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

UNIVERSITÄT ROSTOCK | FAKULTÄT FÜR INFORMATIK UND ELEKTROTECHNIK

Institut 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 Zusatandsraums möglich, Eigenschaften von Petri Netzen auszuwerten. Die dafür notwendige Zusatndsraumsuche wurde dabei bisher durch einen sequentiellen Algorithmus vorgenommen.

In den letzten Jahren haben sich im Hardwarebereich jedoch mehrkern Prozessorarchitekturen durchgesetzt. LoLAs sequentieller Ansatz kann deshalb oft das Rechenpotential moderner Maschinen nicht mehr komplett ausnutzen. Auch ein Versuch den vorhandenen Algorithmus zu parallelisieren konnte bisher noch keine Performanceverbesserung hervorrufen.

In dieser Arbeit wird gezeigt das es möglich ist, eine Performanceverbesserung durch Nutzung von mehreren Threads, zu erzielen. Wir werden den vorangegangenen Versuch analysieren und die Implementation so anpassen, dass eine Skalierung der Performance mit der Anzahl der genutzten Threads zu beobachten ist. Die Verbesserung wird dabei an einem verbreiteten Petri Netzt 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 architectures 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. For that we will analyze the previous parallel implementation of LoLA and improve it to achieve a performance scaling with the amount of used threads. We will show that a common used petri net will benefit from the new implementation.

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

Tag der Ausgabe: 13.10.2017
Tag der Abgabe: 02.03.2018

Contents

1 Introduction

A thesis statement should do the following:

Explain the readers how you interpret the subject of the research Tell the readers what to expect from your paper Answer the question you were asked Present your claim which other people may want to dispute

Make sure your thesis is strong.

- What is paralelization
- What are petri nets
- What should be done in lola
- How should it be done
— profiling
- What was done
- What can be done in the future

2 Related Work

2.1 Parallelization History

2.2 Synchronization Methods

- mutexe
- semaphores
- compare and swap
- spin locks

2.3 Depth First Search Parallelization

- what is a graph

2.4 petri nets

-reachability - what means firing a transition

3 Background

The purpose of this work is to efficiently make use of multiple threads with the LowLevelAnalyzer (LoLA).

Lolas development startet in 1998. It was aimed to be used by third party tools to check properties of petri nets [Sch00]. Since then it was steadily updated to compete with other state of the art tools. Recurring prizes in a model checking contest with focus on petri nets suggest a success in this attempt [FK17]. 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 an undirected graph. The graph is discovered during the search itself.

Parallelization of depth first search is a difficult matter. Although rather general algorithms exist (some of them mentioned in the previous chapter), they usually make use of an assumption which cannot be made with LoLA.

For the quite specialized search in LoLA, a previous attempt of parallelization exists. Unfortunately the performance is usually worse then the single threaded approach. It even gets worse the more threads are used.

This is the staring point of this work. There is an algorithm which is not performing as expected. The most important task, is to find the bottle neck of this algorithm. After that it is desired to remove that bottleneck. The improved version should hopefully outperform the single threaded algorithm with use of a reasonable amount of threads. Additionally a desirable result would show a linear (or better) scaling of the performance with the amount of threads.

3.1 Performance Measurement

3.1.1 A distinction between different views on software

The performance of a program is a property which is always of interest somehow. Even if it is not a critical aspect in the working environment. But as reachable the concept of performance might look at first, the complicated it can get.

Just like that, what is called a performant application is highly dependent on its environment. The possibly most obvious characteristic would be the time it takes to execute a program or a subroutine. But in a lot of szenarios, 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 micro seconds. 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 beside execution time and memory consumption can be network latency, energy consumption or responsibility. Ultimately it is everything that increases the useability of a program.

After knowing what will be measured the reasonable 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. Oaks describes a general distinction between macro, meso and micro benchmarks [Oak14]. Each type addresses a different scope of a programm.

Macro benchmarks give an overview over 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: the difference between the end and the start of the program. The 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 known point in time and therefore the execution time is of limited use.

Micro benchmarks consider the most narrow aspects of an application. They messure 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 optimization strongly influences the results. Assumptions that work for the single code snippet might become false after the compiler integrates that part into the complete program or vise 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 often 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 get more complex. Like Modules or further reaching subroutines. These can be the most interesting benchmarks since they can narrow down bottle necks to specific parts of the software. Unfortunately they often require to manipulate the program in some way. Which again means that the measured performance differs from the actual performance.

3.1.2 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 satisfying results. And there are probably other reasons that can be thought of, but short, temporary measurements of this kind might give o quick overview 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 it and how to interpret its results.

Profilers aid to do the things automatically that otherwise would have to be implemented manually. An often used method to figure out bottle necks 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, 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.

3.2 Implementation Details

3.2.1 Code Base

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 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 where 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 choses 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 multi threading (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 then 1 will tell LoLA to use the `ParallelExploration` class and the bucketing switch will signal to use a hashing store.

3.2.2 Data Synchronization

To keep global data consistent, a method is needed to synchronize this kind of data between threads. 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 this 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 this 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 pthread library [**?**]. A possible substitute would be an own implementation with a Compare And Swap instruction as spin lock. This has the potential to keep the overhead for the locks at a bare minimum.

3.3 Environment

The used development environment consist of two machines. The general development is done on a virtual machine (vm). It runs with 4 threads on a Intel(R) Core(TM) i7-4770S CPU @ 3.10GHz with 4 physical cores. Each core supports hyper threading 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. Ubuntu 4.10 is running as operating system.

As a performant testsystem a powerful server (ebro) was used. Ebro has 4 AMD Opteron(tm) 6284 SE processors. Each with 16 cores and 2.7GHz base clock with a max boost of 3.4GHz [Inc18]. This sums up to 64 physical cpu cores (and threads) in total. As physical memory 995.2 GHz are installed. The operating system is CentOS Linux 7 Core (collected with 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	Ubuntu 4.10	CentOS 7 Core

Table 3.1: Specifications of the two development systems

4 Approach

4.1 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 replace 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 for 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 spin lock like mutexes and semaphores 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 preceeding manipulations are completed. At last there are functions necessary that can block or release the resource in a privileged manner. The resulting implementation is shown in listing 4.2.

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 importend features like a mechanism to reduces the risk of dead locks and different modes for the semaphores.

```
1 #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 4.1: Basic implementation of a spin lock with compare and swap

4.2 Finding the bottleneck

4.2.1 Benchmark characteristics

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

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

As test systems the both machines described in 3.3 where used.

As test data a predefined petrinet 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 [Dij71]. 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 reachability graph is also reasonably branched to allow a parallel discovery. The actual 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 amount of threads that should be used for the state

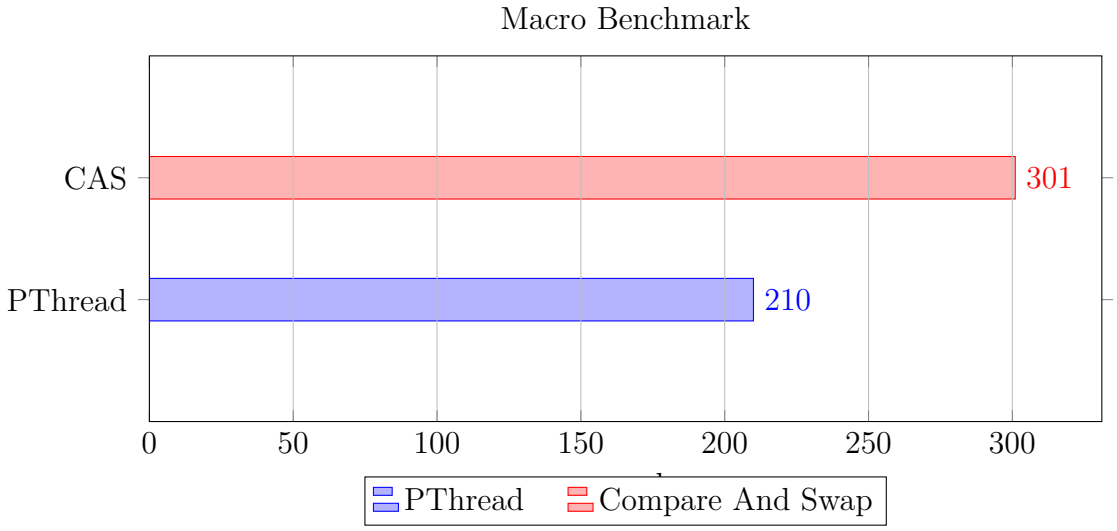


Figure 4.1: Macro benchmark comparison between the original (PThread) implementation and the substitute (Compare and swap)
LoLA call: ”lola –check=full –threads=4 ../tests/testfiles/phils1000.lola”

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.

4.2.2 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 testrun on the VM reveals that there is no performance gain. Figure TODO:ref shows that the new implementation is actually slower than the previous. But as stated in TODO:ref, the performance of the original implementation always stayed behind the sequential algorithm. This means that a relevant speedup would divide the execution time by an order of the used threads. Taking this expectation as base, both implementations still can be considered in the same order of performance. Thus, we can conclude that we either missed the cause of the bottleneck, or the new implementation faces similar challenges.

The benchmark result shows no significant change in performance. Therefore an exhaustive evaluation with multiple testruns is dropped in favor of a detailed bottle neck analysis. In the next section we will use a more precise benchmark method to inspect the search characteristics.

4.2.3 Searching for inefficient application components

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 amount 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.

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: 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 our case we 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 for learning a third party software.

In the next step measurements have to be inserted at 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 in the short method.

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

- Total Thread Time - The cumulative time spend in each thread is called total thread time.
- Synchronization Time - Time spend to synchronize the threads with mutexes.
- Store Search Time - Time spend inside the `searchAndInsert` call. A method to safe the discovered states.
- Idle Time - The Time each thread spent for the `restartSemaphore` to be unlocked
- Work - Time that each thread spend while having states left that can be discovered

- No Work - Time spend 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 4.1: Manual meso benchmark of the `ParallelExploration`

4.2.4 Following the hints

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 spend 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 if the measurements have to be taken with caution, we can take two hints from them. First, the time spend inside the mutexes seem to be insignificant. Whereas the time spend 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.

The store that 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)` shows that the id is completely ignored. 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 seems inevitable.

4.2.5 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 vise versa. This quickly leads to wrong times without noticing it.

The second problem refers to the kind of time that is measured. Especially in a multi threaded environments a distinction between real time (or wall-clock time) and cpu time. 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 a nother 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 amount 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.

4.2.6 Accepting help

If a challenge gets to difficult to face it alone, getting help is something worth considering. In our case, the difficulties and the trust in the own approach depleted so much that the

use of a well designed tool seems to outweigh the effort that has to be spent to use one efficiently.

But before learning to us 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 we decided to select some candidates from an internet research that are mentioned often in similar environments to test them in ours.

In the following a short overview on 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.

4.2.6.1 Gprof

Gprof is a profiler that was developed in the 1980's. It generates a flat and a call graph profile [GKM82]. In the flat profile is listet 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 is integrated into the gcc. Passing the `-pg` switch to the 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 later be written to an output file.

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 instead. This saves a lot of time during the tear down of the program because the destructor of allocated object 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 cases 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 outputfile.

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

4.2.6.2 Valgrind

"Valgrind is a dynamic binary instrumentation framework designed for building heavy-weight dynamic binary analysis tools" [NS07]. Dynamic binary analysis is used to "analyse 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 analysed program. For example the speed of an application analysed with memcheck is reduced by a factor of 10 to 30 [Dev17]. But an even more problematic detail for our szenario 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.

4.2.6.3 Perf

Perf is a profiler that is directly integrated into the linux kernel. It uses hardware performance counters which are unique registers on the processor, that count certain events like cache misses or branch mispredictions. This OS and hardware proximity make the use of perf very low 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 straight forward and produces good organized data. It is easy possible to trace for bottlenecks down to the (with debug symbols annotated) disassembly. But interpreting the data demands some effort for the user. For example one must know how to read the disassembly and how find the corresponding sections in the source code, to utilize the potential that comes with this feature.

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 [Gre18]. 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 und what data it collects.

4.2.6.4 Use of perf

The examples which where 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 only the ”stack

chain/backtrace" feature to generate a call-graph. With this option perf periodically takes (samples) stack traces which can be used to analyse which functions where called the most, from which function they where called and which functions they where 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=10000 phils1000.lola&;
PID=$!;
sudo perf record -p $PID --call-graph dwarf sleep 30;
kill $PID
```

Listing 4.2: 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 the 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 4.2. 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, 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 and make more reliable assumptions about the possible location of the bottleneck.

4.2.7 Result and Conclusion

Figure 4.2 shows that over 95% of the samples where taken somewhere in the `threadedExploration`. This is expected because it is where the search of LoLA is implemented and almost all what LoLA does is searching. We can also see that nearly half of the samples where taken inside the `searchAndInsert` method of the global store. This observation corresponds to the hint we took from the manual benchmark approach. The two different `searchAndInsert` methods are caused by the implementation of the `HashingWrapperStore` which associate each bucket with a `PrefixTreeStore` and forwards each `searchAndInsert`

```
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 4.2: perf report output

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 the 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 perpose 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 on 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 al 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 lib calloc 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 LoLA performance. For example we can reduce the frequent and complex calls to the allocator, by using bigger preallocated chunks of memory (from the system allocator) and manage them by our selfs. This would not only reduce the calls to lib calloc 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 potentially resolves the issue of the general sequential performance of LoLA with the parallel implementation.

An implementation for this approach requires 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 at 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.

4.3 Allocation strategie 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 und 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 aims.

A major part of maras constraints was that once allocated memory will never be rearranged and never be freed until program termination. Thus it can be freed by the operating system and mara is allowed to drop references that where previously shared. 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 now 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 `PrefixTreeStore`s (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 don not need to consider parallelization issues inside the buckets `PrefixTreeStore`. All memory access inside can only be done by a single thread. To use mara as 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. The operating system will free the memory on program termination. After that, 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.

4.4 Results

4.4.1 Profiling

- same environment like the previous prof env
- PIC: data
- libcalloc calls are gone -¿ under significance threshold
- mem... calls and fire list calls are now the most significant values
- meaning the greatest bottleneck seems to be gone

4.4.2 Macrobenchmark

- knowing a major bottleneck is gone, execution and scaling of the whole program is of interest
- simple measurement of execution time is sufficient
- since multiple threads will influence time consumption directly
- lola can log that itself
- all implementations are benchmarked (master, CAS, MaraCas, MaraPthread)
- significant values are: single thread performance, scaling with threads, scaling with bucket count
- PIC: measurements

4.5 Evaluation

5 Conclusion

- parallel dfs is not trivial
- messure the performance bottlenecks before fixing the performance!
- location of bottlenecks quickly become a matter of faith
- do not base actions on faith but on evidence, time and effort can be saved
- use tools which address the own context
- lola has potential in data representations in the context of parallelizations

- extension of single threaded class means that it is unlikely to beat the former single thread performance

- using tools can be frightening might also be worth it

6 Future Work

- messure performance from searchAndInsert
- find the limiting parts which hinder parallelization from using all threads efficiently
- optimize at least one implementation from search and insert for parallel use
- if the bottleneck returns to the exploration after optimizing:
- find a reasonable heuristic to share work between threads
- maybe don't let every single thread look for other idle ones, instead consider implementing a scheduler thread, or a data structure with possible tasks

- the synchronization method should be considered in the parallel design of the algorithm (dont emulate mutexe if using compare and swap)

Bibliography

[Dev17] Valgrind™ Developers. Valgrind's tool suite, 2017. Accessed: 2018-02-02.

[Dij71] Edsger W Dijkstra. Hierarchical ordering of sequential processes. In The origin of concurrent programming, pages 198–227. Springer, 1971.

[FK17] D. Buchs F. Kordon. Model checking contest results 2017, 2017. Accessed: 2018-11-01.

[GKM82] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In ACM Sigplan Notices, volume 17, pages 120–126. ACM, 1982.

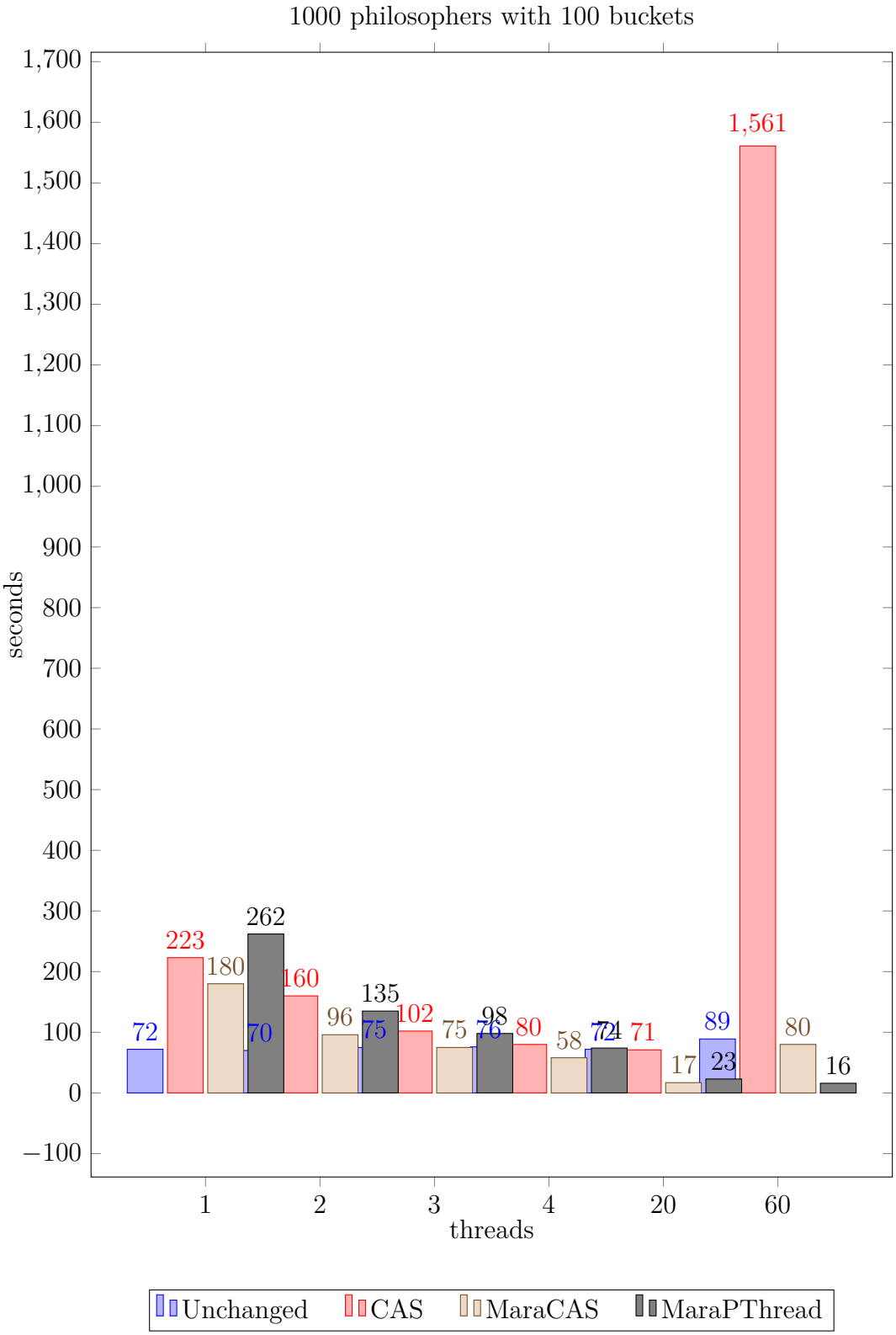
[Gre18] Brendan Gregg. perf examples, 2018. Accessed: 2018-05-02.

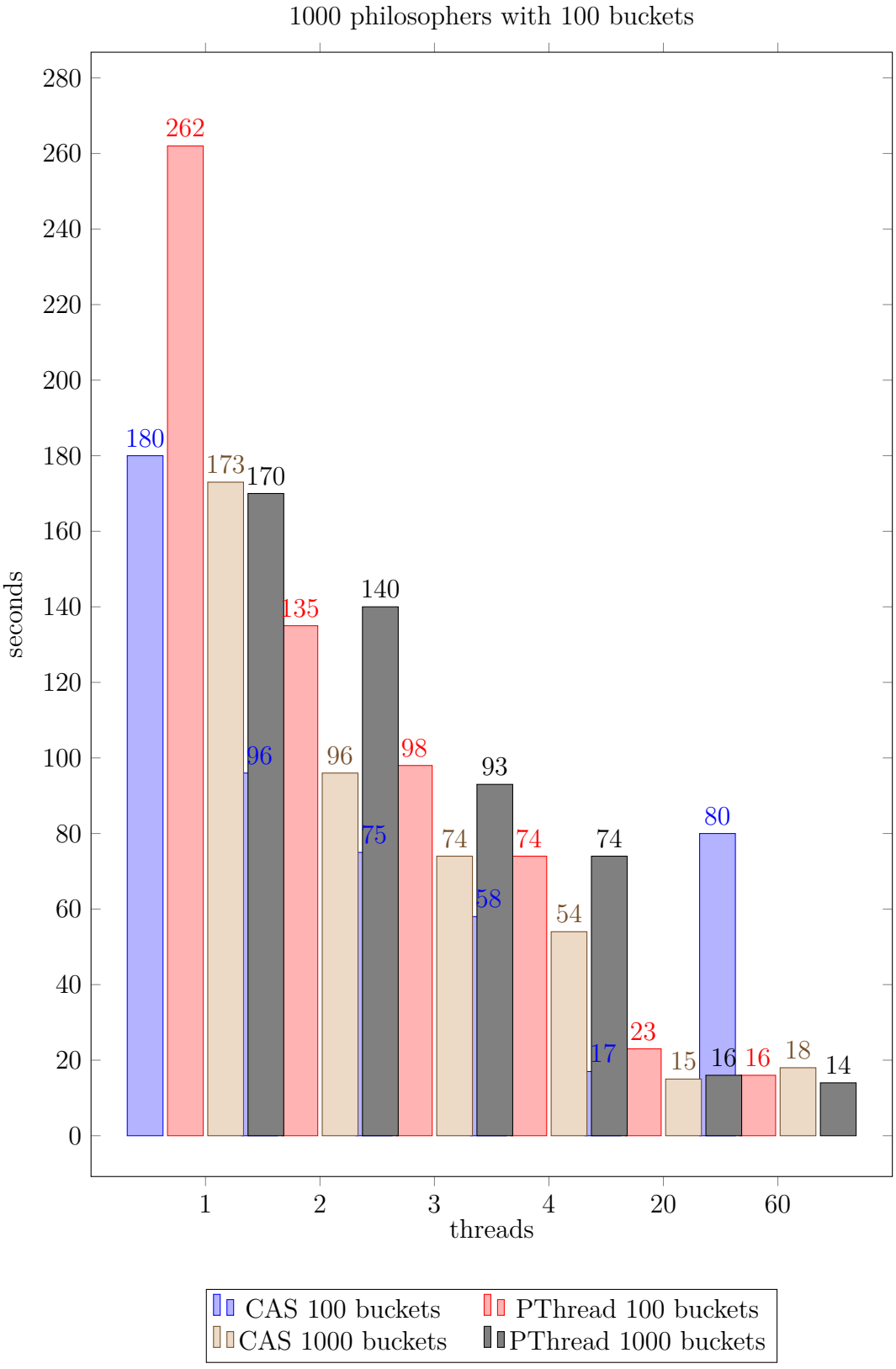
[Inc18] Advanced Micro Devices Inc. Amd opteron 6284 se specification, 2018. Accessed: 2018-15-01.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In ACM Sigplan notices, volume 42, pages 89–100. ACM, 2007.

[Oak14] Scott Oaks. Java Performance: The Definitive Guide: Getting the Most Out of Your Code. " O'Reilly Media, Inc.", 2014.

[Sch00] Karsten Schmidt. Lola a low level analyser. Application and Theory of Petri Nets 2000, pages 465–474, 2000.


1000 philosophers with 100 buckets
seconds
threads
1,700
1,600
1,500
1,400
1,300
1,200
1,100
1,000
900
800
700
600
500
400
300
200
100
0
−100
1
2
3
4
20
60
72
223
180
262
70
160
96
135
75
102
75
98
76
80
58
74
72
71
17
89
23
1,561
80
16
Unchanged
CAS
MaraCAS
MaraPThread


1000 philosophers with 100 buckets
seconds
threads
0
20
40
60
80
100
120
140
160
180
200
220
240
260
280
1
2
3
4
20
60
180
262
173
170
96
135
96
140
75
98
74
93
58
74
74
54
17
23
80
15
16
16
18
14
CAS 100 buckets
PThread 100 buckets
CAS 1000 buckets
PThread 1000 buckets

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