whole loops (which in turn might be the benchmark itself), because the result can be evaluated at compile time. The advantages of micro benchmarks, however, is that different implementations of basic algorithms can be compared directly. Provided they are performing equivalent in the macro scope of the program.

After the previous scopes described a detailed view and an overview, the last scope has to be something in between. And this holds true for the so-called **meso-benchmark**. Perhaps the best explanation for this kind is, that a meso-benchmark is what is not macro nor micro-benchmark. Meso-benchmarks deal with parts of a program that gets more complex. Like Modules or further reaching subroutines. These can be the most interesting benchmarks since they can narrow down bottlenecks to specific parts of the software. Unfortunately, they often require manipulating the program in some way. Which again means that the measured performance differs from the actual performance.

2.4.2 Methods of measurement

For the actual measurements, two basic approaches can be distinguished: manual implementation or usage of a tool.

The quickest and most precise approach is to make own measurements. Self-written and therefore known software can be easily extended e.g. by counters for variables or measurements of time periods. All measurements can be tailored to the individual use case and results can be exported in the most convenient format. On the over hand, this approach can come with several downsides. Some of them might be that the maintenance of the measurements can get increasingly complex the more numerous they get. Added code will influence the performance of the software itself, and own mistakes might hide behind seemingly satisfying results. There are probably other reasons that can be thought of. But short and temporary measurements of this kind might give a quick overview other the own software.

Probably the safer approach is the usage of a tool. There exist a variety of tools which are build for the inherent purpose to measure application performance with common metrics and use cases. Since the metrics reflect some characteristics of an application - a profile - these tools are typically called **profilers**. Using established profilers for performance measurement can help avoiding common traps and mistakes especially when they were designed by experienced developers who are skilled in this area. But it also means to invest time and effort to learn how to use them and how to interpret their results.

Profilers aid automate processes that otherwise would have to be implemented manually. An often used method to figure out bottlenecks is to take samples of the call stack during program execution. The calls that are most frequently sampled, will correlate with the program parts, where the most time was spent in. Another method is to use code injections. Here, the code is inserted automatically into the source or binary code at relevant spots. The injections are then used to count events or measure time intervals.

With the general techniques in mind, we can proceed to the actual software and its characteristics that will be examined in this work.

2.5 Implementation details

We will not be able to improve LoLA, without knowing some basic implementation characteristics. Hence, we will look at some relevant parts of the code in this chapter.

2.5.1 Codebase

This work is based on a previous attempt by Gregor Behnke to use multiple threads for LoLA. For this reason, his code will be taken as a base implementation for the parallel search. In the LoLA project structure, this would correspond to the `ParallelExploration` class in src/Exploration.

This class itself seems to be based on a single-threaded algorithm in the `DFSExploration` class, since the basic structure and several lines of code, as well as comments, are equivalent. The base algorithm was then altered by Behnke to make data manipulation thread safe. This implies that no specialized depth-first search algorithm was used, but the existing algorithm was extended.

The purpose of the exploration class family in LoLA is to keep track of the **paths** that were discovered during the search. The actual net **states** that are the vertices of the path are managed by another family of classes: the store classes. In more detail, this means that the exploration class will store the edges that lead to the current net state and chooses which edges of the graph will be used next to discover another net state. The discovered net states will be handed to the store class. It will tell the exploration class if this net state was already discovered earlier. If that is not the case the net state will be stored permanently

Behnke tried to implement multithreading (inside the exploration class) by exploring a different path with each thread. He, therefore, introduced thread local variables that keep track of the path, determine which edge will be expanded next and what the current net state is. The store is globally shared. Thread safe searching and adding to the store must be implemented by the store itself for this approach. The crucial work of Behnke inside the exploration algorithm was to distribute the work over all threads and keep the data synchronized.

Load distribution is done with a simple approach. If a thread can expand multiple edges and an idle thread exists, one edge will be discovered by the current thread and another will be discovered by the next possible idle thread. The data will be synchronized, so that the woken up thread will copy the previously discovered path of the waken. Cases like: what should be done if a thread has no more work (accessible edges) left or what to do if the current net state satisfies the asked property are also handled.

The store which is used by LoLA in the default case (`PluginStore`) does not provide any thread safety. The thread number is completely ignored within the relevant parts and so the behavior of this store when using multiple threads is undefined. But there is another store implementation which implements several buckets to store the states in: the `HashingWrapperStore`. The buckets should only be accessible by at most one thread. The net states that should be stored are now given a hash value and every bucket has a range of hash values associated with them. With this solution multiple searches can be issued, as long as the hash value of the searched net states differ.

The actual classes which are used at runtime to expand the search tree, are determent by the call switches which are handed to the LoLA executable. The relevant switches for the parallel exploration are `--threads=[threadCount]` and `--bucketing=[bucketCount]`. A thread count greater than one will tell LoLA to use the `ParallelExploration` class and the bucketing switch will signal to use a hashing store.

2.5.2 Data synchronization

To keep global data consistent, a synchronization method is needed. A common way to achieve this, is to restrict the access to one thread at a time. For this, the concept of mutexes and semaphores are often used as locks for access control. The downside of this approach is a bottleneck at the global data. If access is granted to at most one thread, every calculation that is done while accessing is also done at most with the speed that a single thread can provide. Therefore accessing these regions (including locking and unlocking them) should be done as infrequent and as quick as possible.

Since LoLA computes a lot of short and low-level instructions, locking and unlocking data segments might sum up to a significant amount of time. So the locking mechanism used by LoLA might also be a good starting point to search for a reason for that unexpected performance that LoLA shows.

Behnke used locks from the c++ pthread library. A possible substitute would be an own implementation with a compare and swap instruction as spinlock. This has the potential to keep the overhead for the locks at a bare minimum.

2.6 Environment

The used development environment consists of two machines. The general development is done on a virtual machine (VM). It runs with 4 threads on an Intel® Core™i7-4770S CPU @ 3.10GHz with 4 physical cores. Each core supports hyperthreading which makes a sum of 8 threads on the host of the VM. A total of 17.4 GB of physical memory is available for use inside the VM. Kubuntu 17.04 is running as operating system.

As a performant test system, a server (ebro) was used. Ebro has 4 AMD Opteron™6284 SE processors. Each with 16 cores and 2.7GHz base clock with a max boost of 3.4GHz[19]. This sums up to 64 physical cpu cores (and threads) in total. As physical memory, 995.2 GB are installed. The operating system is CentOS Linux 7 Core (collected with the `hostnamectl` command).

If no other source is provided the data was collected with the proc filesystem provided by the linux distributions.

Spec. Name	**VM**	**Ebro**
Max. Clock per Thread	3.1GHz	3.4GHz
Available Threads	4	64
Memory	17.4 GB	995.2 GB
Operating System	Kubuntu 17.04	CentOS 7 Core

Table 2.1: Specifications of the two development systems

3 Approach

After some insight in the underlying matter, we will now try to figure out where the actual bottleneck resides. We will try to change the underlying synchronization mechanism and see if the change has any effect. We will also rework LoLAs heap allocation strategy. At the end of the chapter, we will evaluate the results of different benchmarks.

3.1 Switching the synchronization

The first step taken was switching from the pthread library to a compare and swap (CAS) algorithm. Since C++ 11 there is an equivalent implementation in the standard library called `bool std::atomic::compare_exchange_weak(T& expected, T val)` (or `bool std::atomic::compare_exchange_strong(T& expected, T val)`) that was used for this task. This function compares the current value of an `std::atomic` with `expected` and replaces it with `val` if the comparison returns true. If it returns false it replaces `expected` with the actual value of the atomic. The weak version is allowed to return false in favor of a general performance gain, even if the compared values are actually equal.

To represent a lock that can either be locked or unlocked, a boolean as value is sufficient. To shape a spinlock like with a CAS function, they have to loop until the expected value compares equal. There are two reasonable modes to lock: the first is to just wait until an observed lock is unlocked and the second is to wait for an unlock with an immediate locking. The first approach can be helpful to suspend the execution of the current thread until an external thread is signalling a continuation. The second approach can be used to block access to a resource until all necessary manipulations are completed. The resulting implementation is shown in listing 3.5.

Swapping the old pthread implementation with the new CAS implementation now remains a matter of search and replace. The equivalent of the mutexes `pthread_mutex_lock()` is `waitAndLock()`. `Pthread_mutex_unlock()` corresponds to `unlock()`. The previous `sem_wait()` correspond to a `waitForUnlock()` call after a `lock()`. And `sem_post()` corresponds to `unlock()`.

However, this is a very specialized replacement which acts as a proof of concept. Other parts of the code might have to be replaced in another way, depending on the semantics of the part. Additionally, the CAS implementation is as short as possible. The pthread library comes with some important features like a mechanism to reduces the risk of deadlocks and different modes for the semaphores.

```
#include <atomic>

enum LOCK{
    LOCKED,
    UNLOCKED
};

static inline void waitAndLock(std::atomic<bool>* lock){
    bool expected = UNLOCKED;
    while (!lock->compare_exchange_weak(expected,LOCKED)){
        expected = UNLOCKED;
    }
}

static inline void waitForUnlock(std::atomic<bool>* lock){
    while (lock->load() != UNLOCKED){
        continue;
    }
}

static inline void lock(std::atomic<bool>* lock){
    lock->store(LOCKED);
}

static inline void unlock(std::atomic<bool>* lock){
    lock->store(UNLOCKED);
}
```

Listing 3.1: Basic implementation of a spinlock with compare and swap

3.2 Finding the bottleneck

We now have an application with a bottleneck and a possible solution to fix that bottleneck. Next, we will compare the performances.

3.2.1 Benchmark characteristics

In this case, performance means execution time. The expectation is that the execution time decreases with an increased amount of threads. The time is, therefore, the characteristic of interest.

As test systems, the machines described in 2.6 were used.

As test data, a predefined Petri net of the dining philosophers was used. It is a common concept in theoretical computer science to illustrate problems and risks of parallel processes and was originally introduced by Dijkstra[20]. The philosophers count can be scaled to an arbitrary amount. This is useful to increase the complexity (and therefore the execution time) of the net to a convenient level. The graph to search is also reasonably branched to allow a parallel discovery. The actually used net is a version with one thousand philosophers.

LoLA is executed with the `--threads=[threadCount]` and `--check=full` switches. The first switch simply sets the number of threads that should be used for the state exploration. The second switch cause LoLA to explore the whole state space without exploring a property. This ensures that the application will terminate after all states have been discovered and no varying discovery paths can influence the execution time.

3.2.2 A general performance survey

LoLA will spend most of the time inside the depth-first search of the state exploration. For this reason, a simple execution with the given parameters will uncover if the different synchronization implementations have an impact on the performance. This would correspond to the macro benchmark scope and can be achieved without any additional modification or tools, because LoLA already measures its own execution time.

Unfortunately, a simple test run on the VM reveals that there is no performance gain. Figure 3.1 shows that the new implementation is actually slower than the previous. A relevant speedup would divide the execution time by an order of the used threads. Taking this expectation as basis, both implementations still can be considered (roughly) in the same order of performance. The new implementation needs about twice the time, while a quarter of the time was expected. Thus, we can conclude that we either missed the cause of the bottleneck, or the new implementation faces similar challenges.

Since the benchmark result shows no positive change in performance, an exhaustive evaluation is dropped in favor of a detailed bottleneck analysis. In the next section, we will use a more precise benchmark method to inspect the search characteristics.

3.2.3 Manual benchmarking

To develop a deeper understanding of the internal processes, we have to examine the individual program parts. The most important one remains the search for net states which is done in the `ParallelExploration` class.

LoLA will call the `depth_first()` method of the `ParallelExploration` to initiate the search. There, all search threads will be initialized, started and destroyed (after they finished their work). Starting and destroying the threads takes constant time per thread. Since we will work with very few threads in relation to the number of states we will handle, these parts are insignificant for the performance measurement and can be ignored. But each thread will execute the `threadedExploration` method. This method is the algorithm for the actual search. It will loop until a given predicate is satisfied by any thread. For these reasons, we will focus our measurements on this method.

With this, we now know a precise code section that we want to inspect. Next, we have to choose a new measurement method. We can decide between two general approaches:

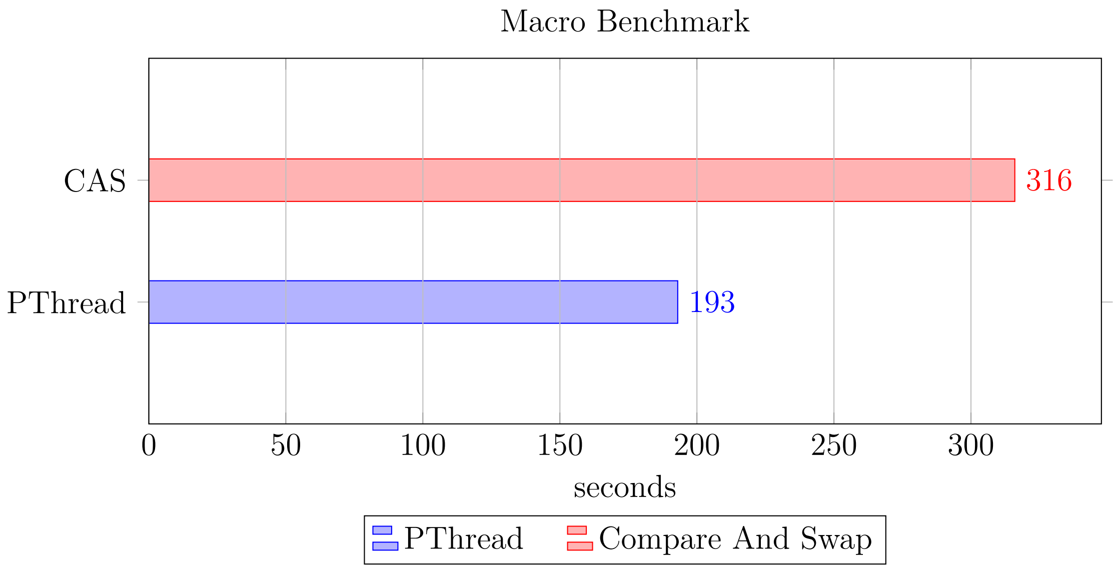


Figure 3.1: Macro benchmark comparison between the original (PThread) implementation and the substitute (Compare and Swap)
LoLA call: "lola –check=full –threads=4 ../tests/testfiles/phils1000.lola"

we can use an existing tool that can hopefully inspect the application parts we are interested in, or we can extend our code manually. Both approaches have their up and downsides. In this case, it was decided to use the manual approach. First, because the interesting code is relatively short and clear and second because it can be done right away, without spending much time on learning a third-party software.

In the next step, measurements have to be inserted into the relevant sections. However, what section is relevant cannot be known before measuring. We have to choose parts that seem to be most likely. This is an inherently subjective process, but some factors will be influential. For example, elemental assignments like `x = 5;` will take an insignificant amount of time, whereas function calls and loops can take arbitrarily long times to be executed. The cost of a bad selection is very low since the measurements can be easily changed.

The time was measured with the functions provided by `std::chrono` from the c++ standard library. The exact code is considered trivial and will not be discussed further.

Table 3.1 is listing the performance of the code pieces that are considered relevant. Other less important measurements are omitted to keep a clear view. Here is an explanation of the values:

- Total Thread Time - Cumulative time spent in each thread.
- Synchronization Time - Time spent to synchronize the threads with mutexes.
- Store Search Time - Time spent inside the `searchAndInsert` call. A method to store the discovered states.

- Idle Time - The Time each thread spent for the restartSemaphore to be unlocked
- Work - Time that each thread spent while having states left that can be discovered
- No Work - Time spent during waiting for new states that can be expanded including reinitialization. 'Work' and 'No Work' should cover the complete search loop.

	Thread 0	**Thread 1**	**Thread 2**	**Thread 3**
Total Thread Time	256s	256s	256s	256s
Synchronization Time	0.00000004s	0.00000004s	0.00000005s	0.00000005s
Store Search Time	58s	58s	56s	58s
Idle Time	0.22s	0.22s	0.22s	0.22s
Work	64s	64s	62s	64s
No Work	0.5s	0.5s	0.5s	0.5s

Table 3.1: Manual meso-benchmark of the ParallelExploration

3.2.4 Evaluating the first data

The benchmark results are quite inconsistent. On the one hand, they give some insight where the bottleneck might be. On the over hand, the values do not add up correctly.

Most problematic is that the time inside the search loop (sum of 'Work' and 'No Work') is not anywhere near the total time spent inside the search. This is very unexpected because almost all the time must be spent there. An unlisted measurement of the time before the loop also could not uncover a missing gap. This is a strong hint that this approach is not applicable in our scenario.

However, even though the results have to be taken with caution, we can take two hints from them. First, the time spent inside the mutexes seem to be insignificant. Whereas the time spent to store the discovered states seem to cause nearly all the work inside the loop. The corresponding method - searchAndInsert - is called by all threads directly on the globally shared store.

What store is used is decided at runtime. A trivial debug session can uncover that the PrefixTreeStore is used with the parameters we pass to the LoLA call. Although the thread id is passed to the method - a look to the signature:

```
bool PrefixTreeStore<T>::searchAndInsert(const vectordata_t *in,
    bitarrayindex_t bitlen, hash_t, T **payload, threadid_t, bool noinsert)
```

Listing 3.2: searchAndInsert signature

shows that the id is completely ignored (no variable name is defined). This means that the multi-threaded behavior of this class is completely undefined and it should not be used in this scenario. We have to change the way we call LoLA so that we can assure a thread safe behavior.

Fortunately, there is a way to use an appropriate store implementation. If LoLA is called with the additional parameter `--bucketing=[numberOfBuckets]` the `HashingWrapperStore` store is used instead of the `PrefixTreeStore`. The `HashingWrapperStore` divides the memory into different buckets and uses hashes of the discovered states to store them. Each bucket is used for a range of hashes and the access to a bucket is only granted to at most one thread at a time. With this implementation, multiple threads can store their discovered states at the same time, as long as the states hashes differ. Because of this insight, all future calls of LoLA will be done with the `--bucketing=[numberOfBuckets]` parameter.

Unfortunately, the new parameter has no relevant impact on the performance of LoLA. Additionally measuring further timings inside the `HashingWrapperStore` lead to even more inconsistencies. This caused a greater lack of trust and an overthinking of the benchmark method.

3.2.5 Reconsidering the approach

Our previous results point to a bottleneck inside the `searchAndInsert` method of the store. But due to inconsistencies between them, they seem unreliable.

The exact reason is unknown and to find the cause might be as difficult as the search for the bottleneck itself. But during the implementation of the benchmarks two problems arose that could be related:

First, it proved difficult in practice to measure exactly those periods of interest. Especially covering the whole implementation is quite error prone because the implementation is not linear. There are several branches, loops and early returns that have to be considered. As a result, one might start a time period more often than to stop it or vice versa. This quickly leads to wrong times without noticing it.

The second problem refers to the kind of time that is measured. Especially in multi-threaded environments a distinction between real-time (or wall-clock time) and cpu time is important. The real-time of a program is the time that passes for an observer during its execution while the cpu time is the time the cpu was active. This means that the cpu time can be actually lesser or greater than the real-time. For example, a program that is temporarily suspended for another process can have a shorter cpu time than real-time. Whereas a program that is executed on multiple cpus will have its cpu time also multiplied by the number of cores (or threads), resulting in a greater cpu time than real-time. In our scenario, blocking a thread with a mutex might influence the time periods that are taken. To measure correct values, a deep understanding of the matter is necessary and the approach should be well-thought-out.

Since the time for this work is limited and the author is a novice in the field of benchmarking, the manual approach is put aside for a more refined third-party solution.

3.2.6 Profiling tool evaluation

If a challenge gets to difficult to face it alone, getting help is something worth considering. In our case, the difficulties rose and the trust in the own approach depleted considerably. The use of a well-designed tool, now, seems to outweigh the effort that has to be spent to use one efficiently.

But before learning to use a profiling tool, we have to know which one we want to use. There are countless profilers to choose from, making an exhaustive evaluation impossible in the scope of this work. As such it was decided to select some candidates from an internet research, that are mentioned often in similar environments.

In the following, a short overview of the candidates shall be given with a reason that this candidate will or will not be used. This will be no detailed comparison with a conclusion why the choice is the best in our case. Instead, it should only give insight on why the choice was made.

3.2.6.1 Gprof

Gprof is a profiler that was developed in the 1980's. It generates a flat and a call graph profile[21]. In the flat profile is listed how often a routine was called and the cumulative time spent in it. The call graph lists which routine calls another and by which itself was called.

The profiler is integrated into the gcc compiler. Passing the `-pg` switch to gcc will add code to the compiled executable that is needed for profiling. A program that is compiled in this way can be executed normally, but during execution, it will gather data that is written to an output file later.

Reading the documentation will reveal an additional precondition: the program has to be closed with the `exit` function. In most cases, this is no problem since it will be called implicitly as soon as the `main` function exits normally. LoLA however, calls the `_exit` function explicitly. This saves a lot of time during the teardown of the program because the destructor of allocated objects is not called. Instead, the operating system just frees the allocated memory.

It is quite easy to change the behavior of LoLA to call the default `exit` function. But it causes LoLA to be unresponsive at the end of the execution for at least a very long time. And because its cleanup process was designed to use the 'dirty' approach, it is not guaranteed that it will return at all. Additionally an attempt with a modified LoLA executable and a simple Petri net produced an empty output file.

As a result, gprof is dropped as a candidate at this point to move to next candidate.

3.2.6.2 Valgrind

”Valgrind is a dynamic binary instrumentation framework designed for building heavyweight dynamic binary analysis tools”[22]. Dynamic binary analysis is used to ”analyze programs at run-time at the level of machine code”. The code that is needed for the analysis ”is added to the original code of the client program at run-time” which is what is called dynamic binary instrumentation.

This means that valgrind can inject code into existing applications and recompile the whole program. What code should be injected can be defined in a plugin for valgrind. A widely known plugin is ”memcheck”. With its help, inconsistencies of allocated memory can be discovered. Valgrind also publishes the ”callgrind” profiling plugin that is able to generate call graphs of an executed application.

The code injections and recompilation of valgrind have a great impact on the performance of the analyzed program. For example, the speed of an application analyzed with memcheck is reduced by a factor of 10 to 30 [23]. But an even more problematic detail for our scenario is, that parallel programs will be executed serially. Some problems and behaviors like race conditions will be obscured this way. And most important: the speedup that a parallel program can achieve over a sequential one cannot be observed. This makes valgrind inapplicable for our performance measurement.

3.2.6.3 Perf

Perf is a profiler that is directly integrated into the linux kernel. It uses hardware performance counters. These are unique registers on a processor, that count certain events like cache misses or branch mispredictions. This OS and hardware proximity, make the use of perf very cheap in execution time. The accumulated data, however, can get quite big very fast.

The documentation of perf leaves a lot to be desired, but the usage is reasonable straightforward and produces well-organized data. It is easily possible to trace bottlenecks down to the (with debug symbols annotated) disassembly. But interpreting the data demands some effort from the user. For example, one must know how to read the disassembly and how to find the corresponding sections in the source code.

Perf can be configured to count and measure innumerous events. Understanding the complete complexity would be way out of scope for this work. But with a bit of searching, guides can be found that introduce perf calls and explain what they are doing[24].

Testing and refining these examples gave some usable data that was used as a basis for further modifications to LoLA. In the next section, we see how perf was used and what data it collects.

3.2.6.4 Use of perf

The examples which were used to get to know perf uncovered a new possible bottleneck quickly. But before we look at the data, we should take a look at the used perf call and its effect. The actual measurement was done on the VM since special rights are needed for execution.

The basic perf command that is used to collect data is `perf record`. It can be executed with every possible bash command or delegated to a process id (pid) and is normally configured with an event that should be measured. We will only use the ”stack chain/backtrace” feature to generate a call-graph. With this option perf periodically takes (samples) stack traces which can be used to analyze which functions were called most, from which function they were called and which functions they were calling. The functions that are sampled most of the time correlate with the functions that are responsible for the most cpu load (if enough samples are taken). The corresponding perf switch is `--call-graph`. The switch itself has three modes which determine how the data is collected. We use the `dwarf` mode since the man page states that the others are either discouraged on gcc programs or require special hardware.

It is also possible to limit the perf call to a time interval with the `sleep n` parameter. It will cause perf to abort after `n` seconds. This comes in handy to reduce the execution time of LoLA and to limit the data size generated by perf (which can grow easily in the GB dimensions).

The final bash call that was used looks like this:

```
lola --check=full --threads=4 --bucketing=100 phils1000.lola&;
PID=$!;
sudo perf record -p $PID --call-graph dwarf sleep 30;
kill $PID
```

Listing 3.3: Profiling LoLA with perf

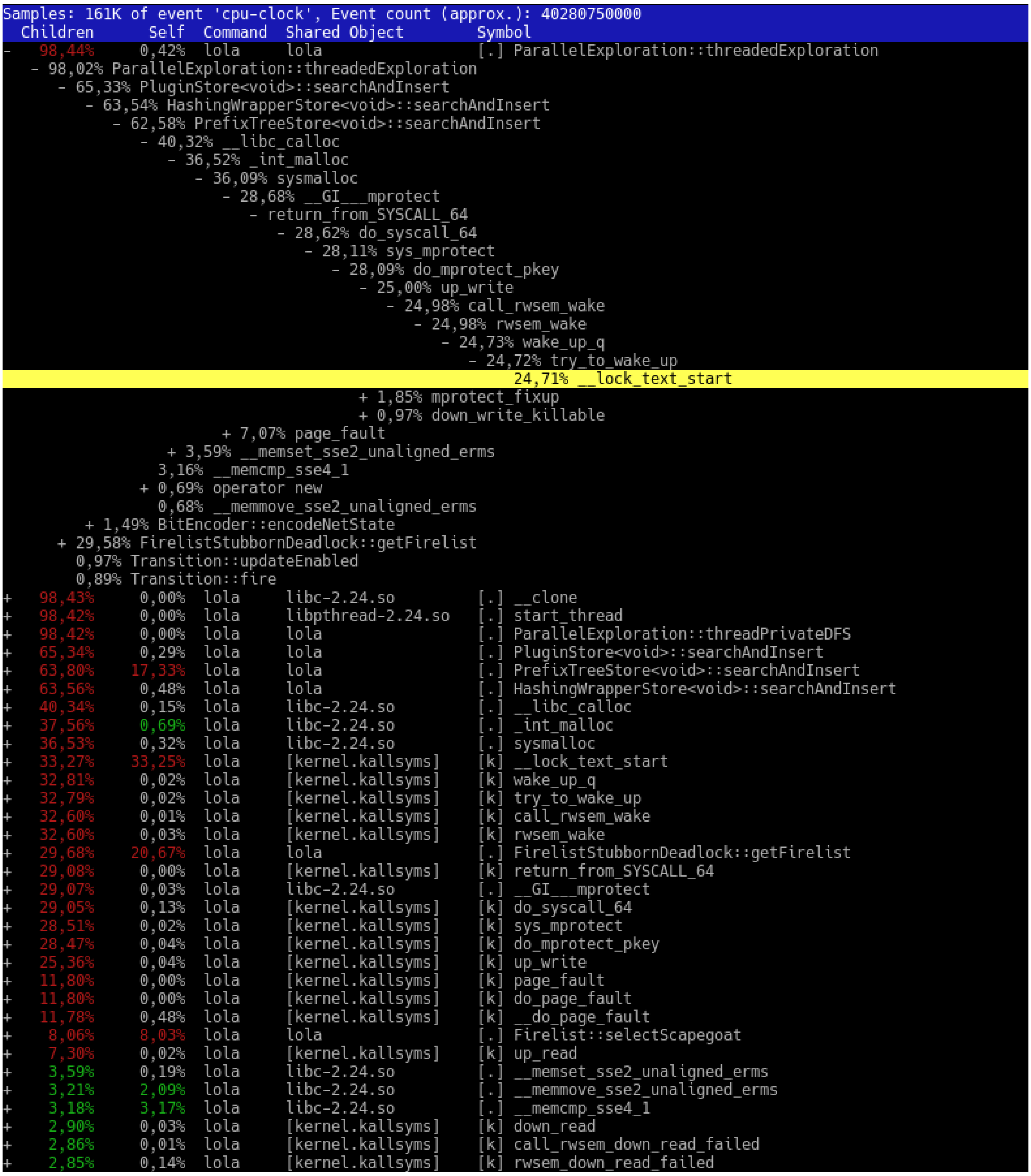
With `lola --check=full --threads=4 --bucketing=100 phils1000.lola&;` LoLA is executed in the background, `PID=$!;` stores the most recent background program in a PID variable, `sudo perf record -p $PID --call-graph dwarf sleep 30;` executes the perf profiler for 30 seconds and `kill $PID` kills LoLA after perf returns.

To analyze the data, perf provide the perf report command. It will process the gathered data and output an interactive data structure like in figure 3.2. In the output, we can see what functions are sampled most of the time (’Self’ column), in which functions children where taken the most samples (’Children’ column), the command name that was executed, the library in which the function resides and the name of the function. A more detailed explanation of the different columns and additional features of the perf report command can be found in the man page.

Knowing the basics for the perf usage, we can now look closer into the generated data. Now, we can make more reliable assumptions about the possible location of the bottleneck.

3.2.7 Result and conclusion

Figure 3.2 shows that over 95% of the samples were taken somewhere in the `threadedExploration`. This is expected because it is where the search of LoLA is implemented and almost all



```
Samples: 161K of event 'cpu-clock', Event count (approx.): 40280750000
  Children      Self  Command  Shared Object        Symbol
-   98,44%     0,42%  lola     lola                 [.] ParallelExploration::threadedExploration
   - 98,02% ParallelExploration::threadedExploration
      - 65,33% PluginStore<void>::searchAndInsert
         - 63,54% HashingWrapperStore<void>::searchAndInsert
            - 62,58% PrefixTreeStore<void>::searchAndInsert
               - 40,32% __libc_calloc
                  - 36,52% _int_malloc
                     - 36,09% sysmalloc
                        - 28,68% __GI___mprotect
                           - return_from_SYSCALL_64
                              - 28,62% do_syscall_64
                                 - 28,11% sys_mprotect
                                    - 28,09% do_mprotect_pkey
                                       - 25,00% up_write
                                          - 24,98% call_rwsem_wake
                                             - 24,98% rwsem_wake
                                                - 24,73% wake_up_q
                                                   - 24,72% try_to_wake_up
                                                        24,71% __lock_text_start
                                       + 1,85% mprotect_fixup
                                       + 0,97% down_write_killable
                        + 7,07% page_fault
                  + 3,59% __memset_sse2_unaligned_erms
                 3,16% __memcmp_sse4_1
               + 0,69% operator new
                 0,68% __memmove_sse2_unaligned_erms
         + 1,49% BitEncoder::encodeNetState
      + 29,58% FirelistStubbornDeadlock::getFirelist
        0,97% Transition::updateEnabled
        0,89% Transition::fire
+   98,43%     0,00%  lola     libc-2.24.so         [.] __clone
+   98,42%     0,00%  lola     libpthread-2.24.so   [.] start_thread
+   98,42%     0,00%  lola     lola                 [.] ParallelExploration::threadPrivateDFS
+   65,34%     0,29%  lola     lola                 [.] PluginStore<void>::searchAndInsert
+   63,80%    17,33%  lola     lola                 [.] PrefixTreeStore<void>::searchAndInsert
+   63,56%     0,48%  lola     lola                 [.] HashingWrapperStore<void>::searchAndInsert
+   40,34%     0,15%  lola     libc-2.24.so         [.] __libc_calloc
+   37,56%     0,69%  lola     libc-2.24.so         [.] _int_malloc
+   36,53%     0,32%  lola     libc-2.24.so         [.] sysmalloc
+   33,27%    33,25%  lola     [kernel.kallsyms]    [k] __lock_text_start
+   32,81%     0,02%  lola     [kernel.kallsyms]    [k] wake_up_q
+   32,79%     0,02%  lola     [kernel.kallsyms]    [k] try_to_wake_up
+   32,60%     0,01%  lola     [kernel.kallsyms]    [k] call_rwsem_wake
+   32,60%     0,03%  lola     [kernel.kallsyms]    [k] rwsem_wake
+   29,68%    20,67%  lola     lola                 [.] FirelistStubbornDeadlock::getFirelist
+   29,08%     0,00%  lola     [kernel.kallsyms]    [k] return_from_SYSCALL_64
+   29,07%     0,03%  lola     libc-2.24.so         [.] __GI___mprotect
+   29,05%     0,13%  lola     [kernel.kallsyms]    [k] do_syscall_64
+   28,51%     0,02%  lola     [kernel.kallsyms]    [k] sys_mprotect
+   28,47%     0,04%  lola     [kernel.kallsyms]    [k] do_mprotect_pkey
+   25,36%     0,04%  lola     [kernel.kallsyms]    [k] up_write
+   11,80%     0,00%  lola     [kernel.kallsyms]    [k] page_fault
+   11,80%     0,00%  lola     [kernel.kallsyms]    [k] do_page_fault
+   11,78%     0,48%  lola     [kernel.kallsyms]    [k] __do_page_fault
+    8,06%     8,03%  lola     lola                 [.] Firelist::selectScapegoat
+    7,30%     0,02%  lola     [kernel.kallsyms]    [k] up_read
+    3,59%     0,19%  lola     libc-2.24.so         [.] __memset_sse2_unaligned_erms
+    3,21%     2,09%  lola     libc-2.24.so         [.] __memmove_sse2_unaligned_erms
+    3,18%     3,17%  lola     libc-2.24.so         [.] __memcmp_sse4_1
+    2,90%     0,03%  lola     [kernel.kallsyms]    [k] down_read
+    2,86%     0,01%  lola     [kernel.kallsyms]    [k] call_rwsem_down_read_failed
+    2,85%     0,14%  lola     [kernel.kallsyms]    [k] rwsem_down_read_failed
```

Figure 3.2: perf report output

what LoLA does is searching. We can also see that nearly half of the samples were taken inside the `searchAndInsert` method of the global store. This observation corresponds to the hint we took from the manual benchmark approach. The different `searchAndInsert` methods are caused by the implementation of the `HashingWrapperStore` which associate each bucket with a `PrefixTreeStore` and forwards each `searchAndInsert` to the corresponding `PrefixTreeStore`. The new information that we get from the data is that the most samples inside the `searchAndInsert` methods are taken in a `__lib_calloc` call. Almost half the samples of all taken originate here. They are almost certainly caused by allocating memory on the heap with `malloc`, `calloc`, or `new`. This makes perfect sense since the purpose of the `searchAndInsert` method is to store newly discovered net states, which has to be done on the heap. In fact, a simplified high-level view of the search could be described as:

1. Discover a net state
2. Store the net state
3. Repeat until all states are discovered

Discovering net states is done by firing a transition from the fire list (a list of all transitions that can be fired from the current net state). Beside the `searchAndInsert` method we can see this behavior in the profiling data too. Getting the fire list is the second most sampled method inside the `threadedExploration` and firing the transition was also sampled often enough to account for almost 1% of all samples.

The calls inside `libcalloc` are a variety of system calls. Understanding what they really do is outside of the scope of this work. But the last sampled functions containing words like 'lock' or 'wake'. These are probably used to synchronize threads and lock memory regions. Ultimately the allocator has to be thread safe too.

Allocating heap space in high frequencies is no common use case, thus the assumption that the allocator just blocks until a request was fully processed seems to be reasonable. Additionally, there is only one system allocator for multiple threads. If they all call the same allocator it is likely that they get in each others ways with higher frequent calls and with an increasing amount of used threads.

With perfs data and the previous assumptions, we can be quite confident that changing the way we allocate heap space, can have an impact on LoLAs performance. For example, we can reduce the frequent and complex calls to the allocator, by using bigger preallocated chunks of memory and manage them by our selves. This would not only reduce the calls to `libcalloc` needed, it also has the potential of reducing the overall consumed memory, since we already store pointers to each net state inside the global store. The fact that we store them permanently until LoLA is terminated makes it unnecessary to manage additional data for defragmentation handling. This information can get quite big in the system allocator, especially when allocating a lot of small data like LoLA does. The chunks can also be associated with each thread or each bucket of the

`HashingWrapperStore` to make their access thread safe. This way we can access multiple memory locations by multiple threads. If thread synchronization is really a problem by the system allocator - like previously assumed, this approach potentially resolves the issue of the general sequential performance of LoLA with the parallel implementation.

An implementation would require a custom allocator that wraps the system calls. Fortunately, we have access to exactly such an allocator from a previous project at our chair called 'mara'. The participation of the author of this work in the development of mara is another advantage. The results of this circumstances lead to an integration of mara into LoLA for the net state search.

3.3 Allocation strategy change

In the last chapter, we saw that the profiling results lead deeper into the `searchAndInsert` method. The data suggest that high frequent calls to the system allocator might be the cause of the bottleneck. To see if this assumption is true, we have to change the way LoLA uses the heap and measure the performance again.

We will use the Memory and Resource Allocator (mara) to wrap calls to the system allocator. Mara was previously developed at the chair of theoretical computer science as a student project by Julian Gaede, Marian Stein, and Tom Meyer. The later use in LoLA was a part of the project goals.

A major part of maras constraints was that once allocated memory, will never be rearranged and never be freed until program termination. Thus mara is allowed to drop references that were previously shared. It can be freed later by the operating system. This allows mara to reduce the overhead for heap allocation in space and time.

Mara publishes a class that can be used as a custom allocator. A call to its `staticNew` method will return a pointer into a previously allocated chunk of memory. The next call to `staticNew` will return another pointer inside the same chunk that is adjacent to the previously returned memory segment. This memory management is equivalent to a stack. Once the chunk is filled, a new one is allocated with a call to the system allocator. The old chunk is not stored and there is no possibility to access it again with mara (but the previously shared pointers remain valid). Each object of the mara class manages disjoint parts of the heap.

With memory allocated by mara we can very easily distribute LoLAs net state memory, to make it thread-safe. With the `HashingWrapperStore` we already have a method to store net states in independent `PrefixTreeStores` (inside the buckets). The access to them is already restricted to one thread at a time, while other buckets still can be accessed by other threads. This means that we do not need to consider parallelization issues inside the buckets of the `PrefixTreeStore`. All memory access inside can only be done by a single thread. To use mara as an allocator, we just have to create one instance of the mara class and substitute all calls to `malloc`, `calloc` and `new` inside the `PrefixTreeStore` with a call to maras `staticNew` method. Because calls to `delete` and free will now produce undefined behavior, these calls have to be removed as well. The operating system will free the memory on program termination. After these steps, all heap allocation for the net states is wrapped by mara.

Altering LoLA in that way we now can start another benchmark to see if the changes impact LoLAs performance.

3.4 Results

To compare the old and the new implementation, we have to rerun our benchmarks. We will look again at the results of a new perf run. After that, we will run new macrobenchmarks to compare the different implementations.

3.4.1 Profiling

Figure 3.3 shows the output of the same mara call we used in chapter 3.2.7. We can see that the data changed in many ways. The `searchAndInsert` and the `getFirelist` methods now contribute to roughly the same amount of samples. They are also the functions with the highest 'Self' time. This means most samples were taken directly in this functions and not in one of their sub-functions.

The most important fact, however, is that the calls to `libcalloc` are completely gone, meaning they are under the significance threshold or where not sampled at all. Instead, we can see that maras `staticNew` method contributes to 0.4% of all samples.

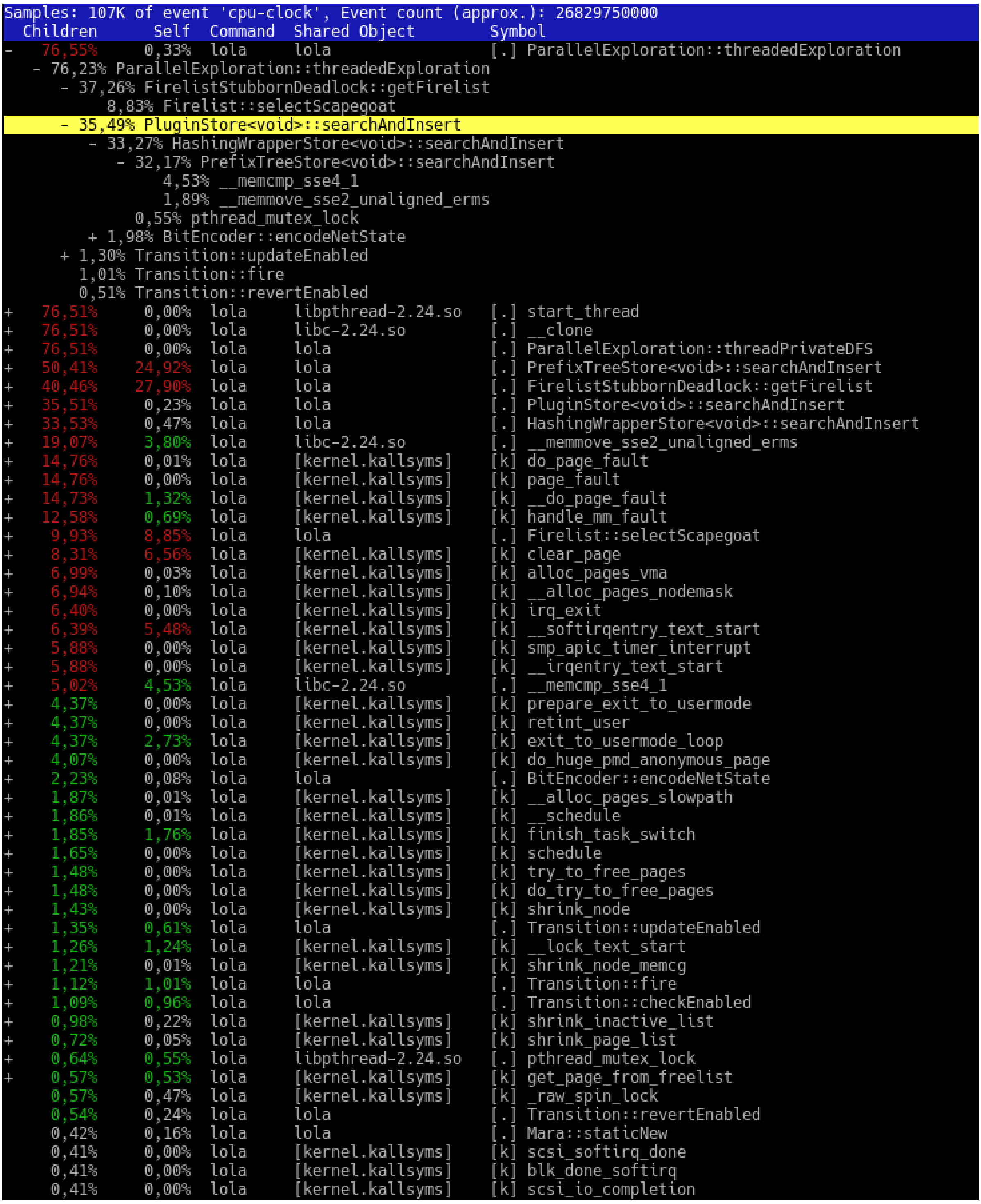
It seems that the integration of mara has a great impact on LoLAs performance. We seem to have greatly improved the way mara uses the heap. But we do not get any information on the execution time and the thread scaling from the profiling data. To draw any conclusion we have to make additional macro benchmarks and see how the changes affect the execution time.

3.4.2 Macrobenchmark

There are different questions that are of interest after changing the implementations. To address a variety of them, multiple tests were issued. We can divide them into three categories:

1. Single thread performance,
2. Implementation comparison and
3. Bucket influence.

The single thread performance will be compared without the use of the `HashingWrapperStore`. This way we can observe any efficiency differences that are caused by mara or the compare and swap (CAS) synchronization. To have insight into the thread scaling



```
Samples: 107K of event 'cpu-clock', Event count (approx.): 26829750000
  Children      Self  Command  Shared Object        Symbol
-   76,55%     0,33%  lola     lola                 [.] ParallelExploration::threadedExploration
   - 76,23% ParallelExploration::threadedExploration
      - 37,26% FirelistStubbornDeadlock::getFirelist
           8,83% Firelist::selectScapegoat
      - 35,49% PluginStore<void>::searchAndInsert
         - 33,27% HashingWrapperStore<void>::searchAndInsert
            - 32,17% PrefixTreeStore<void>::searchAndInsert
                 4,53% __memcmp_sse4_1
                 1,89% __memmove_sse2_unaligned_erms
              0,55% pthread_mutex_lock
         + 1,98% BitEncoder::encodeNetState
      + 1,30% Transition::updateEnabled
        1,01% Transition::fire
        0,51% Transition::revertEnabled
+   76,51%     0,00%  lola     libpthread-2.24.so   [.] start_thread
+   76,51%     0,00%  lola     libc-2.24.so         [.] __clone
+   76,51%     0,00%  lola     lola                 [.] ParallelExploration::threadPrivateDFS
+   50,41%    24,92%  lola     lola                 [.] PrefixTreeStore<void>::searchAndInsert
+   40,46%    27,90%  lola     lola                 [.] FirelistStubbornDeadlock::getFirelist
+   35,51%     0,23%  lola     lola                 [.] PluginStore<void>::searchAndInsert
+   33,53%     0,47%  lola     lola                 [.] HashingWrapperStore<void>::searchAndInsert
+   19,07%     3,80%  lola     libc-2.24.so         [.] __memmove_sse2_unaligned_erms
+   14,76%     0,01%  lola     [kernel.kallsyms]    [k] do_page_fault
+   14,76%     0,00%  lola     [kernel.kallsyms]    [k] page_fault
+   14,73%     1,32%  lola     [kernel.kallsyms]    [k] __do_page_fault
+   12,58%     0,69%  lola     [kernel.kallsyms]    [k] handle_mm_fault
+    9,93%     8,85%  lola     lola                 [.] Firelist::selectScapegoat
+    8,31%     6,56%  lola     [kernel.kallsyms]    [k] clear_page
+    6,99%     0,03%  lola     [kernel.kallsyms]    [k] alloc_pages_vma
+    6,94%     0,10%  lola     [kernel.kallsyms]    [k] __alloc_pages_nodemask
+    6,40%     0,00%  lola     [kernel.kallsyms]    [k] irq_exit
+    6,39%     5,48%  lola     [kernel.kallsyms]    [k] __softirqentry_text_start
+    5,88%     0,00%  lola     [kernel.kallsyms]    [k] smp_apic_timer_interrupt
+    5,88%     0,00%  lola     [kernel.kallsyms]    [k] __irqentry_text_start
+    5,02%     4,53%  lola     libc-2.24.so         [.] __memcmp_sse4_1
+    4,37%     0,00%  lola     [kernel.kallsyms]    [k] prepare_exit_to_usermode
+    4,37%     0,00%  lola     [kernel.kallsyms]    [k] retint_user
+    4,37%     2,73%  lola     [kernel.kallsyms]    [k] exit_to_usermode_loop
+    4,07%     0,00%  lola     [kernel.kallsyms]    [k] do_huge_pmd_anonymous_page
+    2,23%     0,08%  lola     lola                 [.] BitEncoder::encodeNetState
+    1,87%     0,01%  lola     [kernel.kallsyms]    [k] __alloc_pages_slowpath
+    1,86%     0,01%  lola     [kernel.kallsyms]    [k] __schedule
+    1,85%     1,76%  lola     [kernel.kallsyms]    [k] finish_task_switch
+    1,65%     0,00%  lola     [kernel.kallsyms]    [k] schedule
+    1,48%     0,00%  lola     [kernel.kallsyms]    [k] try_to_free_pages
+    1,48%     0,00%  lola     [kernel.kallsyms]    [k] do_try_to_free_pages
+    1,43%     0,00%  lola     [kernel.kallsyms]    [k] shrink_node
+    1,35%     0,61%  lola     lola                 [.] Transition::updateEnabled
+    1,26%     1,24%  lola     [kernel.kallsyms]    [k] __lock_text_start
+    1,21%     0,01%  lola     [kernel.kallsyms]    [k] shrink_node_memcg
+    1,12%     1,01%  lola     lola                 [.] Transition::fire
+    1,09%     0,96%  lola     lola                 [.] Transition::checkEnabled
+    0,98%     0,22%  lola     [kernel.kallsyms]    [k] shrink_inactive_list
+    0,72%     0,05%  lola     [kernel.kallsyms]    [k] shrink_page_list
+    0,64%     0,55%  lola     libpthread-2.24.so   [.] pthread_mutex_lock
+    0,57%     0,53%  lola     [kernel.kallsyms]    [k] get_page_from_freelist
     0,57%     0,47%  lola     [kernel.kallsyms]    [k] _raw_spin_lock
     0,54%     0,24%  lola     lola                 [.] Transition::revertEnabled
     0,42%     0,16%  lola     lola                 [.] Mara::staticNew
     0,41%     0,00%  lola     [kernel.kallsyms]    [k] scsi_softirq_done
     0,41%     0,00%  lola     [kernel.kallsyms]    [k] blk_done_softirq
     0,41%     0,00%  lola     [kernel.kallsyms]    [k] scsi_io_completion
```

Figure 3.3: perf report output of LoLA with mara integration

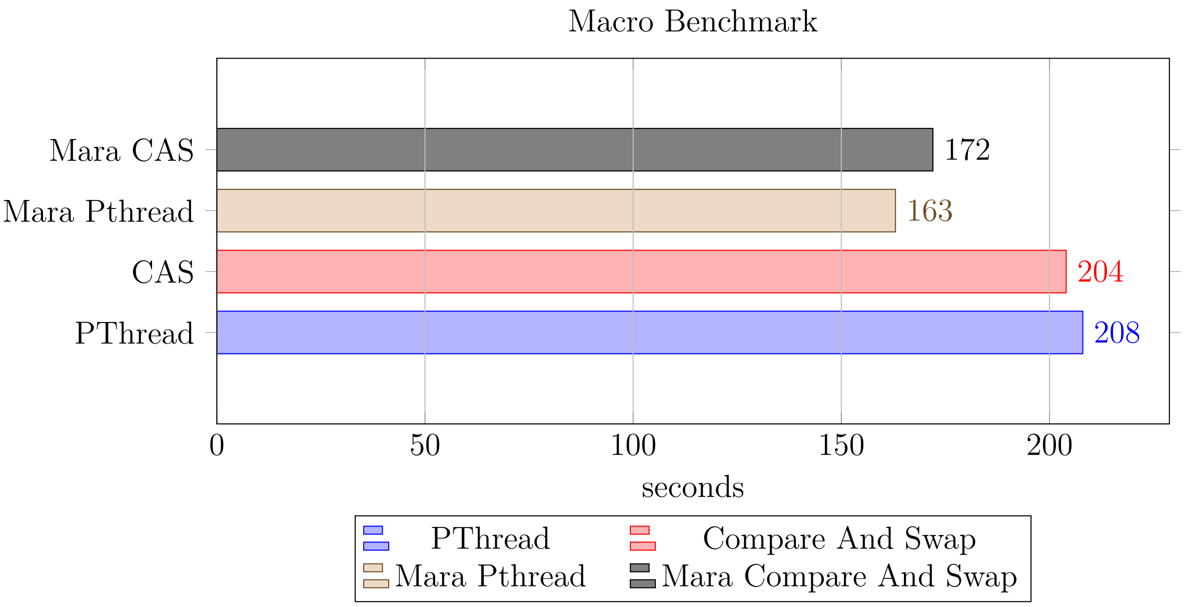


Figure 3.4: Single thread comparison between all implementations
LoLA call: ”lola –check=full –threads=1 ../tests/testfiles/phils1000.lola”

we will compare all implementations with use of the `HashingWrapperStore` combined with different amounts of threads. And to see how the amount of used buckets influences the results, we will look at just the implementations that are scaling with the threads and run them with a different amount of buckets.

We will investigate all implementation changes. The preexisting unchanged implementation will be called ’PThread’. The implementation with the changed synchronization method will be called ’CAS’ and the implementation with the integration of mara will be called ’Mara PThread’. Additionally, we will look at an implementation where the CAS synchronization and the mara integration was used. It will be called ’Mara CAS’.

In this case, we will run the benchmarks on ebro because it can access up to 64 threads. However, we will limit the thread count to a maximum of 50 so that other tasks are unlikely to interfere with our benchmark.

The benchmark values will be an average of 10 runs, so that performance deviations are compensated.

In the single thread benchmark - shown in figure 3.4 - a quite homogeneous characteristic can be observed. The difference between the CAS and the Pthread implementation is insignificant. But we can observe a speedup between the mara and non-mara implementations. With a single thread the pthread mutexes, still seem to work slightly faster than the compare and swap approach.

The benchmark for the implementation comparison shows that the original implementation is quite consistent between the used threads. This is due to bugs and will be explained further in chapter 3.4.3. We can see again that the CAS implementation

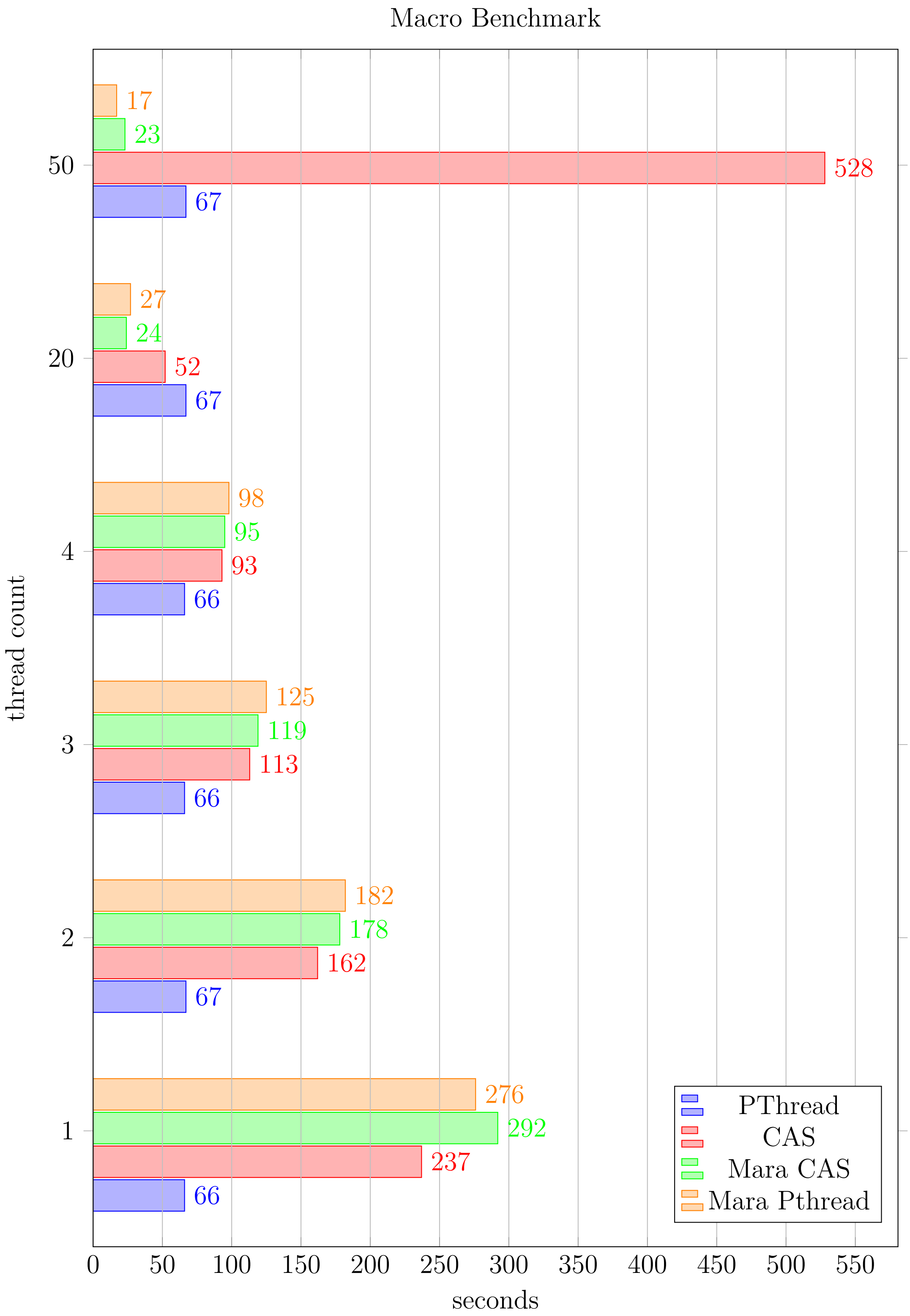


Figure 3.5: Implementation comparison with 100 buckets
LoLA call: ”lola –check=full –threads={1,2,3,4,20,50} –bucketing=100 ../tests/testfiles/phils1000.lola”

nether really outperforms the original implementation. Only with the addition that at a very high thread count, the execution time rises again to very high levels. Both mara implementations, however, are able to outperform the PThread implementation at a certain thread count. Even though they are performing considerably worse on lower thread counts.

But to utilize a maximum of the parallelization capabilities, the memory has to be used efficiently. Increasing the number of threads will increase the probability of concurrent access of the same bucket. In such a case the bucket will grant access to a single thread and will lock the access for others. Thus a higher thread count might ultimately cause an - again - decreasing performance.

We can decrease the chances of concurrent access by increasing the number of used buckets. Therefore figure 3.6 shows the performance of the mara implementations with a different number of buckets. We can observe a general performance gain, with a higher number of buckets. Even single-threaded. The difference slowly diminishes with more threads. However, a greater bucket count preserves a better thread scaling. We can observe a further decreasing of the execution time, with more buckets, on a higher thread count. Whereas the execution time of CAS with 100 buckets increases again between 20 and 50 used threads.

At this point, we can summarize that we have affected the performance with the new implementations. In some cases even for the better. In the next chapter, we will evaluate the data in more detail.

3.4.3 Evaluation

We saw in the results of the benchmark, that the new implementations perform differently. We can now analyze what the causes of these differences are.

But before that, it should be stated again that the benchmarks where run on a single Petri net and on a single machine. While the used hardware should hardly influence the relative performance differences, the test data can have a major effect. For example, a Petri net, where always at most on transition is fireable in each state, cannot be searched in parallel by the current algorithm. In such a case we would see no benefit by using multiple threads and parallelization would certainly perform worse due to its synchronization overhead.

Additionally, the use of LoLA with the `--check=full` parameter is by far not the only possible use of LoLA. It is probably most used with a predicate to check. In that case, the Petri net is often reduced to a smaller - with respect to the predicate - equivalent net before the search is done. This again can influence the number of fireable transitions and therefore the potential of the parallel search.

That said, the following should be seen as a constrictive comparison between the sequential and parallel capabilities of LoLA, not as an exhaustive one.

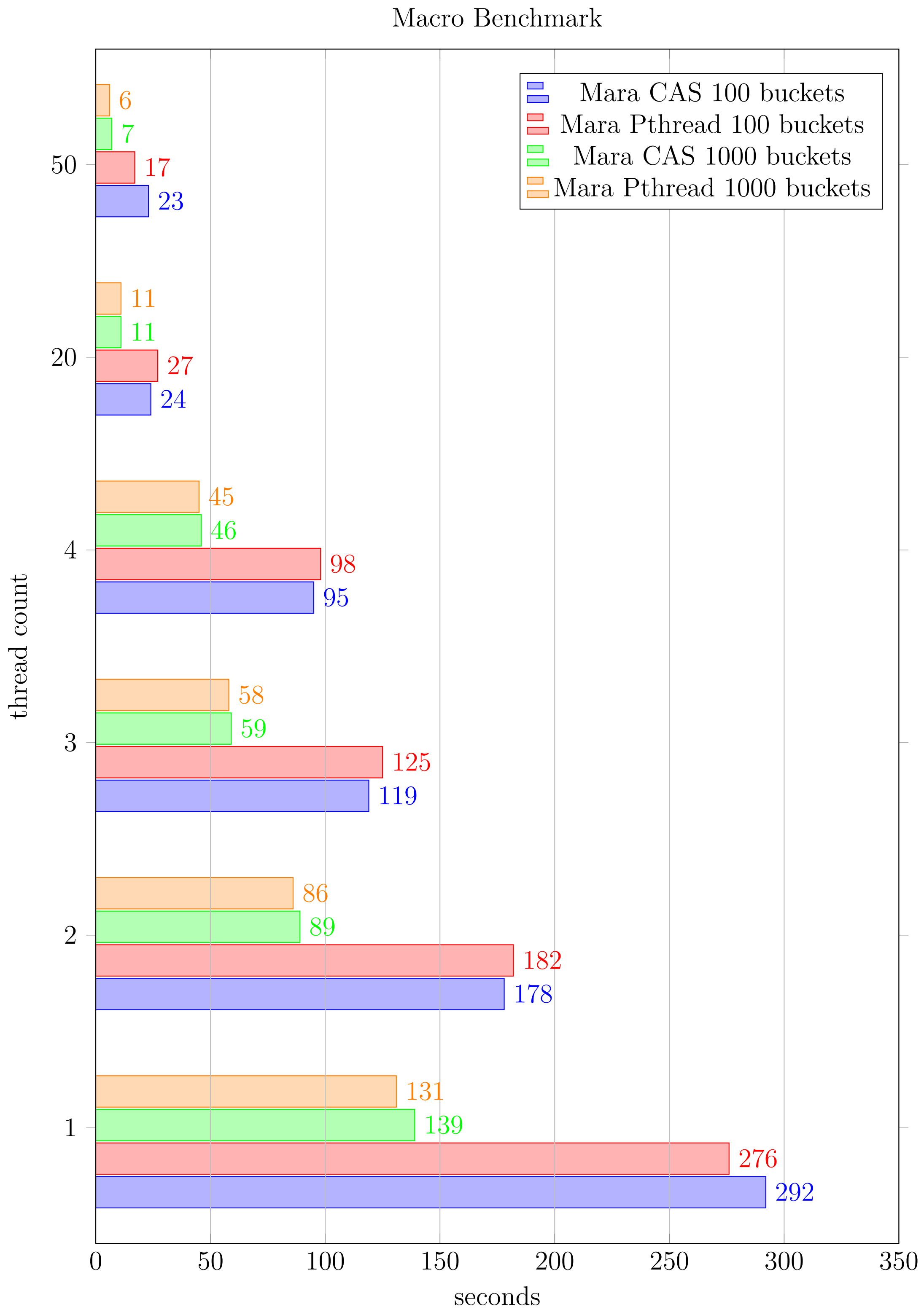


Figure 3.6: Mara comparison with 100 and 1000 buckets
LoLA call: ”lola –check=full –threads={1,2,3,4,20,50} –bucketing={100,1000} ../tests/testfiles/phils1000.lola”

3.4.3.1 Single thread performance

We have two different results on the single thread performance of LoLA. If the `HashingWrapperStore` is used all new implementations perform worse than the initial implementation. If it is not used the mara implementations perform better.

The performance gain in the latter can be explained exactly by the use of mara. Benchmarks in the development of mara showed that it is faster than a call to `malloc` because it can reduce an allocation call to basically three stack operations:

1. Save the current stack pointer
2. Increase the stack pointer by the size of the requested data
3. Return the saved pointer

Because LoLA requests a lot of heap space, this adds up to a significant amount of time.

But when we introduce the `HashingWrapperStore` with its buckets, the performance of all new implementations drop considerably, while the initial implementation gains performance.

The performance gain can be explained by the concept of the `HashingWrapperStore`. When we distribute all states evenly between the buckets, we can do an informed search later because we know the bucket associated with our hash value. Since every insert of a net state implies a previous search to avoid duplicates, it is possible to reduce the execution time with a suitable amount of buckets.

The performance loss can be explained by the implementation of the `HashingWrapperStore`. The original implementation was not guarded by mutexes and therefore not thread safe. Additionally, the bucket count from the lola call was ignored and a hard-coded value was used instead. This suggests that the implementation was incomplete and the use of buckets was experimental. For the use in the new implementations, the `HashingWrapperStore` had to be altered. Now, before a bucket will be accessed via hash, a mutex has to be passed to prevent concurrent access and a modulo check is done to assure that a bucket to insert exists. These new operations can accumulate a lot of time with frequent use by LoLA.

However, aside from the synchronization issues, it seems that mara has the potential to speed up the execution in general. The heap allocation is independent of the depth-first search and the optimization techniques incorporated by LoLA.

Also, it might be possible to affect the memory space consumption with mara. Though not benchmarked explicitly, while testing the implementations during development, a considerable memory reduction was observed. While the 1000 philosophers caused a premature termination of the tests with the `HashingWrapperStore`, the mara integration enabled LoLA to finish the test successful under the right circumstances. With the approximately 17GB of RAM on the VM, the original implementation terminated at about the half of net states to discover. But with a chunk size of 500MB it was possible to fit all net states in the memory. It can be assumed that this is a saving of multiple gigabytes and might be a matter that is worth investigating more in depth.

To address the worse performance with the changes in the `HashingWrapperStore`, it might be possible to reduce the execution time by checking with how many threads LoLA runs. A condition to skip the mutex blocks when using only one thread, or a completely new class that is loaded dynamically single-threaded, can be implemented. Anyway, the performance gap between the hashed and unhashed execution of the PThread implementation suggests that this matter can buy in execution time at expense for memory space.

3.4.3.2 Thread scaling

The main goal of this work was to achieve a scaling with the threads. As we can see from the benchmark data, the initial PThread implementation is not affected by the number of threads at all. The reason for this is that - due to a bug - the configured thread count is completely ignored.

The bug is caused by differing function signatures. LoLA uses different classes for the depth-first search when using one thread, and when using more than one thread. If LoLA is initialized with one thread it uses the `DFSExploration` class. With more threads, it uses the `ParallelExploration` class (that is derived from the `DFSExploration`). To start the search, the `depth_first` method is called with this signature:

```
bool depth_first SimpleProperty , NetState , Store<void>, Firelist , int)
```

Listing 3.4: Signature in `DFSExploration`

This method is implemented in the `DFSExploration` class. But in the `ParallelExploration` class the method is implemented with the signature:

```
bool depth_first SimpleProperty , NetState , Store<void>, Firelist ,
    threadid_t)
```

Listing 3.5: Signature in `ParallelExploration`

This probably causes an additional entry in the objects virtual table, so that the `depth_first` method appears multiple times with the two different signatures. Since LoLA calls the search method in the `ParallelExploration` with the first signature, the (sequential) base implementation is used instead of the new one. The bug is fixed simply by changing the child signatures `threadid_t` to an `int`.

If this bug is resolved we can observe an increasing performance with an increasing amount of threads. At least in the lower thread range. With 50 threads the CAS implementation performs worst (the striking outlier). Since the test was run with 100 buckets, this behavior can be explained with an increased blocking of threads not only on entering a bucket but also on allocating heap memory with `calloc`. The latter seems to have a greater effect if we compare the results with the mara implementations.

After switching to mara, the allocator blocking seems to be gone as expected. The mara implementations are the first to beat the single-threaded implementation with a high amount of threads. This leads to the assumption that allocating heap space with the system allocator was the main cause of the previous bottleneck. We can even see a linear scaling in the range from two to four threads. Only between one and two threads, the performance does not split in half, because the single-threaded algorithm does use the original `DFSExploration` which comes with much less synchronization overhead.

In contrast to the first impression, it is even possible that the linear scaling reaches the higher thread range. Eventually, LoLA has a single-threaded build up and teardown phase. This phases are included in LoLAs internal measurement and therefore are included in the results. But since we do not have reliable data for that phases, this matter will remain speculative for now.

3.4.3.3 Bucket influence

We can see continuous improvement with the use of additional buckets in the mara implementations.

One reason for that is probably the decreased chance for concurrent access. This gets more important with increasing threads. In fact, we can see a slight decrease of the mara CAS performance between 20 and 50 threads that seems to be compensated with the use of more buckets.

But the increasing single thread performance cannot be explained by fewer bucket conflicts. As described in 3.4.3.1, this behavior is probably caused by a reduced store search time. Because LoLA distributes all states in the buckets and knows in which bucket to search for a given net state, the overall search time is decreasing.

However, the extent of both effects is not reflected in the data. To evaluate this matter a macro benchmark does not suffice.

4 Conclusion

Optimizing software is a common part of its development cycle, but the approach depends highly on the issue.

A generally good order of actions is to first narrow down the performance issue and base implementation changes only on empirical test results after that. This approach might seem obvious, but during this work, a general temptation was observed to 'just check if this idea works'. The change between the Pthread and the Compare and Swap synchronization illustrates that. It was done before any serious measurement because a problem with the synchronization was a likely cause for the bad thread scaling. As the data shows, this assumption was right in principle, but in practice, another unexpected part of the synchronization was the cause. A better preparation could have saved work and time at this point.

Another part that could have been improved is the benchmarking approach. The third-party tools where much more reliable and easy to use than the manual approach. At least in this work, it can be said that the time spent, to learn how to use these tools, was well spent. With the help of this tools, it was possible to find and eliminate a serious bottleneck.

The data produced by the tools lead to the mara integration in LoLA, which is able to utilize additional threads for a higher performance. Though the synchronization overhead prevents an early improvement, we saw that there exist Petri nets which can be searched faster with these implementations and enough threads. Using more buckets can enhance this effect further. Additionally, the new heap allocation strategy can improve the general performance of LoLA independent of the parallelization.

In contrast - as implied earlier - the switch between the Compare and Swap synchronization and the PThread synchronization has no major effect on the performance. The compare and swap algorithm tends to perform generally slower than the PTread mutexes. It seems that the system allocator is the root cause of the bottleneck.

The real-life benefit of the parallelization still has to be validated. but with the insights of this work, LoLA can be understood better and we can deviate some work that can be done to further improve LoLA.

5 Future Work

The results show, that it is possible to further speed up LoLAs performance. But they draw not the complete picture. Using different optimization methods and other Petri nets, the performance gain will most certainly ease noticeably.

To classify the extent of this behavior, further benchmarks should be evaluated. They will show if the observed scaling is applicable in a productive environment. Additional a memory benchmark might be of interest, that compares memory load with the mara implementation (depending on the chunk size) in comparison to the old implementation.

However, LoLAs potential for parallel and sequential execution is by far not exhausted. For example, the new implementation of the `HashingWrapperStore` seems to be much slower than the previous version. As stated in chapter 3.4.3.1, a concept for a single-threaded use might decrease the execution time considerably. A special subclass that is specialized for - and only used on - single-threaded execution, might be a good way to achieve this goal.

As discussed in chapter 2.5.1 the implementation of the `ParallelExploration` class seems to be based on the single-threaded base class. With this approach, all threads used for the exploration, have the same job. They are also accessing equally, the same global store. A more specialized approach might deliver even better scaling effects, or reduce the overhead that is needed for the synchronization.

For example, currently, the store is accessed via the `searchAndInsert` method. As the name implies, this method not only searches the store for a net state - which requires reading access, it also inserts the state into the store if it was not found - which requires writing access. If it is possible to divide this two operations in a thread-safe manner, it might be possible to reduce the chance of blocking mutexes. Especially if we consider that the search depends on explored states, whereas inserting can be done in constant time.

Another possible improvement might be a master thread that queries for leftover work by worker threads, that can then be handed over to idle worker threads. This way it can be avoided to check for idle threads on each search cycle like it is done currently. This is basically another approach for load-balancing. A work from Holzmann already shows that an improved load-balancing can improve the results of related software[25].

However, this would be solutions that are bound to the current structure of LoLA. It might be worth to completely reimplement the parallel search. The recent development in depth-first search is trying to introduce efficient parallel extensions of existing algorithms[17]. With an algorithm and a data structure that is well tested and specifically designed for parallel use, it might be possible to achieve a better performance for more use cases.

Bibliography

[1] C. A. Petri, “Kommunikation mit automaten,” 1962.

[2] T. Murata, “Petri nets: Properties, analysis and applications,” Proceedings of the IEEE, vol. 77, no. 4, pp. 541–580, 1989.

[3] K. Schmidt, “Lola a low level analyser,” Application and Theory of Petri Nets 2000, pp. 465–474, 2000.

[4] D. B. F. Kordon. (2017) Model checking contest results 2017. Accessed: 2018-11-01. [Online]. Available: https://mcc.lip6.fr/2017/results.php

[5] D. W. Wall, Limits of instruction-level parallelism. ACM, 1991, vol. 19, no. 2.

[6] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in Proceedings of the April 18-20, 1967, spring joint computer conference. ACM, 1967, pp. 483–485.

[7] J. L. Gustafson, “Reevaluating amdahl’s law,” Communications of the ACM, vol. 31, no. 5, pp. 532–533, 1988.

[8] R. H. Netzer and B. P. Miller, “What are race conditions?: Some issues and formalizations,” ACM Letters on Programming Languages and Systems (LOPLAS), vol. 1, no. 1, pp. 74–88, 1992.

[9] E. W. Dijkstra, “Cooperating sequential processes,” in The origin of concurrent programming. Springer, 1968, pp. 65–138.

[10] J. A. Bondy, U. S. R. Murty et al., Graph theory with applications. Citeseer, 1976, vol. 290.

[11] S. W. Golomb and L. D. Baumert, “Backtrack programming,” Journal of the ACM (JACM), vol. 12, no. 4, pp. 516–524, 1965.

[12] J. Freeman, “Parallel algorithms for depth-first search,” 1991.

[13] J. R. Smith, “Parallel algorithms for depth-first searches i. planar graphs,” SIAM Journal on Computing, vol. 15, no. 3, pp. 814–830, 1986.

[14] V. N. Rao and V. Kumar, "Parallel depth first search. part i. implementation," International Journal of Parallel Programming, vol. 16, no. 6, pp. 479–499, 1987.

[15] A. Aggarwal, R. J. Anderson, and M.-Y. Kao, "Parallel depth-first search in general directed graphs," in Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing, ser. STOC '89. New York, NY, USA: ACM, 1989, pp. 297–308. [Online]. Available: http://doi.acm.org/10.1145/73007.73035

[16] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM journal on computing, vol. 1, no. 2, pp. 146–160, 1972.

[17] V. Bloemen, A. Laarman, and J. van de Pol, "Multi-core on-the-fly scc decomposition," ACM SIGPLAN Notices, vol. 51, no. 8, p. 8, 2016.

[18] S. Oaks, Java Performance: The Definitive Guide: Getting the Most Out of Your Code. " O'Reilly Media, Inc.", 2014.

[19] A. M. D. Inc. (2018) Amd opteron 6284 se specification. Accessed: 2018-15-01. [Online]. Available: https://www.amd.com/de/products/cpu/6284-se

[20] E. W. Dijkstra, "Hierarchical ordering of sequential processes," in The origin of concurrent programming. Springer, 1971, pp. 198–227.

[21] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in ACM Sigplan Notices, vol. 17, no. 6. ACM, 1982, pp. 120–126.

[22] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in ACM Sigplan notices, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[23] V. Developers. (2017) Valgrind's tool suite. Accessed: 2018-02-02. [Online]. Available: http://valgrind.org/info/tools.html

[24] B. Gregg. (2018) perf examples. Accessed: 2018-05-02. [Online]. Available: http://www.brendangregg.com/perf.html

[25] G. J. Holzmann, "A stack-slicing algorithm for multi-core model checking," Electronic Notes in Theoretical Computer Science, vol. 198, no. 1, pp. 3–16, 2008.

Eidesstattliche Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Rostock, 02. März 2018

Tom Meyer