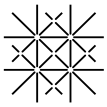

University of Basel

Investigation of the Relation Between the Linux Operating System Scheduler and Scheduling Decisions at Thread and Process Level

Master Project

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
HPC Group
https://hpc.dmi.unibas.ch/

Advisor: Prof. Dr. Florina M. Ciorba
Supervisor: Jonas H. Müller Korndörfer

David Kuhn
david.kuhn@stud.unibas.ch
16-057-960

30.06.2022

Table of Contents

1 Introduction

The behavior of parallel applications is influenced by multiple factors during the execution. The distribution of work, system noise, and OS scheduler routines introduce additional work that causes overhead to the execution of applications.

The Linux operating system (OS) uses the complete fair scheduler (CFS). CFS triggers thread migrations to balance the load among different cores. To let other threads execute, CFS interrupts running threads and initiates a context switch. All this additional work slows down single threads of the application, which leads to load imbalance. Also, the amount of idle CPU time is a factor that influences the application.

In this work, we investigate how much overhead the Linux scheduler causes on different applications with different thread level scheduling techniques. We want to quantify the overhead that scheduling introduces. Measuring the influence of the OS is not easy. There are many tools to measure different performance aspects of applications. But few distinguish between the OS background work and the application. Perf is a tool that achieves this with little overhead during the measurement.

We use perf to measure the OS scheduling overhead on the execution of parallel applications. We measure the impact of thread migrations, context switches, and CPU idle time. We measure the impact of these events on different types of parallel applications.

To investigate the relation between OS scheduling and application thread level scheduling, we compare the results of executions with different thread level scheduling techniques.

Besides this, we also evaluate how OS level scheduling overhead and application thread level scheduling influence two applications executed at the same time. We execute two applications that share a system and compete for resources. The OS scheduler has to balance the load and give all threads a fair share of execution time.

In chapter 2 we discuss related work that quantified OS scheduling in different ways. We explain our procedure and how we conducted the measurements in chapter 3. In chapter 4 we present the results of our experiments. We end with the discussion and future work 5.

2
Related Work

Petrini et al. [17] present how they improved the performance of the supercomputer ASCI Q. They describe several different techniques and tools to analyze the performance of the system. One technique is to use an application with different system configurations and measure how it performs. For this purpose, they used SAGE, an application paralyzed with MPI. They show that the performance of the application executed on more than 256 nodes improved when they used fewer processors per node. On ASCI Q each node had four processors. Although they used fewer cores in this experiment, which results in 25% less processing power, they reported better performance. The reason for this is, that the OS uses the idle cores for background work unrelated to the application. This work does not affect the performance of the application. There is less delay, which influences the performance of all nodes when they wait for the slowest node at the next barrier. Another problem was, that there was high variability in the performance for the individual cycles. To address this, they improved the synchronization phase of the application, which did not reduce the overhead significantly. Not all applications are impacted the same by each noise frequency. Fine-grained applications are more influenced by fine-grained noise. Coarse-grained applications are more affected by low-frequency noise. Applications that communicate less frequently are less affected by high-frequency noise because they become co-scheduled. On the other hand, fine-grained applications are more impacted by this noise. Petrini et al. reduced noise by removing unnecessary daemons and reducing the frequency of heartbeats that are necessary for the correct functioning of the system. These alterations to the system lead to higher performance. In this work, we concentrate on the influence of the OS scheduler. We also do not investigate the influence of idle cores on the scheduling overhead.

Akkan et al. [3] review methods to reduce interruptions to the HPC applications. They consider compile- and runtime measurements on an unmodified Linux kernel. To measure the effect of kernel-induced noise, they used a series of benchmarks. Another way to find information about the system interrupts is in the file /proc/interrupts. There is a list of the total accumulated counts of interrupt sources since the last system boot. The highest number is usually Local Timer Interrupts. At each of those time ticks, several tasks are executed that often are not relevant to the HPC application. For example, scheduling accounting and possible preemption of the executing task, or global kernel time updates. Akkan et al.

analyzed existing ways to reduce this OS noise. An easy way to reduce load balancing is to pin each application process to a CPU with the job launcher. But this does not pin system services to a CPU. These tasks are therefore often migrated for load balancing. It is possible to use one or several CPUs less than are available, to leave them for the OS tasks [17]. This reduces the computing power for the application, but it decreases the migration overhead. HPC job launchers, for example, SLURM, use kernel Control Groups (cgroups) to create virtual partitions for a set of CPUs. This prevents interference with other jobs and system services. It is also possible to turn off scheduler load balancing in cgroups. They used Fixed Work Quantum (FWQ) benchmark to measure the system noise under different conditions. They reported the least noise with scheduler load balancing explicitly turned off. Simply pinning tasks to a CPU does not provide the same results. To identify events, that cause overhead, they used the tool ftrace. Akkan et al. modified the Linux kernel, to isolate the application from OS jitter with dedicated cores. These OS cores execute OS tasks. This improves the performance of HPC applications because the tasks that would interrupt the application can execute on these separate cores. They also describe how they customized the Linux kernel to resemble a lightweight kernel. This reduces the number of interrupts for the application, which is a major source of overhead. Another alteration, they presented, is to remove clock tick from cores that are dedicated to the HPC application. Additionally, allowing I/O processes to execute on OS processors fully parallelizes the communication of the application. The drawback of this is that not all functionalities of a normal kernel are supported. Without ticks, not all bottom half handlers are processed. This did not allow the application to make progress when using the Ethernet network. They used the PAPI tool to measure the numbers of cache misses with this modified kernel. Without ticks the application experiences no L1 cache misses. In our work, we focus on the influence of the Linux OS scheduler. We measure its influence with perf.

Akhmetova et al. [2] investigate the interplay between task granularity and scheduling overhead. Task-based programming models are a promising approach for HPC applications. The workload is divided into small tasks, which define basic units of computation. The number of tasks is much larger than the number of processors, so there are very few idle cores. These tasks are mapped to the processors by the runtime scheduler. There are many different schedulers that can be chosen. Simpler schedulers have a smaller runtime overhead, but more sophisticated schedulers may increase the application performance by considering the task locality or power efficiency. But this requires more execution time for scheduling which increases the overhead. For the systematic analysis of the impact of the task granularity, they have an algorithm that analyses the directed acyclic graph (DAG) of the application and aggregates it into corresponding coarser-grained tasks. The DAG is generated by Prometheus, a system emulator for task-based applications [12]. The experiments were performed with a system emulator. The optimal granularity depends on the scheduler overhead. It varies between $1.2 * 10^4$ and $10 * 10^4$ cycles. Larger granularity leaves the system idle, and smaller granularity introduces too much scheduling overhead.

Dursun et al. studied the effect of the Linux OS on the execution of parallel applications with Perf [7]. For that, he recorded the tracepoint events of the scheduler during the execution. In the Perf output, there are many threads that interrupt the application. Also,

the OpenMP threads migrate between CPUs. This migration can be prevented by binding the threads to a specific CPU. The analysis of the GNU and Clang compiler shows that the OpenMP scheduling techniques guided and auto do not provide good load balance. They concluded that the influence of the Linux scheduler is greater than the overhead caused by the preemption, context switches, and migration of OpenMP threads. Building on these results, we want to investigate different scheduling methods. Not only the standard OpenMP scheduling methods. We also want to find out what the influence of the measurement tools on the applications is.

3

Measuring the Scheduling overhead of an Parallel Application

3.1 Linux Scheduler

The OS scheduler is responsible to let all threads on a system execute on the processor, according to the scheduling policy [14] [19] [10]. For this, the scheduler can interrupt an executing application thread and let other threads execute. The new thread has to load data to the cache, which removes the cache of the application thread. When the application can execute again, it has to reload this data. So every scheduling decision affects the performance of an application.

Linux uses a complex scheduling framework since the kernel version 2.6.23. The OS scheduler is divided into two components, a set of Scheduling Classes and a Scheduling Core [9]. When deciding which thread can execute next the scheduling class with the highest priority, with runnable threads, is chosen. This scheduling class decides which of its threads based on its scheduling policy. This framework guarantees that no thread with low priority is scheduled if there is a runnable thread in a Scheduling Class with higher priority. It also allows having several scheduling policies for different tasks at the same time. By default, there is a class, with the highest priority, for real-time threads. Normal threads belong to the Complete Fair Scheduler (CFS) class. The completely fair scheduler (CFS) simulates a real multitasking processor by allocating $1/n$ of the total processor time to a thread. Where n the total number of runnable threads is. So the allocated processing time of a thread is lower when there are many other threads requesting processing time. To guarantee that not too many switches deteriorate the performance of all threads, this share of computing time has a lower bound. CFS uses the Nice value of threads to give some threads more computing time. The *target latency* is the interval in which each runnable thread is executed so that no thread has to wait for too long. A smaller target latency results in better interactivity for I/O-bounded processes. If there are too many threads, the allocated timeslice would become too small and the threads would switch too often. To prevent this, CFS has a *minimal granularity*. This is the minimal time that a thread should execute to prevent the switching costs from affecting the performance of the whole system. The default minimal granularity is 1 millisecond. To determine the next thread, the CFS has a red-black-tree

(rbtree) ordered to the runtime of each thread. With this, the next thread is found in the left-most leave of the rbtree. This is the thread that had the least time on the processor. With this choosing the next thread to execute is trivial. To balance the load between CPUs, dynamic load balancing is used. CFS checks the system for load imbalance at regular intervals. If the imbalance is too big CFS moves threads from CPUs with high loads to CPUs with less load.

When the runqueue is empty, the Linux scheduler *idle scheduling class* starts the idle thread. This is a special thread that activates architecture-specific hardware features to save energy. This thread is only executed when nothing else needs execution time.

In this work, we explore the CentOS 7.9 with the Linux kernel version 3.10.0 x86 64.

3.2 Perf

We use the Linux kernel tool `perf` to measure the OS scheduling overhead of the applications [11][22]. Perf was introduced to the Linux kernel version 2.6.31 in 2009. As with the rest of the kernel source code, `perf` is open source. The idea of `perf` was to have a built-in tool to make use of the performance counters of the Linux kernel. It can observe the performance of applications or hardware events. Perf uses events from many parts of the system. Hardware events come from the CPU performance monitoring counters (PMC). PMC depends on the hardware on which the system runs. Typically, it is only possible to record a few PMCs at the same time. They contain among many others CPU cycles and cache misses on all levels. Software events are low-level events that are based on kernel counters like CPU migrations and page faults. Kernel tracepoint events are instrumentation points on the kernel level. They are hard-coded in points of interest in the kernel. Tracepoint events allow tracing high-level behavior of the system, for example, network events, file or disk I/O events, or system calls. These events are grouped into tracepoint libraries, for example, socket events are called "sock", CPU scheduler events "sched", or "kmem" for kernel memory allocation events. Other events are tracepoints for user-level programs. These events are hard-coded into the source code of applications, usually with macros. Many applications can be compiled with the Dtrace flag to support DTrace. The static tracing interface is more stable and easier to use than dynamic tracing. But it is possible to enable dynamic tracing on a system without restarting it.

We used `perf` version 3.10.0 to record the scheduling events during the execution of applications. With `perf` we recorded the scheduling events during the execution of the application. For this, we used the command:

```
$ perf sched record -a -R -o output ./app
```

This command records and saves all scheduling events in a binary file during the execution of an application. It is designed to have a small overhead to make as little impact on the performance of the application as possible. The flag *-a* tells `perf` to collect events on all CPUs on the system. With *-R* we collected the raw sample records from all counters for later analysis. The generated data can be analyzed with several `perf` tools after the application is finished. We used:

```
$ perf sched timehist
```

Timehist outputs a list of every recorded scheduling event. For each event, there is the timestamp and name of the process, the CPU ID on which the event was recorded, and what type of event it was. The last information are the different time measurements for these events. For each event, `perf` reports the *wait time*, *sched delay*, and *execution time*. *Wait time* is the time between a sched-out and the next sched-in event. Also, the time a process waited to wake up, while other processes were executed. For the first recorded event of each process on a CPU, the wait time is zero. There is no time elapsed since the last event of this process. *Sch delay* (scheduler delay) is the time between wake-up and actually executing. This is the scheduler latency, the time the scheduler needs to assign the next thread to a processor. *Runtime* is the time this thread could execute. For an application, it is the time until the next interrupt. Perf shows all time measurements in milliseconds.

```
 1            time    cpu  task name                    wait time  sch delay   run time
 2                         [tid/pid]                       (msec)     (msec)     (msec)
 3 --------------- ------  ------------------------------ ---------  ---------  ---------
 4 18086211.584002 [0013]  <idle>                           0.000      0.000      0.000
 5 18086211.584006 [0013]  rcu_sched[9]                     0.000      0.000      0.003
 6 18086211.585208 [0021]  <idle>                           0.000      0.000      0.000
 7 18086211.585300 [0028]  <idle>                           0.000      0.000      0.000
 8 18086211.585301 [0021]  perf[10909]                      0.000      0.002      0.092
 9 18086211.585304 [0028]  rngd[1908]                       0.000      0.002      0.003
10 18086211.585493 [0032]  perf[10907]                      0.000      0.000      0.000
11 18086211.586007 [0032]  <idle>                           0.000      0.000      0.513
12 18086211.586012 [0032]  rcu_sched[9]                     2.001      0.002      0.004
13 18086211.587604 [0023]  <idle>                           0.000      0.000      0.000
14 18086211.587611 [0021]  <idle>                           0.092      0.000      2.310
```

Figure 3.1: Example for the output of *perf latency*.

With this information, we calculate the overhead of thread migrations, context switches, and CPU idle time. The migration overhead for an application is the sum of all wait times and scheduling delays for each migration event that affects an application thread. The overhead of CPU idle time is the sum of the runtime of all recorded idle threads. The overhead of context switches for the application is the sum of all wait times and scheduling delays of all entries of the application. These are events when the application threads were interrupted to let some other process execute.

3.3 Applications

For our measurements, we selected three applications with different characteristics regarding load imbalance and compute or memory bound.

1. **Mandelbrot** is a simple code that calculates the Mandelbrot-set. This app is extremely compute bound and load imbalanced. The version of Mandelbrot we used is implemented in a time-stepping fashion. It is composed of three main loops with different load imbalance characteristics across time steps: constant, increasing, and decreasing.

2. **STREAM-Triad** is one of the kernels available at the STREAM benchmark [1]. This kernel performs one addition, one multiplication, and one copy operation each step. Therefore, this application is extremely memory bound and load balanced.

3. **SPH-EXA** [6] is a smoothed particle hydrodynamics (SPH) mini-app which can be used to simulate different SPH problems. The problem we approached is named Sedov blast wave. This problem is mixed in terms of memory/compute bound characteristics and it is slightly load imbalanced.

We present the details about the parameters for the applications in the table of factorial experiments **??**

All applications were executed without thread pinning. This gives the OS scheduler the most freedom to choose the best cores for the application threads.

In addition to the executions with one single application, we also investigated the impact of the OS scheduling overhead when two applications are executed on the system at the same time.

3.4 Thread Level Scheduling Techniques

To investigate the relationship between the OS scheduler overhead and scheduling on the thread level, we compare the different thread level scheduling techniques of the OpenMP standard. Additionally, we chose four scheduling techniques that are added to the LLVM's OpenMP runtime library by LB4OMP [13][15] We compare seven thread level scheduling techniques, with different characteristics.

We use all scheduling techniques from the OpenMP standard. The straight forward scheduling technique `static`. This technique has the least overhead to distribute the work. `dynamic,1 (SS)` [16] archives the highest load balance by assigning one iteration for each work request by a thread. This can lead to considerable overhead because it disregards data locality. `guided (GSS)` [18] is a dynamic and non-adaptive scheduling technique implemented in the OpenMP standard. `Static_steal` from the LLVM, works similarly to static. In the beginning, the work is distributed equally, but in the end, threads can steal work from other threads. This lowers the load imbalance. `FAC2` [8] is another dynamic and non-adaptive technique. Both, `GSS` and `FAC2` achieve good load balance and low overhead, by assigning large chunks of work at the beginning of the execution. In the end, this chunk size decreases for a good load balance. With adaptive weighted factoring (`AWF`) [5] and adaptive factoring (`AF`) [4] we have two dynamic and adaptive scheduling techniques from LB4OMP. These two techniques collect information about the executed loop to adapt the chunk size. The measurement and calculation of the chunk size lead to more overhead, and small chunk sizes can lead to loss of data locality.

3.5 Two Applications

We also want to investigate how the scheduler behaves when two applications execute at the same time. For this, we start the applications Mandelbrot and STREAM-Triad at the same time. Each application with ten threads. Every other configuration is the same as with the other experiments sumariced in table **??**. No threads are pinned so that the OS scheduler can migrate the threads to any core. The thread level scheduling technique is the same for both applications that are executed at the same time.

For the execution, perf is mounted only on the application that takes longer, not for both applications. We execute *perf sched record* with the *-a* flag to record events on all CPUs. So the scheduling events of the other application are also recorded and perf is only executed once. We did not investigate whether perf would influence the results if perf is executed twice, once for every application. We made sure that perf is mounted on the application that takes longer.

Since both applications execute with only ten threads, the execution time is larger than in the other experiments where they used twenty threads. We normalize the scheduling overhead by the execution time so that we can compare these results.

3.6 Table of Factorial Experiments

Table 3.1: Design of factorial experiments resulting in a total of 210 experiments.

Factors		**Values**	**Properties**
Applications		Mandelbrot	N = 262,144 — T = 100 — Total loops = 3 — Modified loops = 3
		SPH-EXA Sedov	N = 125,000 — T = 100 — Total loops = 16 — Modified loops = 3
Microbenchmark		STREAM-Triad	N = 2,000,000,000 — T = 400 — Total loops = 1 — Modified loops = 1
Thread-level Scheduling	OpenMP Standard	`static`	Straightforward parallelization
		`guided (GSS), dynamic,1 (SS)`	Dynamic and non-adaptive self-scheduling techniques
	LB4OMP	`FAC2`	
		`AWF_D, AF`	Dynamic and adaptive self-scheduling techniques
		`static_steal`	LLVM implementation of work stealing
Operating System		Linux	CentOS 7.9, Linux kernel version 3.10.0 x86 64
Computing nodes		miniHPC-Xeon	Intel Broadwell E5-2640 v4 (2 sockets, 10 cores each) P=20 cores without hyperthreading,
Metrics		Average application execution time	
		Thread migration overhead %	$avg(sum(wait\ time + sch\ delay))/avg(application\_execution\_time) \times 100$
		Context switches overhead %	$avg(sum(wait\ time + sch\ delay))/avg(application\_execution\_time) \times 100$
		CPUs idle time overhead %	$avg(sum(run\ time))/avg(application\_execution\_time) \times 100$

4
Results

In this chapter, we present the results of our measurements. First, we show that the applications behave as we expect regarding memory or compute boundness. Then we show that the overhead of our measurements with perf does not introduce too much overhead. After that, we present the measurements of the OS scheduling overhead for the different applications and thread level scheduling techniques. In the end, are the results for the measurements with two applications executing at the same time.

4.1 Rofline Model

To show that the applications we chose are indeed memory or compute bound on our system, we show their performance in a roofline model [23] [20]. The roofline model compares the arithmetic intensity, on the x-axis, with the performance, on the y-axis. The arithmetic intensity is the number of floating-point operations per byte loaded to memory. The horizontal line in roofline plots shows the peak floating-point performance of a system. The diagonal line in a roofline plot shows the maximum performance of the memory system for a given operational intensity. These two lines create the roofline. This line shows the maximum performance a system can achieve. Most applications will not achieve this performance. We measured the performance of the system and applications with Likwid [21]. In figure 4.1 we show the roofline model for the applications Mandelbrot, STREAM-Triad, and SPH-EXA on the miniHPC-Broadwell nodes. We see that Mandelbrot is compute bound, and STREAM-Triad is extremely memory bound. SPH-EXA is memory bound, but not as extreme as STREAM-Triad.

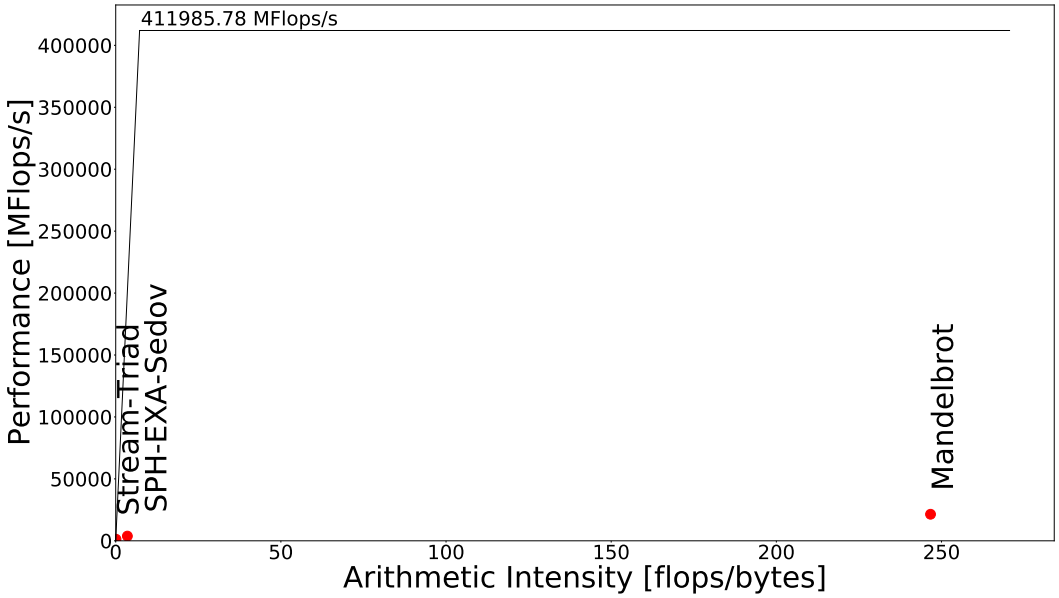

Figure 4.1: Roofline model of the applications on the Intel Broadwell system

4.2 Overhead of Measurement

Every measurement during the execution of an application adds a bit more work. Although perf is designed to introduce only a small overhead we want to quantify this overhead. For this, we compare executions of the applications with and without perf measurements. The configurations are the same as in the table **??** described and each measurement is repeated ten times.

In figure 4.2 we show the execution time of the application Mandelbrot with and without measurements with perf. There are results for all thread level scheduling techniques. We see that the overhead of perf is neglectable.

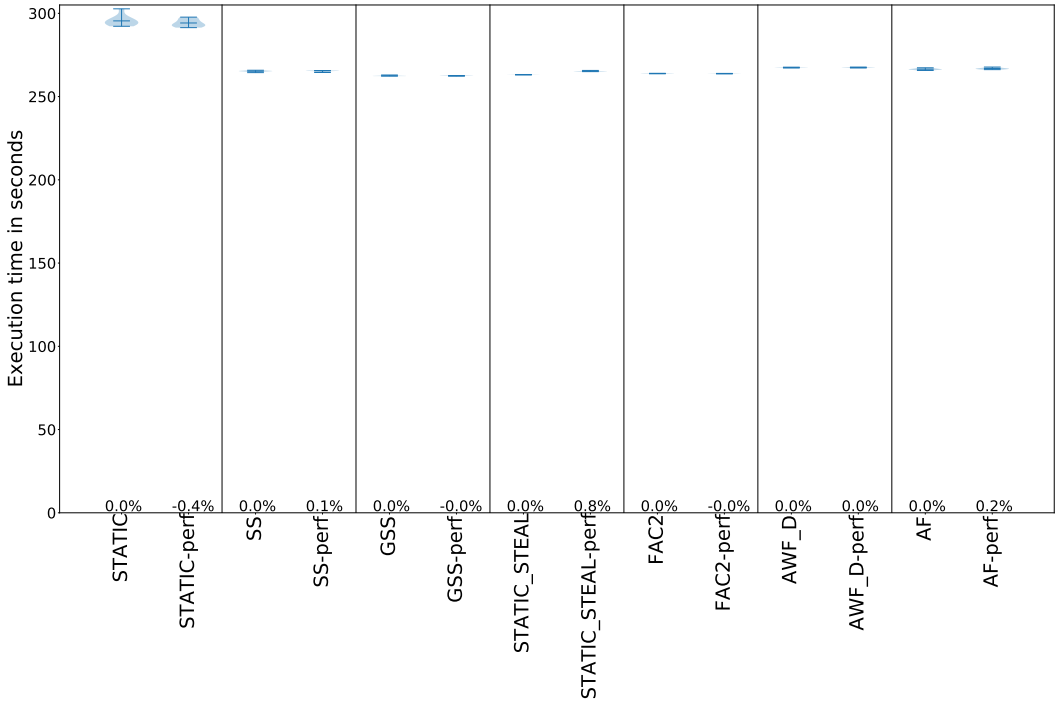

Figure 4.2: Execution time of the application **Mandelbrot**. Without measurements with perf (left), compared to the execution time of the application with the additional overhead that measurements with perf introduce (right). For the different thread level scheduling techniques

The application STREAM-Triad is highly memory bound. So perf has a higher influence on this application because it adds more memory that needs to be written to the disc. This lowers the amount of data that the application can use. So the overhead of STREAM-Triad (see figure 4.3) show that the measurements introduce a bit more overhead than compared to the application Mandelbrot. The different thread level scheduling techniques influence the performance of STREAM-Triad much more than Mandelbrot. Also, the overhead of perf is much higher with some thread level scheduling techniques. For example, `static` shows an overhead of 20%. This is caused by the imbalance that the perf measurements cause. They slow down the data for some threads, which leads to load imbalance. The scheduling technique `static` can not balance the work.

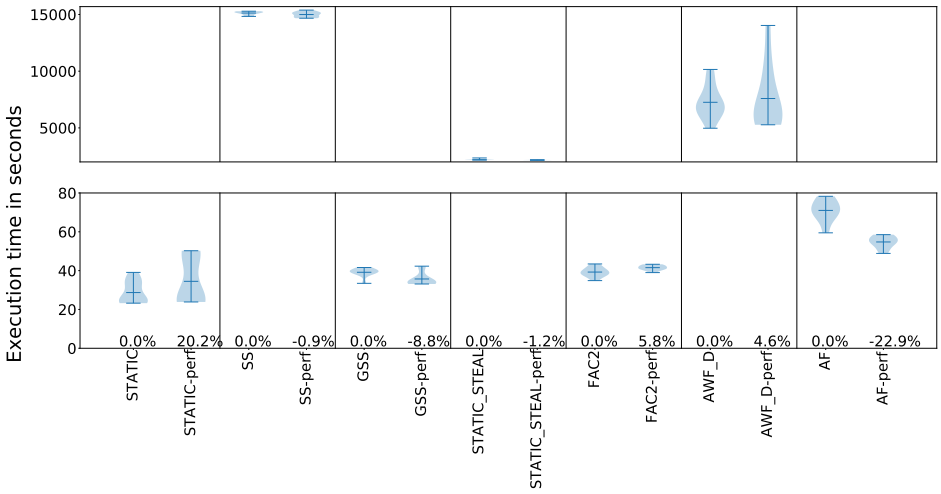

Figure 4.3: Execution time of the application of the measurements with perf, compared to the execution time of the application without any additional measurements.

4.3 Scheduling Overhead

In this section, we present the results for the scheduling overhead measurements. The data is shown in heatmaps. On the x-axis are the different thread level scheduling techniques, and on the y-axis the different applications. For details on how the applications were executed, check table **??**. In each field are two numbers, the upper value is the overhead of the measured events normalized by the execution time of the application. The lower number in brackets is the average execution time of the application. Each experiment was repeated ten times. The scheduling overhead for the different applications is derived with the output of the *perf sched timehist* command. We show the results for the overhead of migration, context switches, and CPU idle time.

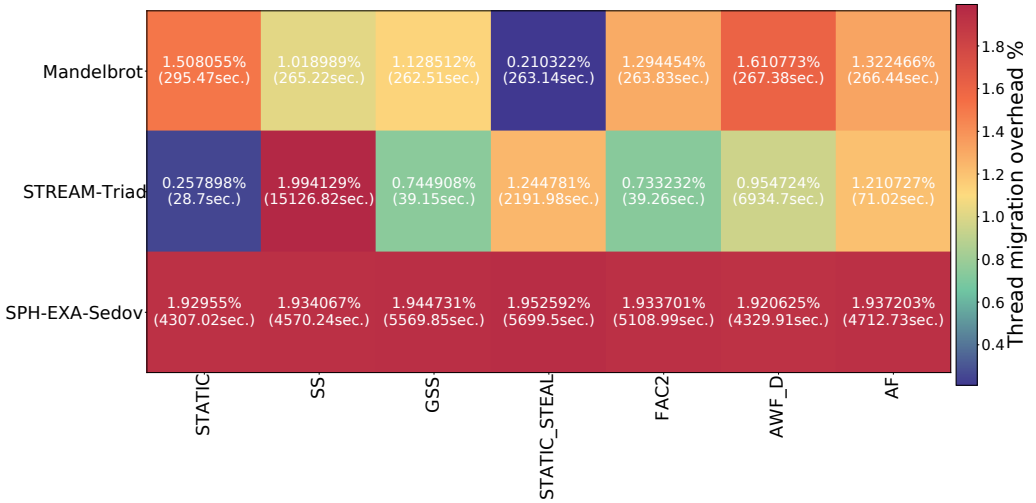

Figure 4.4: Percentage of time spent on thread migration over the execution of the applications. Average execution time of the application in brackets.

4.3.1 Migration Overhead

The migration overhead is the sum of all wait times and scheduling delays of all migration events related to the application threads. In figure 4.4 we see the three applications on the x-axis and the thread level scheduling techniques on the y-axis. The top number in the fields is the overhead, that thread migrations cause. Note that the thread level scheduling techniques have a high influence on the execution time. The migration overhead is normalized by the execution time. For SPH-EXA the overhead of thread migrations is for all thread level scheduling techniques around 1.9%. The scheduling techniques have a high influence on the overhead for the applications Mandelbrot and STREAM-Triad. The differences in the overhead are quite large between the different scheduling techniques. STATIC introduces the least migration overhead to Mandelbrot and `static_steal` the least overhead to STREAM-Triad.

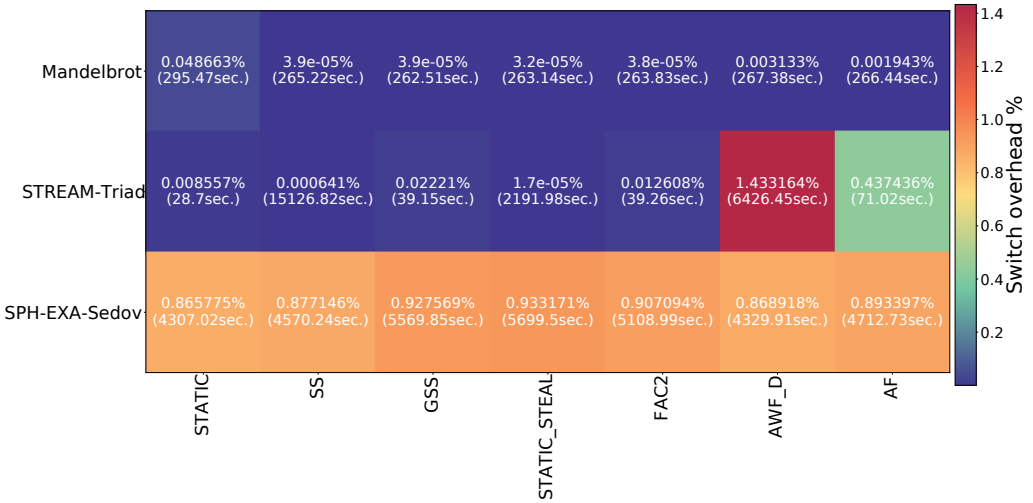

Figure 4.5: Percentage of time spent on context switches over the execution of the applications. Average execution time of the application in brackets.

4.3.2 Context Switch Overhead

The overhead introduced by context switches is very small of Mandelbrot and STREAM-Triad with most scheduling techniques. SPH-EXA experiences much more overhead from context switches. The thread level scheduling techniques have only little influence on the overhead from context switches. Except for `AWF` and `AF` for the application STREAM-Triad.

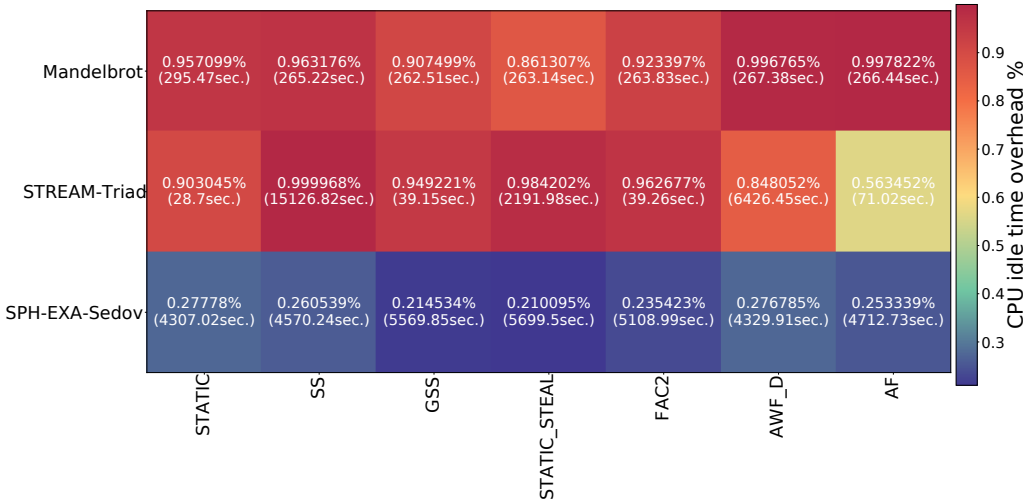

Figure 4.6: Percentage of time spent in CPU idle time over the execution of the applications. Average execution time of the application in brackets.

4.3.3 Idle Time Overhead

The thread level scheduling technique does not influence the overhead of idle times much. A scheduling technique should use all available execution time and leave as little as possible unused. All thread level scheduling techniques achieve this. The difference between the application is much bigger. It is not surprising that STREAM-Triad has a high CPU idle time because the threads have to wait for the data. Mandelbrot has nearly the same overhead in CPU idle time as STREAM-Triad. SPH-EXA has a much lower overhead. A reason for this is that SPH-EXA is much more balanced than Mandelbrot. So there are fewer threads that have to wait on other threads at barriers.

4.4 Scheduling Overhead for Two Applications

In this section, we explore the overhead that scheduling introduces when two applications are executed at the same time. We executed the applications Mandelbrot and STREAM-Triad with ten threads each. All experiments are repeated ten times.

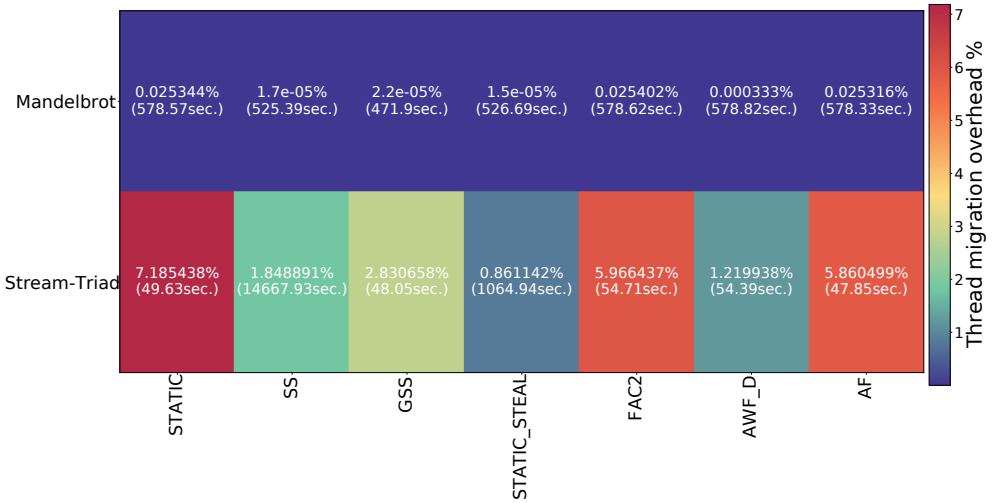

Figure 4.7: Percentage of time spent on thread migration over the execution of the applications. Average execution time of the application in brackets.

We observe a big difference in the overhead for thread migration between the two applications (see figure 4.7). Mandelbrot has a very low overhead for all thread level scheduling techniques. STREAM-Triad has with most scheduling techniques a very high overhead. If we compare this overhead with the execution when the application does not share the system (figure 4.4) we see that the migration overhead for Mandelbrot is much lower with this setting than compared to the executions with twenty threads. For STREAM-Triad the migration overhead with ten threads is much higher. The only exception are the executions with the scheduling technique `static_steal`, which has a little bit lower migration overhead compared to the executions with one application. The execution time for STREAM-Triad is shorter with this configuration and the scheduling techniques `(SS)`, `static_steal`, and `(AWF)`. This is because these scheduling techniques do not consider data locality, which is important for this very memory bound application. So fewer threads actually finish faster.

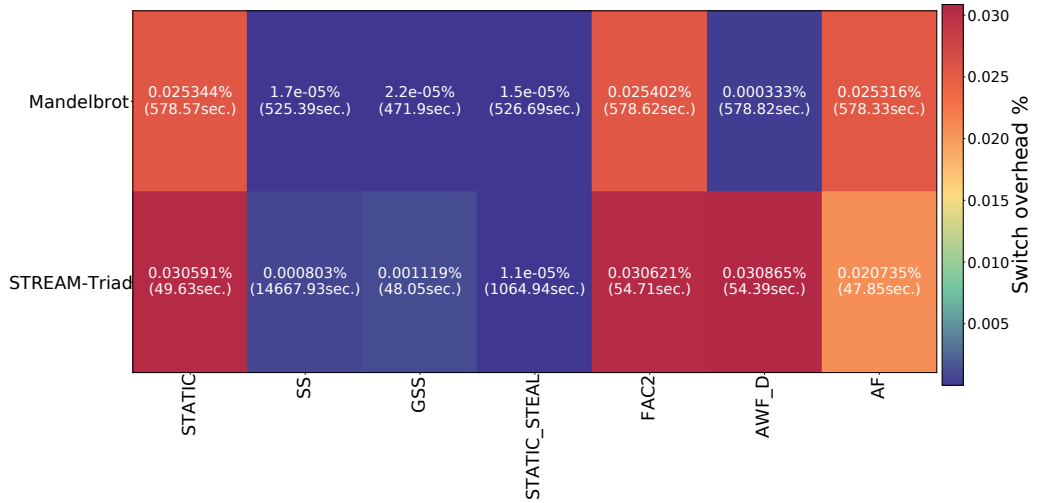

Figure 4.8: Percentage of time spent on context switches over the execution of the applications. Average execution time of the application in brackets.

The overhead of context switches differs widely between the different thread level scheduling techniques. `AWF` only for the application Mandelbrot and `SS`, `GSS`, and `static_steal`, for both applications, have much fewer overhead from context switches than the other scheduling techniques. This difference is really astonishing. We expected to see more context switches for scheduling techniques that lead to load imbalance, as we see for example with `STATIC`. But it is interesting that scheduling techniques that have the same properties, for example, `GSS` and `FAC2`, or for Mandelbrot `AWF` and `AF`, have such different switch overheads.

The application STREAM-Triad has in most cases a bit higher overhead than the application Mandelbrot. This is what we expected because the threads of STREAM-Triad have to wait for data.

When comparing the overhead of context switches for the measurements with two simultaneously executed applications with the results when only one application is executed 4.5, we can clearly see that the thread level scheduling influences the context switch overhead more with two applications executed simultaneously.

5
Discussion and Future Work

Our results show that different aspects of OS level scheduling overhead are influenced more by different parameters. The overhead of context switches and the CPU idle time depends mostly on the application. For some applications, the thread level scheduling techniques introduce more thread migrations.

5.1 Limitations and Future Work with Two Applications

The results for the two applications executed at the same time show that thread level scheduling has a big influence on the OS scheduling overhead. At least for some applications.

The plot for the idle time overhead for the experiment with two applications executed simultaneously is in the appendix (see figure A.1). The problem with these results is that we record all idle events on all CPUs. When the first application finishes, perf records on all CPUs until the second application finishes too. The second application uses only ten threads, which leaves ten CPUs unused. The scheduler can put all background work on these unused CPUs and leave ten cores for the application. But this will most likely result in a lot of idle time for the unused cores.

This issue does not affect the results for thread migration and switch events. We assigned these events to the different applications. We did not assign an idle event to the application that was executed last on the same core. We are not sure if it is correct to assign an idle time to one application. If we would assign an idle event to one application it would not be sufficient to assign the event to the last application that executed on this core. Additionally, it is necessary to make sure that this thread was not migrated to another core. This would be possible with the output of perf. One could argue that both applications are equally responsible for idle time as long as both execute. A better solution to this problem would be to mount perf on the application that executes shorter. Then the idle events would only be recorded when both applications are executing.

We mounted perf to the application with the longer execution time to compare these executions with the execution when only one application is executing. It would be difficult to compare the overhead of the OS scheduler when for one result only a part of the execution is measured. Our comparison between the two different execution is that the number of

threads is different. Once we had single applications with twenty threads and once two applications with ten threads each. We can see the different performances in the execution times.

There are many different ways to explore the behavior of the OS scheduler and different thread level scheduling techniques with more than one application. We only executed the application with ten threads each. It would be interesting to investigate the OS scheduling overhead when for example both application executed with 20 thread, so that these threads really compete for execution time. We executed both applications with the same thread level scheduling technique. It would be interesting to see what happens when two applcations with different thread level scheduling techniques are executed.

5.2 Perf on different Hardware

We tried to execute all measurements on the GPU node on miniHPC. This node has an Intel Xeon Gold 6258R processor with 56 cores. Perf is highly dependent on the hardware in use. On this system, the normal *perf sched record* command does not work correctly. It generates the data but the other perf tools can not analyze it. The cause of this problem is that perf can not find the number of CPUs on the system. We can help perf by specifying on which CPUs to record with an additional flag *–cpu 0-55*. With that perf records the scheduling event on all 56 CPUs on the system.

The timehist command and other analysis tools from perf can work with this data. But there is no name for most of the processes. Instead, there are numbers (see figure 5.1) Not all process names are replaced with numbers but the application threads, in which we are most interested, are replaced. Therefore, we can not calculate the scheduling overhead for the applications the same way as with the results from the Broadwell nodes. The latency command summarizes the scheduling latency for each process. So we have a list of each process that was executed at the same time as the application. Most of these are kworkers and daemons. Besides this also perf is listed. But most names that are human-readable in the output from other systems, are replaced by numbers. These numbers are not process IDs.

We executed the application with 56 threads. Also, the number of these "processes" that are named by numbers is often 56, or very close. So it is possible that only the threads of the application are not listed by their names. If this is the case, it would be possible to treat these numbers all as threads from the application and get the results we want from all events that involve a thread with these numbers. This would only work if the application threads are the only threads that are not listed by their names. If other threads are listed with these numbers too, this approach would mix the scheduling events of the application with these other threads. Then we would measure a similar overhead as in my thesis, where we measured the scheduling overhead during the execution of an application.

Task	Runtime ms	Switches	Average delay ms	Maximum delay ms	Maximum delay at
crond:3699	0.033 ms	1	avg:60000.357 ms	max:60000.357 ms	max at: 11394273.108167 s
acctg:147350	0.560 ms	1	avg:30000.134 ms	max:30000.134 ms	max at: 11394254.222011 s
gmond:129475	1.837 ms	5	avg: 7524.035 ms	max:20001.092 ms	max at: 11394281.922117 s
lsmd:2797	0.061 ms	4	avg: 7507.536 ms	max:30030.139 ms	max at: 11394277.217101 s
kworker/u676:0:138136	0.446 ms	14	avg: 2857.192 ms	max:40000.663 ms	max at: 11394258.817458 s
kworker/u677:1:129040	0.504 ms	16	avg: 1875.070 ms	max:20000.997 ms	max at: 11394268.817520 s
kworker/u673:3:146000	183.729 ms	8	avg: 1436.005 ms	max:11487.989 ms	max at: 11394279.102349 s
:147548:147548	63318.456 ms	32	avg: 1260.603 ms	max:40339.187 ms	max at: 11394294.334202 s
zabbix_agentd:(2)	34.040 ms	155	avg: 354.936 ms	max:22010.674 ms	max at: 11394266.896538 s
:147549:147549	39638.443 ms	133	avg: 354.101 ms	max:22938.127 ms	max at: 11394267.863096 s
:147526:147526	12232.920 ms	220	avg: 336.363 ms	max:23120.100 ms	max at: 11394266.600079 s
in:imjournal:3250	0.719 ms	70	avg: 334.615 ms	max:20720.177 ms	max at: 11394266.604050 s
:147555:147555	9737.337 ms	214	avg: 313.791 ms	max:23189.128 ms	max at: 11394267.685093 s
tuned:3343	1.848 ms	79	avg: 304.105 ms	max:22022.096 ms	max at: 11394267.337051 s
:147527:147527	37680.214 ms	239	avg: 276.078 ms	max:21395.299 ms	max at: 11394266.171296 s
:147540:147540	63108.209 ms	151	avg: 268.567 ms	max:40549.164 ms	max at: 11394294.506194 s
:147518:147518	14466.560 ms	265	avg: 256.174 ms	max:23993.121 ms	max at: 11394268.593083 s
:147523:147523	16226.513 ms	293	avg: 237.201 ms	max:23103.080 ms	max at: 11394267.975081 s
:147521:147521	11048.705 ms	317	avg: 232.781 ms	max:38932.157 ms	max at: 11394283.523147 s
:147524:147524	18190.248 ms	308	avg: 220.431 ms	max:23036.129 ms	max at: 11394267.570088 s
:147533:147533	33166.500 ms	328	avg: 214.925 ms	max:23002.088 ms	max at: 11394267.667073 s
:147537:147537	18823.788 ms	315	avg: 213.518 ms	max:21721.944 ms	max at: 11394266.480914 s
:147541:147541	36026.959 ms	322	avg: 210.048 ms	max:23276.094 ms	max at: 11394267.891078 s
:147514:147514	26928.221 ms	275	avg: 207.334 ms	max:21901.982 ms	max at: 11394266.230969 s
:147543:147543	17393.845 ms	339	avg: 202.300 ms	max:24479.077 ms	max at: 11394267.069099 s

Figure 5.1: Example for the perf output on the GPU node. Most task names are not human readable names.

Bibliography

[1] STREAM Microbenchmark. http://www.cs.virginia.edu/stream/ref.html. Accessed: March 29, 2021.

[2] Dana Akhmetova, Gokcen Kestor, Roberto Gioiosa, Stefano Markidis, and Erwin Laure. On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems. In *2015 IEEE International Conference on Cluster Computing*, pages 428–437. IEEE, 2015.

[3] Hakan Akkan, Michael Lang, and Lorie Liebrock. Understanding and isolating the noise in the linux kernel. *The International journal of high performance computing applications*, 27(2):136–146, 2013.

[4] Ioana Banicescu and Z. Liu. Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes. In *P. of th H. P. C. Symp.*, pages 122–129, 2000.

[5] Ioana Banicescu, Vijay Velusamy, and Johnny Devaprasad. On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. *J. of Clus. Comp.*, pages 215–226, 2003.

[6] Ruben Cabezon, Aurelien Cavelan, Florina Ciorba, Michal Grabarczyk, Danilo Guerrera, David Imbert, Sebastian Keller, Lucio Mayer, Ali Mohammed, Jg Piccinali, Tom Quinn, and Darren Reed. Github repository of the miniapp application SPH-EXA. https://github.com/unibas-dmi-hpc/SPH-EXA_mini-app, (26.01.2022). Commit #26.

[7] Mustafa Dursun. Analysis of openmp applications with linux perf tracepoint events. University of Basel Department of Mathematics and Computer Science High Performance Computing, 2018.

[8] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *J. of Comm.*, pages 90–101, 1992.

[9] Roberto Gioiosa, Sally A McKee, and Mateo Valero. Designing os for hpc applications: Scheduling. In *2010 IEEE International conference on cluster computing*, pages 78–87. IEEE, 2010.

[10] Redha Gouicem. *Thread Scheduling in Multi-core Operating Systems*. PhD thesis, Sorbonne Université, 2020.

[11] Brendan D. Gregg. perf examples. http://www.brendangregg.com/perf.html, February 2022.

[12] Gokcen Kestor, Roberto Gioiosa, and Daniel Chavarrıa-Miranda. Prometheus: scalable and accurate emulation of task-based applications on many-core systems. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 308–317. IEEE, 2015.

[13] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Lb4omp: A dynamic load balancing library for multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):830–841, 2022. doi: 10.1109/TPDS.2021.3107775.

[14] Robert Love. *Linux Kernel Development*. Pearson Education, 2010.

[15] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. unibas-dmi-hpc/LB4OMP: LB4OMP v1.0, 2020. URL https://doi.org/10.5281/zenodo.3872907.

[16] Tang Peiyi and Yew Pen-Chung. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In *P. Intern. C. on Par. Proc.*, pages 528–535, 1986.

[17] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 55–55. IEEE, 2003.

[18] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *J. Trans. on Compu.*, pages 1425–1439, 1987.

[19] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[20] Thomas Gruber Thomas Roehl, Georg Hager. Tutorial: Empirical roofline model. https://github.com/RRZE-HPC/likwid/wiki/Tutorial%3A-Empirical-Roofline-Model, February 2022.

[21] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: Lightweight performance tools. In Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010*, pages 165–175, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-24025-6.

[22] Unknown. Perf wiki. https://perf.wiki.kernel.org/index.php/Main_Page, February 2022.

[23] Samuel Williams, Andrew Waterman, and David A. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures, April 2009. URL https://doi.org/10.1145/1498765.1498785.

A
Appendix

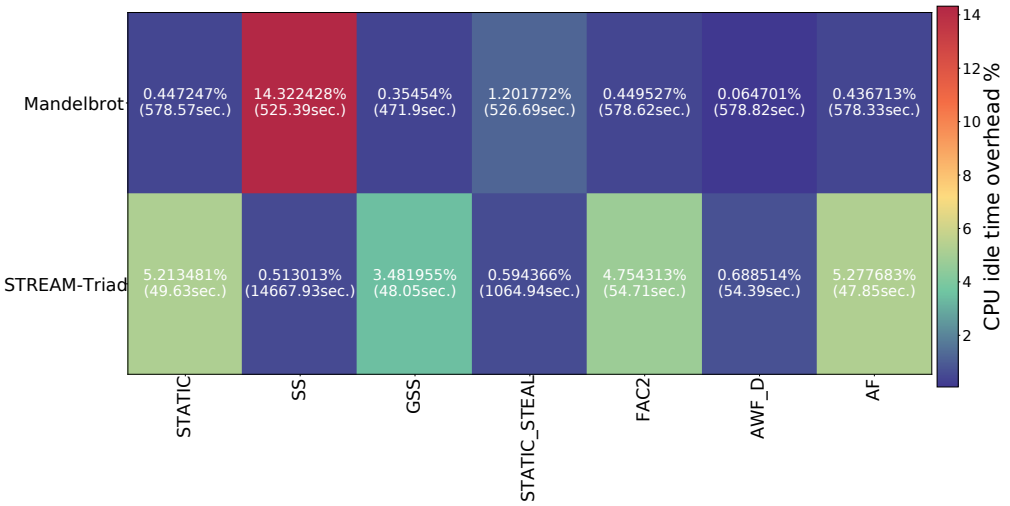

Figure A.1: Results with two applications executing at the same time with ten threads each. Percentage of the execution time, the CPU is idle. Average execution time of the application in brackets. The result is not comparable because perf recorded when one application already finished and the events are not assigned to one specific application.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor

David Kuhn

Matriculation number — Matrikelnummer

16-057-960

Title of work — Titel der Arbeit

Investigation of the Relation Between the Linux Operating System Scheduler and Scheduling Decisions at Thread and Process Level

Type of work — Typ der Arbeit

Master Project

Declaration — Erklärung

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.
Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 30.06.2022

Signature — Unterschrift