University
of Basel

Impact of the Linux Operating System on the Performance of Parallel Applications

Master Thesis

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

26.02.2022

Acknowledgments

I would like to express my sincere gratitude to my advisor Professor Florina M. Ciorba for the opportunity to write the thesis in her research group. Her support and constructive feedback were invaluable for this thesis. I am deeply grateful to my supervisor Jonas H. Müller Korndörfer for his timely, and valuable guidance throughout the whole work. I would also like to thank all the members of the HPC group for their kind help and supportive ideas.

Abstract

Most Operating systems for HPC clusters are based on Linux. The role of the OS scheduler is to assign execution time to processes. The OS scheduler may introduce overhead by enforcing thread migrations or context switches. The role of the OS scheduler is to allocate processing time to all processes executing on a system. For this, the scheduler can swap or stop processes according to the scheduling policies. We investigate if there is a relation between the application thread level scheduling and the overhead that the OS may cause. For that, we use many scheduling techniques at the application thread level. For the measurements, we use the tools PAPI, perf, and Likwid. In our measurements, we investigated the OS overhead on different applications, with three parallelization methods. We used a set of thread scheduling techniques with several thread configurations on different computing nodes. We show that all these aspects are important and can, in some combinations, positively or negatively influence the performance of applications.

Table of Contents

1 Introduction

In the last decades, many cores per node and many nodes per system became normal for high-performance computer clusters. To fully use all these CPUs, parallel applications rely on the Operating System (OS) scheduler to assign their threads to the optimal CPU. Linux uses the completely fair scheduler. This scheduler gives every process an equal share of the processing time. When the time for one process is up, the scheduler interrupts the application and enforces a context switch. To balance the load on different cores, the scheduler can also move threads to a different processor. Threads that wait for data or are idle for other reasons, will be swapped for other threads that request execution time.

All these interruptions by the OS scheduler cause overhead to the executing application. Before resuming to execute, the cache needs to be loaded. This takes some time during which no progress is made. The influence of OS scheduling is not clear.

To investigate this, we used the performance measurement tools PAPI, perf, and Likwid. Perf provides good recordings of OS scheduling events. With PAPI and Likwid detailed measurements of the important part of an application are possible. With these tools, we recorded context switches, thread migrations, idle time as well as cache and memory performance. Furthermore, we investigated the overhead that these tools introduce.

For this master thesis, we analyzed how the OS noise affects different parallel applications. We analyze memory- and compute-bound applications with different degrees of load imbalance. This is important because different applications react differently to other thread scheduling techniques or thread configurations. We investigated how the parallelization methods OpenMP, MPI, and hybrid MPI and OpenMP react to OS noise. To investigate how different OpenMP thread scheduling techniques behave regarding OS noise, we used many scheduling techniques provided by LB4OMP and Auto4OMP. We also examined the influence the thread scheduling techniques have on the performance of applications. For this, we leave some cores idle, so that the OS can use them.

We show that all these factors are important for an efficient execution of a parallel application. In some combinations the measured influence is high. Other combinations result in relatively low overhead.

In chapter 2 we explain the Linux scheduler with its policy and introduce the performance measurement tools we used. In chapter 3 we discuss related work. We explain how

we performed our experiments in chapter 4. We present the results of our measurement in chapter 5 We end with the conclusion and future work in 6.

2
Background

In this chapter, we introduce the basics of the Linux OS and its scheduling system 2.1. We identify potential sources of noise that the OS generates 2.1.2 . Then we present the parallelization techniques MPI and OpenMP 2.2. We end this chapter with the performance measurement tools we to used for our experiments 2.3.

The terms tread, process, program, and task are often used ambiguously in literature. Therefore, we use the same definition as Gouicem et al. in this work [16]. A *thread* is the smallest entity. It consists of an instruction pointer, a stack pointer, and registers. A *process* consists of memory mapping, file descriptors, sockets, etc. A process contains at least one thread. Threads from the same process share the process resources. This makes communication among these threads easy. Communication between processes relies on the inter-process communication mechanisms of the OS. A *program* or *application* consists of at least one process. All processes work together to fulfill the goal of the program. Figure 2.1 illustrates a multi-process program. In literature, the term *task* is used for threads and processes depending on the context. Because of this, we will not use this term in this work.

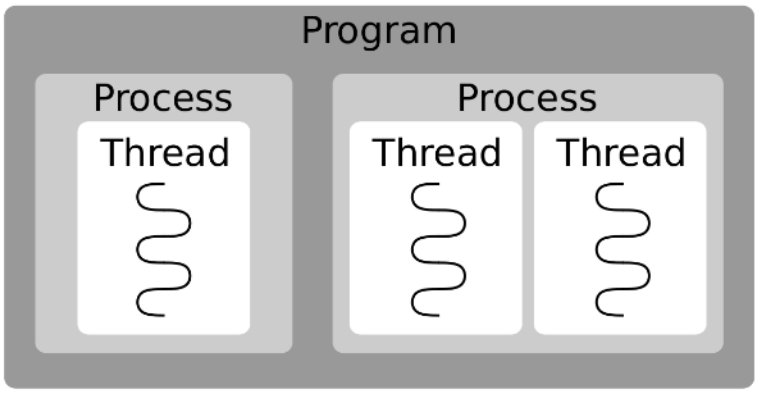


Figure 2.1: Illustration of multi-processing and multi-threading. This figure is taken from Gouciem et al. [16].

2.1 Linux Operating System

Operating Systems (OS) are an independent layer between hardware and applications [25] [32][15][13][16]. They allow the execution of the same code on different hardware. The applications do not have direct access to the hardware. Applications communicate to the OS through system calls. The OS communicates to the hardware with control registers. These three layers are independent of each other except for these interfaces.

There are two types of kernels. Microkernels introduce less OS noise but do not offer all OS services. Services like process management, memory allocation, or virtual file systems have to be implemented in userspace or are absent. Monolithic kernels, like Linux, deliver all these services but they introduce more overhead. Microkernels often support only a few hardware architectures.

Linux uses, as most Operating systems today, preemptive multitasking. This means that it can interrupt an executing process and let other processes run. An alternative to preemptive multitasking is cooperative multitasking where a process has to voluntarily free the processor. Today this approach is rarely used because the OS can not enforce a context switch. If a process executes for a long time and does not interrupt voluntarily, the whole system is blocked.

2.1.1 Linux Operating System Scheduler

For Linux and any multitasking operating system, the scheduler is a key component for correct and efficient work [25][32]. The OS scheduler administrates the scarce resource of execution time on the processor. It decides which thread can execute when and for how long. On an imaginary computing system where only one thread is executing, this single thread could occupy the processor until it finished. But on real systems, this is not normal. Often there are a lot of different threads that compete for resources like execution time, memory, network bandwidth, etc. It is not possible to execute more than one thread at a time on one processor.

Multitasking OS address this problem by interleaving the execution of several threads on one processor. This gives the impression that several threads are executed at the same time. This also allows for parallel applications where several threads of the same application are executed at the same time. The scheduler can interrupt executing threads and let another thread run. This allows threads to react to events like user input or loaded data in a short time. Also, no idle thread should occupy the processor when another runnable thread is waiting. In this case, the scheduler should preempt the idle thread and let another one execute.

The kernel uses a hardware clock to generate periodic interrupts, this time interval is also called tick. These ticks call a kernel routine that handles OS activities. For example, the OS daemons and the scheduler can execute during these events. The OS also looks for pending signals or network data that arrived. During these ticks, the scheduler can decide whether the execution time of the currently running thread is used up and start a context switch to another thread.

On multiprocessor systems, the scheduler must not only decide which thread can run

when but also where. The scheduler has to decide on which processor a thread should execute. Resources that are shared between the cores, for example, memory, become an issue. Not all processors can access these resources at the same time. Also, threads from the same process may profit from the loaded cache if they run on the same CPU. The scheduler has to account for all these problems. There are two main scheduler options for multi-core systems, partitioned scheduling and global scheduling [7]. In partitioned scheduling, every processor has its own ready queue. Therefore, a CPU may be idle while on other CPUs there are waiting threads. Load balancing is harder with partitioned scheduling. The optimal placement of threads among the processors is an NP-Hard problem. Global schedulers have one queue for the entire system. This makes load balancing easier. But a global scheduler has to account for the cost of migration from one processor to another.

For parallel applications on multi-core systems load imbalance is an issue. The scheduler has to use all processors equally. For this load balancing is necessary. When the scheduler decides to move one process to an idle CPU. The moved process has to reload the cache which delays the restart on the new CPU. Each interrupt and migration introduces direct overhead. When no process is executing no progress is made. But also the indirect overhead is introduced because the cache needs to be loaded.

2.1.1.1 Scheduling Policies

The scheduling policy determines the behavior of the whole system. Depending on the use of the system, the scheduler can prefer one sort of process over others. There are different types of processes with different needs. Input-/ Output- (I/O) bounded processes often need very short computing time until they have to wait again for the next event. This event can be loading and storing data to the disk, waiting for a network packet to arrive, or waiting for the next user input. When this event occurs they have to react fast to optimally use the bandwidth to the disk or network, or to give the user the requested output. On the other hand processor bounded processes execute for long times until they have to wait for an I/O event or barriers. The scheduler policy has to balance the needs of the two process types. I/O bounded processes need a short latency and processor bounded processes need high throughput. These requirements are in conflict, with each other. If the scheduler preempts processes often then each process must not wait for long until it can execute again. This improves the latency. But each context switch costs time to load memory to the cache. This time is not spent executing a process, which reduces the throughput that processor bounded processes aim for. To address all needs the scheduler has to guarantee low latency and high throughput at the same time, which is not possible. Like other UNIX systems, Linux prefers I/O- bounded processes over processor bounded processes.

To determine how long a thread should execute on a processor, some schedulers calculate a timeslice. This is the allocated time a thread can execute. The timeslice has a lower bound, to prevent performance degeneration of the system, because each swap has an overhead. While a thread executes its timeslice is reduced. When it reaches zero the scheduler preempts the thread and starts another one. If a thread preempts voluntarily, for example, to wait for an I/O event, it can save up its timeslice for later.

Not all threads are equally important. The scheduler should prefer some threads over

others. To differ between such threads, each thread is assigned a priority. Threads with higher priority should execute before threads with lower priority. At the same time, low priority threads should not starve, should get some processing time. In Linux, there are two different priority values. A *Nice value* between -20 and +19. A large Nice value corresponds to lower priority. The second priority range is the *real-time priority*, normally between 0 and 99. A large real-time priority value corresponds to a higher priority.

2.1.1.2 Linux Scheduler Framework

Linux kernel 2.6.23 introduced a scheduler framework which is divided into two components, a set of Scheduling Classes and a Scheduling Core [15]. Scheduling Classes select the next thread to execute according to their policy. These classes are objects that provide scheduling policies for threads they hold. Each processor can have several Scheduling Classes which contain a list of runnable threads. By default, there is a class, with the highest priority, for real-time threads. Normal threads belong to the Complete Fair Scheduler (CFS) class, CFS is explained in the next section. Idle tasks are in a separate class. The Scheduler Core is called when the scheduler looks for the next thread. The Scheduler Core looks for the Scheduler Class with the highest priority which has runnable threads in its queue. This Scheduler Class can then decide which of its thread can execute next. 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.

2.1.1.3 Complete Fair Scheduler

Before the Linux kernel 2.6, there were several problems with the scheduler [19]. In Linux version 2.4 - 2.6 several different scheduling algorithms were tried, but they had severe drawbacks. For example, the O(n)-scheduler, the name is derived from its complexity O(n), has an overhead that grows too big, if there are too many threads. At a high load, a significant part of the computation power is spent only to schedule the next threads.

The completely fair scheduler (CFS) was introduced to Linux by Ingo Molnar in 2007 to the Linux kernel version 2.6.23. It has a complexity of O(1). CFS simulates a real multitasking processor. The CFS does not allocate a hard timeslice to a thread. Instead, each thread receives a share of the processor. The scheduler allocates $1/n$ of the total processor time to a thread. Where n the total number of runnable threads is. The allocated processing time depends on the load of the system. If there are more threads, the allocated time reduces accordingly. CFS uses the Nice value as weight. The share of a low-priority thread is smaller than that of a higher priority thread. To simulate a real multi-thread system, this timeslice should be infinitely small. But to prevent too many switches this share of computing time has a lower bound. To guarantee that in a given time interval each thread can execute, CFS has a *target latency*. 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 that the switching costs affect the performance of the whole system. The default minimal granularity is 1 millisecond. Each thread has a *virtual runtime*. This is the execution time normalized by the number of runnable threads. 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. CFS uses dynamic load balancing. In regular time intervals, CFS moves thread from CPUs with high loads to CPUs with less load.

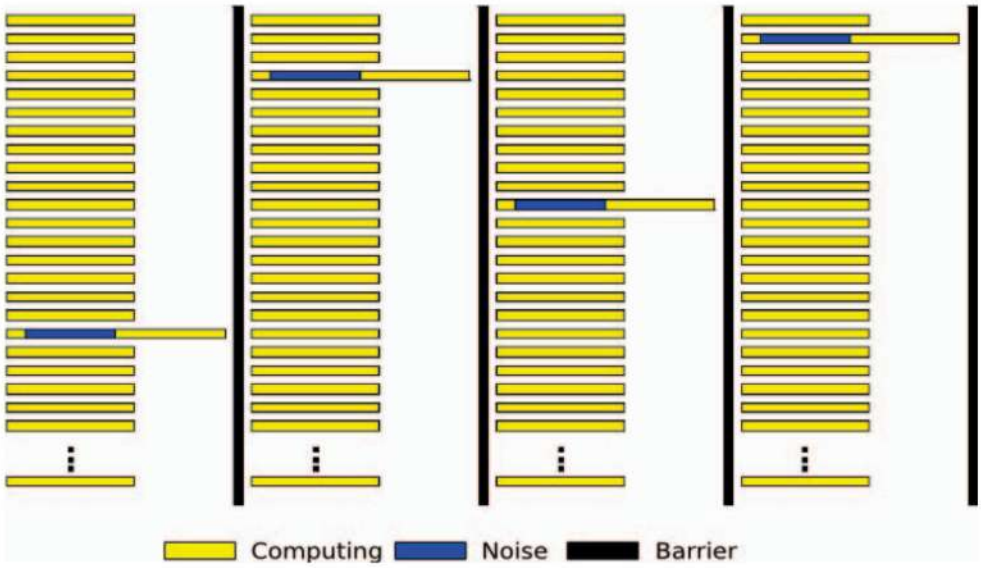
2.1.2 Operating System noise

General Operating Systems are designed for high interactivity and the scheduler is responsible for time-sharing computing time on the CPU [2][15]. The OS should provide a smooth user experience. There are potentially thousands of threads that share very few processors. This requires frequent interrupts to change the executing thread. The frequency of these swaps determines the interactivity, of a system. For higher interactivity, a higher frequency is needed. Linux uses normally 250 to 1000 Hz for desktop systems and 100 Hz for server systems.

On HPC clusters there is normally only one application running. Interrupts occur because of OS processes like daemons and timer ticks [37]. For an HPC application, any activity not directly related to itself is overhead. It is shown that the most overhead comes from timer interrupts [4]. During each tick the kernel does some work, for example, daemons can execute, the scheduler decides whether the current thread can execute again or another thread should run. All this additional work delays the application. It introduces direct overhead by occupying the processor for some time and not letting the application execute. It also generates indirect overhead by loading its own data to the cache, which replaces the data of the application.

Processor bounded processes are more affected by this overhead than I/O bounded processes. I/O bounded processes often need a short execution time. This is often shorter than the frequency, in which timer ticks occur. Although this overhead occurs for all applications, parallel applications are more affected by this overhead than sequential ones. Interrupts occur on all processors at different times. This leads to load imbalance between the different threads of the application. In the worst case, one thread is interrupted before every synchronization (see figure 2.2).

These interrupts do not only generate overhead but are also responsible for performance variation. The execution time for the same application can vary depending on how often and when it was interrupted. Also, CPU migration, sparked by the scheduler, affects the performance of applications.



(a) Possible effect of OS noise

Figure 2.2: In the worst case, a parallel application is slowed down by noise between every synchronization barrier. All threads have to wait for the one thread that is slowed down by noise. A lot of time, threads are idle and waste executing time. This is an example of why parallel applications suffer heavily from OS noise and how noise leads to load imbalance. This figure is taken from Betti et al. [4]

2.1.2.1 Context Switch

A Context Switch is the process of stopping an executing thread to let another thread run. This is necessary that several threads can share one processor in a multitasking operating system. For a context switch, the register and memory map of the old thread is stored in the memory so that it can be restored later and continue its work. The state of the next thread has to be loaded from memory before it can start. In Linux, the switch between threads from the same process is not as expensive as switching between different processes. This is because threads from the same process share the same virtual memory map. A context switch can be triggered, when the scheduler preempts a thread to let another thread execute. It is also possible that a thread voluntarily frees the processor. For example as a result of an interrupt for disc storage access or to synchronize with other threads. Overhead of context switches occur because it takes time to store and load the data from memory. This time is not spent executing. Additionally, the cache has to be loaded for the new thread. This takes some time until the new thread can execute with the best performance.

2.1.2.2 Timer Interrupt

Normally the processor is shared by multiple threads. To share the processing resource, the OS has to interrupt a thread regularly to let the scheduler decide who can run next [2]. Preemptive Operating systems achieve this with on-chip timers which interrupt the CPU at regular intervals. Linux uses normally 100 Hz for server systems. At each of these

intervals, several OS-level bookkeeping operations can take place. Also, the scheduler may trigger a context switch if the executing thread used its time up. This timer interrupts takes place many times a second even if they are not necessary. I/O bounded processes are not highly affected by timer interrupts. Parallel HPC applications suffer from these interrupts because some threads experience less overhead than others. This leads to load imbalance and additional waiting time at the next barrier (see figure 2.2). Cache pollution is also an issue because the kernel operations load some data to the cache which affects the performance of the application afterward.

2.2 Parallel Programming

In this section, we introduce the parallelization techniques Message Passing Interface (MPI) and Open Multi-Processing (OpenMP). MPI allows parallelization of processes. OpenMP is used for thread-level parallelization. MPI often needs a restructure of the code, which OpenMP does not need. However, MPI delivers higher performance than OpenMP. Often these two approaches are combined on a computer cluster for better performance than using just one. In this case, MPI is normally used for parallelism between nodes and OpenMP for parallelism inside a node on CPUs.

2.2.1 MPI

The Message Passing Interface (MPI) is designed for application on MIMD distributed memory systems [12][34]. It is the de-facto standard communication protocol among processes. The first MPI version was released in 1994, the latest version MPI 4.0 in 2021. The MPI standard defines the semantic and syntax of library routines. These routines can be used in C, C++, and Fortran. There are several open-source MPI implementations available. The goals for MPI are high performance, scalability, and portability. The communication protocol of MPI supports point-to-point as well as collective communication.

2.2.2 OpenMP

Open Multi-Processing (OpenMP) is an application programming interface (API) that enables shared-memory multiprocessing [8]. OpenMP supports the programming languages C, C++, and Fortran and is supported on most operating systems, including Linux, Windows, macOS, Solaris, AIX, and HP-UX. There exist many compilers for OpenMP. The first version of OpenMP was introduced by the OpenMP Architecture Review Board (ARB) in 1997 [5]. In November 2020 the current version 5.1 was released. OpenMP consists of compiler directives and library routines to express shared-memory parallelism. It is a simple interface to develop multi-threaded applications. In OpenMP a primary thread forks sub-threads. The code section that should be executed in parallel is marked with a compiler directive. The system shares the work among threads. After the execution of the parallel section, the sub-threads join the primary thread. Because the threads share the memory space, each thread has direct access to the memory of each other thread. To manage correct execution it is possible to declare certain parts of the memory private for each thread. OpenMP supports

task- and data parallelism.

2.2.2.1 OpenMP Standard Scheduling Techniques

The OpenMP standard specifies three loop scheduling techniques: static, dynamic, and guided [26][28]. The used scheduling technique is determined by the programmer. It can influence the performance of the application because each scheduling technique has different performance and overhead characteristics. The scheduling technique provides a hint for how iterations of the corresponding OpenMP loop should be assigned to the individual threads. Static is the basic scheduling technique. It divides the work into equal parts. Each thread has to do the same number of loop iterations. This scheduling technique has the smallest overhead but it can lead to load imbalance between the threads. The scheduling technique dynamic assigns some loop iterations to each thread. When a thread finished its iteration it requests more work until all iterations are assigned to a thread. This method introduces some overhead to the execution time because the work allocation is not done at compile time. Guided scheduling is a self-scheduling method similar to dynamic. The difference is the size of the chunks. In the beginning, the chunk size is large. The chunk size is proportional to the number of unassigned chunks and the number of threads. The chunk size decreases exponentially during the execution to reduce the overhead during the execution.

2.2.2.2 LB4OMP and Auto4OMP

The standard loop scheduling techniques of OpenMP do not deliver the best performance. Other scheduling methods are better but it is not easy to find out which technique to use. LB4OMP is a dynamic load balancing library and extends LLVM OpenMP RTL version 8.0 [24][22]. It implements 14 dynamic and some adaptive scheduling algorithms which add more scheduling options to OpenMP. There are also features for performance measurements of the performance of specific loops to analyze the load balancing and loop scheduling. Thread execution time, which reports the execution time per loop, allows to detect load imbalance. For performance analysis, there is also a calculated chunk size for each thread. LB4OMP allows a fair comparison between different scheduling techniques.

The scheduling techniques, that are implemented in LB4OMP and we used in our experiments are practical variant of factoring FAC2, adaptive factoring AF and static_steal. FAC2 is a dynamic but non adaptive self-scheduling technique. AF is a dynamic and adaptive technique.

Since dynamic and adaptive loop scheduling adds overhead, the authors measured the overhead of the different scheduling techniques of LB4OMP. The best combination of scheduling techniques depends on the application and system but it outperforms a single scheduling technique in most cases. Often adaptive scheduling techniques outperform the non-adaptive. Other experiments show that most scheduling techniques achieve good load balance but this results not always in better performance because the scheduling technique introduces additional overhead. The chunk parameter also impacts the performance. Too small chunk sizes introduce a higher scheduling overhead while too large chunks result in load imbalance. The best chunk parameter always improves the performance. But to find this

value, extensive experimentation is needed for each loop and system. Therefore, dynamically adaptive scheduling techniques promise better performance without manual tuning.

Auto4OMP extends the OpenMP runtime library with improved scheduling algorithm selection methods [23]. Auto4OMP does not require a careful selection of the best scheduling algorithm and chunk size. It replaces the scheduling option auto of OpenMP. In Auto4OMP there are four scheduling algorithm selection methods. *RandomSel* selects the scheduling algorithm randomly. *ExhaustiveSel* selects the best performing scheduling algorithm. *BinarySel* improves the selection time of ExhaustiveSel with binary search. For BinarySel the scheduling algorithms have to be ordered by their introduced overhead. *ExpertSel* uses fuzzy logic and expert rules to select the best-performing scheduling algorithm. They also consider *GAC*, the schedule(auto) algorithm from LLVM OpenMP runtime library. Auto4OMP can change the used scheduling algorithm during the execution. Expert chunk of Auto4OMP selects the chunk size based on the number of loop iteration and the number of threads. To low chunk size result in more scheduling overhead, to large chunk size result in more load imbalance. A scheduling algorithm has to satisfy the following properties for Auto4OMP. A scheduling algorithm should not need user input. To reduce overhead the algorithm should use lightweight synchronization. The last requirement is that the algorithm should not need time-stepping loops to adapt to application performance. The performance analysis shows that BinarySel and RandomSel can cause severe overhead because they may choose a non-ideal scheduling algorithm. ExpertSel and ExhaustiveSel improve the application performance. Also, the expert chunk size improves the performance of applications.

2.3 Performance Measurement Tools

In this section, we present the performance measurement tools we used for our experiments. We explain how we used these tools in chapter 4. To understand the behavior of OS noise, we use tools that make the actions of the computing system observable [29]. The measurement tools interrupt the applications to record events or count hardware counters.

There are two types of performance measurement tools, profiling, and tracing. Profiling updates summary statistics of events. This has a relatively low impact because it saves only a few data to a profile data file. Tracing records log of timestamped events and their associated attributes. This has a higher overhead but allows a more detailed analysis of the application.

The tools we used are Perf, PAPI, and Likwid. These are mostly profiling tools, but it is also possible to trace an application with them. We only used the profiling measurements. All three tools are often used to record the performance of applications. It is not easy to measure the performance of the scheduler or other OS components [16]. Perf is the only tool of these three that is designed to help developers of the OS improve the performance of their code. The measurement conducted during the execution, should not affect the performance of the application. The measurement tool should introduce as little overhead as possible. All three tools are designed to keep their influence low.

The Linux kernel provides several ways to understand the behavior of applications [16]. Besides Perf, there are other tools. Ftrace is a tracing tool for kernel tracepoint events.

These are hard-coded events in the kernel code. They allow among other things to find out the time that the kernel spends in different functions. The procfs pseudo-file system provides aggregated values or snapshots of the current state of the system. Ftrace is a tracing tool so we did not use it. In the preparation of this master thesis, we planned to use Pin. We did not use it because it does not provide any additional use that Perf Papi and Likwid do not provide.

2.3.1 PAPI

The Performance Application Programming Interface (PAPI) provides platform and OS independent access to hardware performance counters [33][9]. PAPI supports a wide range of different performance counters on many parts of a computer system, for example, CPU, GPU, memory, power, network interface cards, and many more. These counters can observe different events. These counters are small sets of registers that count events. Examples for predefined events are cache coherence, cycle and instruction counts, pipeline status, and many more.

The PAPI project originated at the University of Tennessee's Innovative Computing Laboratory. Many additions and improvements are still published [38]. The latest version PAPI 6.0.0 was published in March 2020.

PAPI has two interfaces to the hardware counters. A high-level interface for simpler measurements. It allows the user to start and stop recording counters. The low-level interface that manages hardware events. Some of which may be machine-specific. PAPI also provides access to native events. Because PAPI is used inside the code, it is possible to record the performance of a small part of an application, not the whole execution. This allows the analysis of the most important loops separately from the rest. Another advantage of PAPI is that the user can decide what to do with the measurements. For simple testing, it is sufficient to plot the measured values. It is also possible to store the data directly to a file.

2.3.2 Perf

Perf is a Linux kernel tools for performance analysis [11][30][17][40]. Perf was introduced to the Linux kernel version 2.6.31 in 2009. As the rest of the kernel 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 is also possible to use Perf as a tracing tool. Perf is a tool to observe the performance of applications or a system.

Most events can be monitored via the Perf command in the terminal. Some features are not supported by Perf commands but are accessible with the FTrace interface. 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 PMC 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 hardcoded in points of

interest in the kernel. They 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 hardcoded 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.

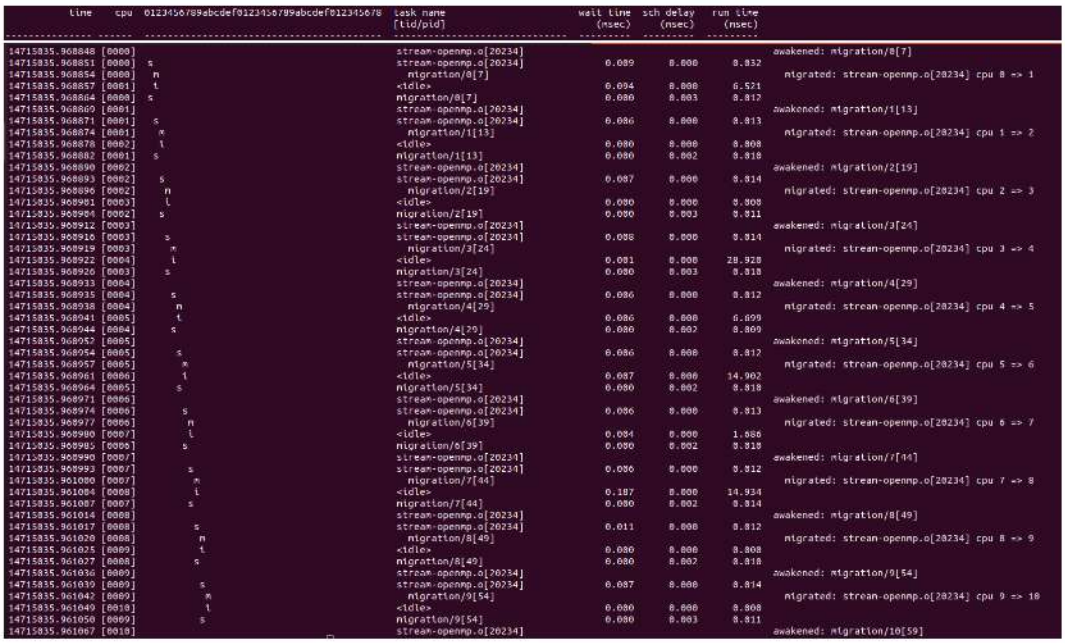


Figure 2.3: An output for the command perf timehist with the application stream, executed on Broadwell

Figure 2.3 shows the output of the command *perf timehist* for the application Stream on Broadwell. This is not from the very beginning, there were other events before the shown events. This output shows how the threads are distributed among the cores. The first column of the perf timehist output is the timestamp. The first awake of stream in this example happened at 14715035.960848. In this execution, the first timestamp is 14715035.782999 and the last 14715070.928144. So the total execution took 35.145145 seconds. This is from the first recorded event to the last. Perf recorded the whole execution of the application. In this time is the overhead of perf included. Perf wakes up several times to store the recorded events. The second column is the CPU ID. Then there is a CPU visualization. This visualizes where the events occurred. The fourth entry is the task name and the task ID (TID or PID) After that are the wait time, sch delay (scheduler delay), and run time. The data we are mostly interested in. The last entry is some additional information about the events.

As an example, we look at how a thread of the application Stream is migrated to CPU 3. We see at 14715035.960890 that on processor 2, the wakeup for the migration process happens. Then at 14715035.960893, 3 ms later, processor 2 begins the context

switch and stops the application. The application was executed for 14 ms. The next event at 14715035.960896 shows the migration from CPU 2 to CPU 3. This is followed by a relatively long idle time on CPU 3. This time is needed to load data. The next event is again on CPU 2 at 14715035.960904. There the migration process is terminated. Perf reports a scheduler delay of 3 ms and a runtime of 11 ms. The scheduler delay is the time between the wakeup of the migration process and the termination of Stream. The runtime is of the migration is between the termination of Stream and the termination of the migration.

2.3.3 Likwid

Likwid is a command-line tool for Linux and supports many architectures [36] [39]. It provides several sub tools. They are designed to address four problems. *Likwid-topology* probes the hardware thread, cache, and NUMA topology. It shows how processors share the cache hierarchy and the cache sizes, how processor IDs are mapped to the CPU cores. Also, details about the NUMA domains and memory sizes are shown. This is an easy way to see how the system is built. It is also possible to use this information inside an application code. To optimize the execution to different systems. *Likwid-perfctr* is a tool for hardware performance counters. In wrapper mode, it can be used to measure the whole execution of an application. With flags of the marker API, Likwid measures the performance between two points in the application code. There are some predefined event sets with the most common measurements. It is also possible to create custom event sets. *Likwid-pin* allows to pin multithreaded applications on the command line. It is possible to pin the threads to specific CPUs with the process ID. Other options include pinning to nodes, sockets, cache groups, or memory domains. Likwid-pin supports all threading models that are based on POSIX threads. It also works for MPI and OpenMP hybrid parallelism. *Likwid-bench* is a benchmarking platform to measure the performance of the system.

In our measurements we used the *likwid-perfctr* commands. This measures the counter specified by the -g flag. Likwid records the counters of the CPU to which the application is pinned with the *likwid-pin* tool. The output shows some information about the system on which the measurement was executed. Then follows the output of the application. The last part is the results of the measurement. Depending on the counter group, the data for every single CPU is shown. First are the counters for the events then the derived metrics. For all counter groups at the end is a summary with the averages, sum, minimum, and maximum of the measured metrics. In figure 2.4 is the output of a measurement with Likwid. The application was executed on Broadwell with 10 CPUs. The output from Likwid shows the event counters that Likwid recorded for every CPU that was used. The second table is a summary of these events. The third table shows the derived metrics for every CPU and the last table is the summary of these metrics.

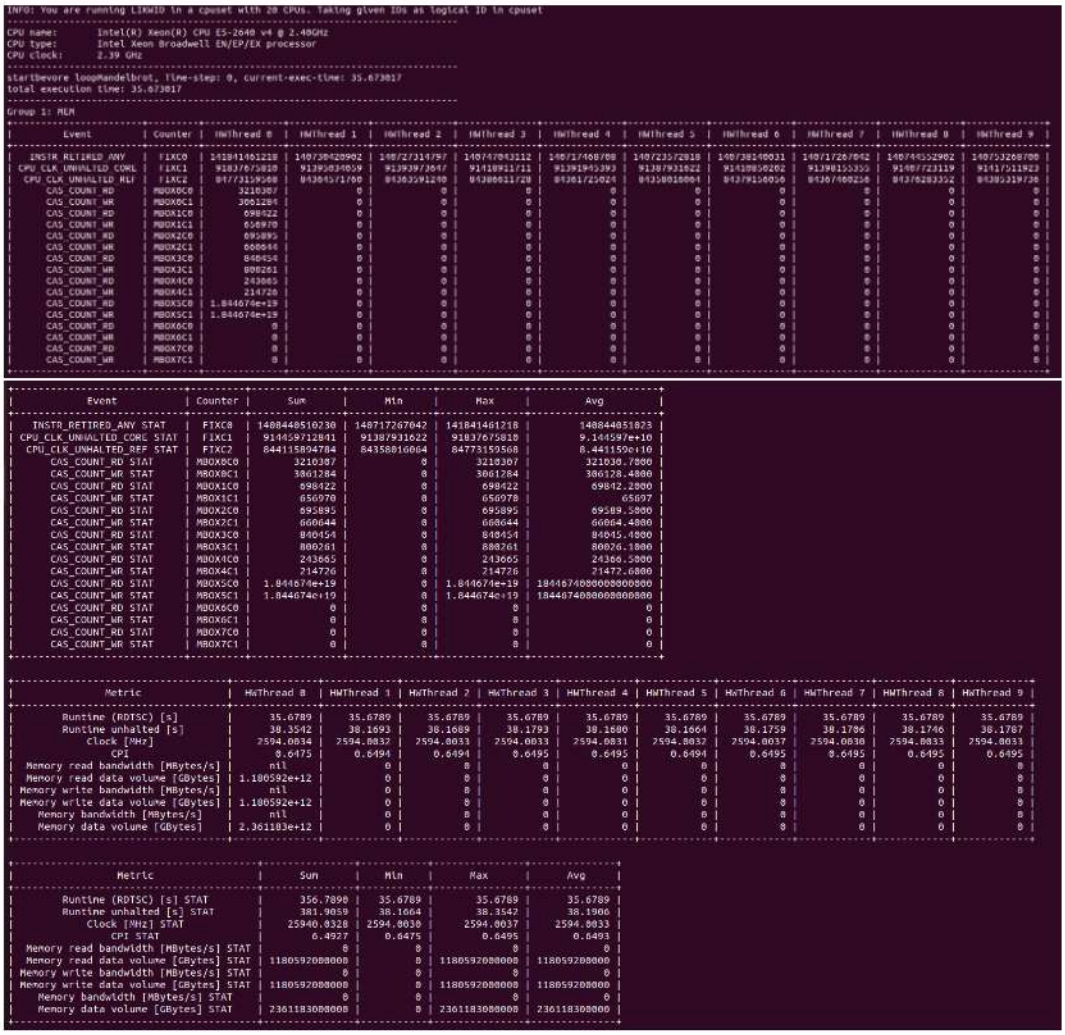


Figure 2.4: Output for the command likwid-perfctr for the counter group *MEM*, with the application Mandelbrot, executed on Broadwell with 10 CPUs.

3
Related Work

In this chapter, we present many papers that are related to Linux scheduler improvements and observation. First, we introduce several papers that present an improvement to the Linux scheduler. Then we have a look at other performance measurements. In the end, we discuss what we will do differently.

Betti et al. [4] introduced CAOS (Cluster Advanced Operating System). This is an extension to the Linux kernel, which addresses the problem of temporal synchronization in HPC clusters. CAOS replaces the local timers on nodes with NetTick. NetTick is a global timekeeping architecture, in which network signals replace the local time devices. The master node sends a heartbeat signal to all nodes. This message is treated as a timer interrupt. This allows the cluster to synchronize the activities on all nodes, see figure 2.2 where this synchronization does not happen. Normally each node has its own time device. The different devices may not exactly be synchronized.

Betti et al. showed in several experiments, that CAOS can reduce the OS noise generated by timer interrupts and the scheduler events. Timer interrupts generate a high-frequency noise and are the main source of OS noise. The local timer raises an interrupt, which the kernel executes. Often there are no pending operations to execute. NetTick reduces this noise. Network tasklets generate also OS noise, this is less frequent but larger. NetTick can only prevent the noise from timer interrupts. The performance of the HPC application is not reduced by the removal of the timer interrupts. Another source of noise are OS daemons. The OS may schedule a daemon instead of the HPC application. This delays the computing on different nodes. NetSched can schedule these background operations during the synchronization phase of the HPC application. For their measurements, they used Fixed Time Quanta (FTQ). FTQ measures the numbers of basic operations done in a fixed time quantum. This benchmark samples the number of basic operations that were performed in a time interval. The difference to the theoretical maximum number of basic operations is due to other activities that were not performed by the benchmark. They report that timer interrupts are indeed the main source of overhead for parallel applications. Also, other programs like daemon cause significant overhead. They show that NetTicks improve the performance of different clusters.

Gioiosa et al. [15] present High-Performance Linux (HPL) to address the problems of

the standard Linux kernel with modern HPC clusters. HPL is based on Linux version 2.6.34. It is designed to improve the performance of HPC applications on clusters by reducing the noise that the OS generates. It also reduces the variability of the performance of parallel applications. The Linux OS introduces some overhead. This overhead is not synchronous on all nodes which introduces performance variation. The scalability of many cores is affected by performance variation if the impact of this overhead increases with the number of cores. Gioiosa et al. use Perf to measure the impact that the Linux OS has on the performance of HPC applications. They report that the execution time increases with the number of context switches and CPU migrations. As in [4] they report that the main sources for overhead in an HPC application are timer interrupts and scheduler events.

To address these problems Gioiosa et al. [15] introduce a new HPC scheduling class between the Real-Time and CFS classes from the standard Linux scheduler. Because the scheduler will not schedule any task from a lower scheduling class if there is any runnable task in a higher class. HPC tasks from the HPC scheduler are not interrupted by tasks from CFS. This could not be achieved by simply giving HPC tasks a higher priority in CFS because the dynamic priority decreases while the process executes. This dynamic priority can change every tick. Therefore, CFS may preempt an HPC task. A higher priority does not guarantee that a process will not be preempted. It is also not good enough to schedule HPC tasks with the real-time scheduler. In this case, the scheduler would not select tasks from CFS until all tasks in the RT scheduler finished, which would improve the performance variability a bit. But process preemption is not eliminated completely. Load balancing can force process migration. The Real-Time scheduler has more problems with load balancing than CFS. If a CPU has no runnable real-time task because all tasks scheduled for this CPU are idle, it tries to pull another real-time task from another CPU. Another scenario is when the numbers of real-time tasks are odd or do not map well to the numbers of nodes, the lead will always be imbalanced. This triggers a lot of task migrations to balance the load which degrades the performance of the whole system.

HPL addresses these issues. It prefers HPC tasks over other processes but performs no load balancing if this is not required for performance improvements. HPL schedules non-HPC tasks synchronously when the HPC application does not use the CPU. They measure the numbers of CPU migrations and context switches with perf. In their experiments, HPL reduces the scheduler overhead severely. This reduces the OS noise and improves the performance variability of the HPC applications. Gioiosa et al. focus on the scheduler influence. They do not address micro-noise from local timer interrupts. HPL use NetSched [4] to reduce periodic timer interrupts.

Jones [20] describes a kernel scheduling algorithm that improves the scalability of an HPC system with many nodes. The performance of parallel applications degenerates with OS jitter. This is generated by different OS activities that run during the execution of an HPC application. If these interrupts happen at random on the different nodes, then the threads have to wait longer on synchronization barriers. This impacts the scalability of an OS. The scalability of an OS describes its ability to support a parallel application without introducing scaling issues that come with larger systems. A cascading effect occurs when a slower process impacts all other processes. Often the OS can slow down a single

process, all other processes have to wait for that slower one at the next barrier. Jones addresses this problem with coordinated scheduling. The OS processes are scheduled at the same time among different nodes. This increases the time when all HPC processes can make progress at the same time without interruptions. The amount of non-application threads execute stays the same, but their impact on the performance of the HPC application is reduced. Coordinated scheduling can be dynamically turned on and off with a system call. During coordinated scheduling, a single OS instance manages all processes. Jones performed measurements on a benchmark. The results show a significant improvement in the performance of the benchmark as well as the variability of the results.

The problem of increasing system noise by larger clusters was also analyzed by Tsafir et al. [37]. They provide a probabilistic argument that under certain conditions the delay OS noises introduce is linearly proportional to the cluster size. For their measurements, they used micro benchmarking to estimate the time to execute an empty loop many times. This yields the granularity of the application. But this benchmark uses very little memory which results in very optimistic values. For kernel profiling, they used the kernel logger KLogger. They compare the noise that is introduced by different schedulers to make sure that the noise is not introduced by system daemons. The main source of fine-grained noise is system interrupts. Also, the frequency of these interrupts has an high influence on the performance variability. They experimented with different tick rates and show that lower tick rates reduce the variability. This variance is mostly due to the indirect overhead by cache misses.

To show that cache misses are the main reason for performance variability the performed measurements with a disabled cache. When adjusting for ticks and network interrupts there is no variability left. They conclude that the performance variability is caused by ticks and network interrupts and cache misses they cause. On high-performance clusters, there is often dedicated hardware, that deals with network interrupts so that the performance of HPC application is not affected by the network communication. As Petrini et al. [27] they draw a connection between the computation granularity and the overhead. When the noise frequency is much smaller than the granularity the overhead becomes smaller. Also, the hardware has an impact on the overhead. On several different systems, the amount of L1 and L2 cache misses differs. This can be influenced by the size of the cache or memory and bus speed. Also interesting is that the frequency of the ticks has a high influence on L1 cache misses but not on L2 cache misses. Tsafir et al. argue that removing ticks promises better results than synchronizing ticks. Because synchronization among nodes has again additional overhead. A sufficient alternative to timers would be smart timers. Smart timers combine accurate timing with reduced overhead by aggregating nearby events and avoiding unnecessary ticks.

Gouicem et al. [16] show three ways to improve schedulers. To ease the development of new schedulers they developed a domain-specific language (DSL) Ipanema. With their DSL they developed schedulers that have smaller footprints and have comparable performance to the current CFS. For the analysis of the newly developed scheduler, they also implemented a set of monitoring and visualization tools. These tools use the given Linux kernel events as well as additional events. They show that modern processors suffer from frequency inversion.

On modern CPUs, dynamic voltage and frequency scaling can lead to busy cores that run with lower frequency than idle cores. This behavior is due to the long frequency transition latency of the processor and the scheduler that does not correctly account for the frequency of the processors. The third contribution is a feature model of schedulers, which allows the implementation of modular schedulers. There are also methodologies to evaluate these schedulers. With that, they extract a scheduler frame, a set of features that influence a given application. Which they used to design application-specific schedulers automatically.

Petrini et al. [27] describe how they improved the effective performance of the supercomputer ASCI Q. ASCI Q [31] was in 2003 the second in the list of top 500 supercomputers. 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 to measure how it performs. For this purpose they used SAGE, an application paralyzed with MPI. To gain insight into the performance of an application they used micro benchmarking. This measures the performance of a part of the application. With software simulation of the physical system, they examined questions that they could not do on the system due to time constraints of ASCI Q. Example for this are different cluster configurations. They also used analytical modeling to predict the expected performance of an application on hypothetical machines. 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. This is because the OS noise does not interfere with the computation of the application. This delay influences the performance of all nodes when they wait for the slowest node at the next barrier (See figure 2.2).

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. With more experiments, they could rule out that computational noise on the processor is the source of the overhead, but they discovered a pattern in the noise on node-level. Not all applications are affected the same by each noise frequency. Fine-grained applications are more affected by fine-grained noise. Coarse-grained applications are more affected by low-frequency noise. Applications that do communicate less frequently are less affected by high-frequency noise because they become co-scheduled. On the other hand, fine-grained applications are more affected 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.

In contrast to [4][15][20][37][16][27] we did not introduce a new feature to the Linux scheduler. We focused on the analysis of the noise the OS scheduler and other OS activities generate and how this affects the performance of parallel applications.

Akkan et al. [2] review measurements to reduce interruptions to the HPC application with 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 for the HPC application. For example, scheduling accounting and possible preemption of the executing task, global kernel time updates, expired timers are executed bottom half handlers.

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 [27]. 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 they used ftrace. To improve the performance of the HPC application they modified the Linux kernel. The tickless Linux moves all tasks that are not related to the HPC application to dedicated OS CPUs. They remove clock tick from cores that are dedicated to the HPC application.

Akkan et al. [2] isolate the application from OS jitter with dedicated cores that execute OS tasks. The reduction in interrupts increases the performance of the HPC application. 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. They state that most drawbacks that come without clock ticks on application cores can be prevented by allowing the bottom half processing on OS cores. 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.

Akhmetova et al. [1] investigate the interplay between task granularity and scheduling overhead. Task-based programming models is a promising approach for HPC application. 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 [21]. 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.

Gioiosa et al. [14] describe the methods to analyze and evaluate the impact of system calls. They use benchmarks to identify the impact of system events. With the estimated time the benchmark should theoretically take, and the measured time they calculated the overhead, noise introduced. With the profiling tool OProfile, they find the source of relevant events. Not all noise sources have the same impact. Some interrupting functions are called more frequently than others. For example, the timer interrupt handler is one of the most frequent events. On Symmetric multiprocessing (SMP) systems there are two timers, the global timer and the local timer for each CPU. Each is called many times per second.

Dursun et al. studied the effect of the Linux OS on the execution of parallel applications with Perf [10]. 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.

In this master thesis, we investigated how the Linux OS affects parallel applications. This work builds on [2][1][14][10] but we investigate how different parallelization- and scheduling techniques react to OS noise. For the measurements, we use different tools and different applications.

4 Methodology

In this chapter we explain how we performed the measurements. A summary of our measurements is in table 5.1. In section 4.1 we present the applications and explain why we have chosen them. We introduce the computing node in 4.2 and also show how the arithmetic intensity for the applications changes for different systems. In section 4.3 we explain the different thread configurations for the experiments. We show how we used Likwid and Perf for our experiments in section 4.4. We end this chapter with the measurements of the overhead that is introduced by the measurement tools. We show these results in section 4.4.3.

4.1 Applications

For our experiments, we used four applications. The goal was to investigate applications with different properties. Mandelbrot calculates the Mandelbrot set. This is computationally intensive and therefore, Mandelbrot is compute-bound. The load of the different threads is imbalanced. The application Stream loads a lot of data and does very little computation with it. We only used the kernel triad. So the loaded data is only used for one computation. Stream is memory-bound. Merge is a combination of Mandelbrot and Stream. In each step, it executes Mandelbrot and Stream. This simulates a real application where there are alternating phases of computation and loading data. We did not use this application for all experiments. The last application is SPH_EXA [6]. It simulates how fluids behave under complex physical conditions. SPH_EXA is computational demanding. We chose it as an example for real applications that load and store a lot of data and also are computational demanding. With our configuration SPH_EXA is less imbalanced than Mandelbrot. On some computing systems SPH_EXA is compute-bound on others memory-bound. In section 4.2 we show on which computing systems these applications are memory- and computation-bound.

We want to investigate how the OS influences application with different properties. So we have applications that are highly imbalanced, as well as balanced ones. We also have compute- and memory bound applications.

4.2 Computing Nodes

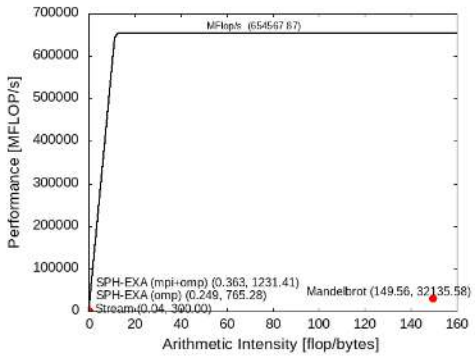
We conducted all experiments on miniHPC. There are different nodes that we used. The Broadwell nodes have an Intel Xeon Broadwell EN/EP/EX processor with two sockets and 10 cores per socket. There is only one thread per core. The L1 cache has 32kB, L2 256kB, and L3 25MB The Cascadelake node has an Intel Cascadelake SP processor with two sockets and 28 cores per socket. Each core has two threads. So there are 112 hardware threads in total. The L1 cache has 32kB, L2 1MB, and L3 38 MB Broadwell and Cascadelake have the same NUMA topology with one NUMA domain for each socket. There is also a GPU on the Cascadelake node, but we did not use it for this work. The KNL nodes have an Intel Xeon Phi (Knights Landing) (Co)Processor with only one socket, but 64 cores with 4 threads. So in total 256 hardware threads. The L1 cache has 32kb and L2 1MB. KNL has no L3 cache. There are two NUMA domains, both with all processors.

To measure if the applications we use are memory-bound or compute-bound we use the roofline model [41] [35]. 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. The roofline is different for every system. The horizontal line in roofline plots shows the peak floating-point performance of a system. Although it would be possible to look these values up in the manuals provided by the producer, we measured the maximum achieved floating-point instructions per second with benchmarks. For this, we used the benchmarks provided by Likwid. The performance of any kernel can not be higher than the roofline, because this is the limit that the hardware can achieve. 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. The formula for this roofline is:

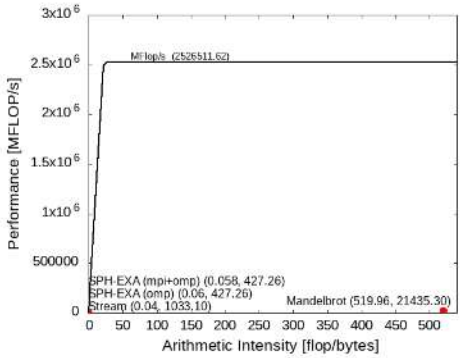
$$Max.Performance = \begin{cases} PeakPerformance \\ PeakMemoryBandwidth \times OperationalIntensity \end{cases}$$

The point where the two lines intersect is called the ridge point. The ridge point indicates the operational intensity that an application needs to be compute-bound. Applications with higher operational intensity than the ridge point are compute-bound. Applications with lower operational intensity are memory-bound. Memory bounded applications can not reach the maximum performance because they have to wait for data.

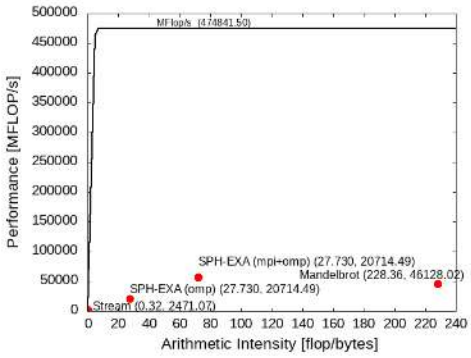
To verify that the applications we chose are indeed memory- respectively compute-bound. we measure the operational intensity and performance on the different systems for the applications. In figure 4.1 we show the results for the applications Mandelbrot, Stream, and SPH_EXA on all three nodes. As expected the application Mandelbrot is compute-bound and Stream is memory-bound on all systems. On Broadwell and Cascadelake nodes the application SPH_EXA is memory-bound. On KNL nodes SPH_EXA is compute-bound. This highlights that the same application can behave differently on other computing systems.



(a) Roofline Model for the Broadwell nodes.



(b) Roofline Model for the Cascadelake nodes.



(c) Roofline Model for the KNL nodes.

Figure 4.1: Each plot shows the roofline model for one computing system with the applications Stream, Mandelbrot, and SPH_EXA (once parallelized only with OpenMP and once with MPI and OpenMP).

4.2.1 Roofline Plot for SPH_EXA Kernels

To understand the bahaviour of SPH_EXA bether, we analyzed the individual kernels. The results are shown in figure 4.2. All kernels of SPH_EXA are compute-bound. But the arithmetic intensity differs widely between the kernels. The highest arithmetic intensity was measured for the kernel sph::computeMomentumAndEnergyIAD. The kernels domain.update, domain.synchronizeHalos, domain.buildTree, sph::computeTimestep,sph::computePositions, and sph::computeTotalEnergy have too similar operational intensity and performance, so that they appear as one point.

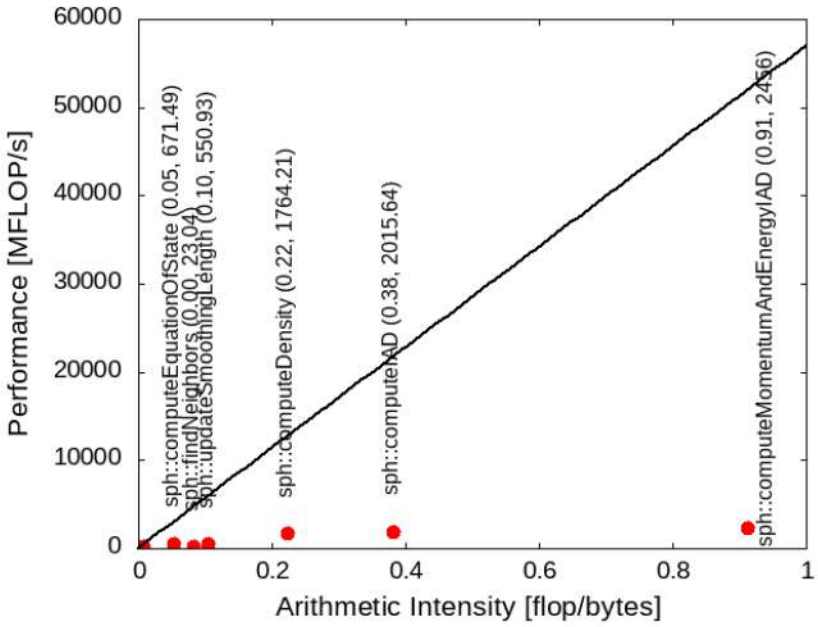


Figure 4.2: In this graph, we show the performance of the different kernels of SPH_EXA on a Broadwell node.

4.3 Thread Configurations

In our experiments, we want to investigate if the placement of OpenMP threads affects the application. According to Petrini et al. [27] an application performs better if some CPU cores are left idle. The idea is that these idle cores can be used by the OS. The scheduler does not need to interrupt a thread from the application. All OS processes can be scheduled on the unused cores and do not interrupt the application. This should reduce the number of interrupts for the application. Fewer interrupts lead to better performance which may make up for the lost computing power. Note that we executed our measurements only on one node. These nodes have more cores. ASCI Q had four cores per node. We used Broadwell nodes with 20 cores, Cascadelake nodes with 56 nodes, and KNL nodes with 64 cores. So the performance loss for the application is not so big.

To analyze the behavior with idle cores, we investigate different thread configurations. The baseline is to use all available cores. This is what is normally done, to execute HPC applications. The application makes use of the full available computation power of a node. To make sure, that the scheduler does not move threads between the cores, we use OpenMP pinning. Then we have measurements with one idle core. So the application can use all cores except one. The third measurements were conducted with two idle cores. On Broadwell and Cascadelake nodes there was one idle core on each socket. KNL nodes have only one socket. So we selected two cores in different cache groups. With the command *likwid-topology* it is easy to see which cores belong to which cache group. The last configuration is to leave one socket idle. This will probably never lead to a shorter execution but we wanted to see if an idle socket reduces to OS overhead because there are cache groups that are not used by the application. On KNL we used half of the cores with half of the cache groups.

4.4 Performance Measurements

To measure the impact of the operating system we used three tools PAPI, perf, and Likwid. We already introduced the background of these tools in section 2.3. For most of our experiments, we used perf and Likwid. We did not use PAPI that much, because Likwid used with the marker API can perform the same kind of detailed measurement that we planned to do with PAPI. Also because the counter group is defined at execution time in the Likwid command it is also easier to measure different counters than with PAPI. With PAPI everything is defined at compile time. In order to change the counters for the measurement, it is necessary to change the the code and recompile it.

PAPI uses events and Likwid uses counter groups. The hardware counters that these tools use depend on the hardware on which they execute. The nodes have different counters available. So not every measurement is available on every system. For example, on Cascadelake, only a few metrics for the L1 cache are available. Perf is built into the Linux OS. The measurements for which we used it did not depend on the system. Therefore, we could use perf in the same way on all three nodes.

4.4.1 Measurements with Perf

We recorded thread migration, context switches, and idle time events with perf [17]. For this, We used the command *perf sched record* to register all events listed in the design of factorial experiments. With the flag *-a* we recorded events on all CPUs in the system. With *-R* we collected the raw sample records from all counters for later analysis. This analysis was done with *perf sched timehist* and *perf sched latency*. The record command dumps all events in a binary file for later analysis. Timehist shows the individual schedule events. For each event it displays wait time, sch delay, and runtime. All measurements are in milliseconds. *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 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. *Runtime* is the time needed for this event. For an application it is the time it could execute until the next interrupt.

We recorded this data for different events: *thread migration*, *context switches*, and *idle time*. These events are initiated by the scheduler. Migration is the movement of a thread from one processor to another. During a context switch, the running process is stopped and frees the processor. Another process can start executing on this processor. The idle time is a time interval in which no computation takes place. The CPU is unused and this time is wasted.

4.4.2 Measurements with Likwid Counters

For the memory and cache measurements, we used Likwid. On Broadwell we used the counter group *ICACHE*, *L2*, *L2Cache*, *L3*, *L3Cache*, *TLB_DATA*, and *TLB_INST*. On Cascadelake, ICACHE, for L1 cache counters, is not available and we could not replace it with

a custom-made counter group. KNL does not have an L3 cache, so these counters are not available on this system. With these counter groups, we measured the metrics listed in the design of factorial experiments. For these measurements, we used the *likwid-perfctr* command. These commands require as input the counter group, the application with its parameters, and thread pin configurations. Pinning is done the same way as with the *likwid-pin* tool. Pinning is necessary when using Likwid because Likwid needs to know where the application executes.

The Likwid counter groups have several metrics, we observed. The cache request rate, miss rate, and miss ratio are the same for all three cache levels. The *cache request rate* is calculated by the number of Cache requests divided by the number of instructions. So this is the number of data access per instructions. It shows how much data is required by an application. The *cache miss rate* is the fraction of cache misses by the number of instructions. This gives the measure of how often data needs to be loaded from higher cache levels. *Cache miss ratio* is the fraction of Cache misses to the number of cache accesses. This indicates how many cache accesses are needed to load data from memory or higher cache levels. The cache miss rate is often determined by the algorithm. The cache miss ratio can be optimized with better cache reuse.

The *L2 cache bandwidth* is computed by the number of cache lines loaded from L2 to the L1 data cache and the writebacks from the L1 data cache to L2 cache. The bandwidth is measured in MBytes/s. The *L2 data volume* is the data volume transferred between L2 and L1. Data volume is measured in GBytes. The *L3 cache bandwidth* and *L3 data volume* are computed analogously for data transfer between L3 and L2. The counters *L3 load bandwidth*, *L3 load data volume*, *L3 evict bandwidth*, and *L3 evict data volume* differ between data loaded from L3 to L2 cache and data that is released from L2 to L3. Cache eviction is a procedure where data from the cache is released. Likwid counts only the cache counters from cache groups that belong to the assigned cores. Cores not specified in the pinning instruction are not considered. If several cores have access to the same cache, and only one of these cores is used by Likwid, then these cache counters are measured.

Memory data volume is the sum of *Memory read data volume* and *Memory write data volume*, *Memory bandwidth* is the sum of *Memory read bandwidth* and *Memory write bandwidth*. Likwid can measure the memory only per socket, not for individual cores. As with the cache data volume, the memory data volume is measured in GBytes and the memory data bandwidth in MBytes/s. Likwid differs between the total amount of bandwidth and data volume and the read and writes to the memory.

The translation lookaside buffer (TLB) stores the translation of virtual memory addresses to physical memory addresses [3]. TLB reduces the time to access memory. Some systems have several layers of TLBs, similar to the cache. We measure only the lowest TLB layer. Often the TLB are separated for data- and instruction addresses. When a process does not find an entry for searched memory, a page fault exception is raised. This exception is normally handled by the OS. *L1 DTLB load misses*, *L1 DTLB store misses*, and *L1 ITLB misses* is the number of TLB misses during the measurement. *L1 DTLB load miss rate*, *L1 DTLB store miss rate*, and *L1 ITLB miss rate* is the rate of TLB misses per instructions. *L1 DTLB load miss duration*, *L1 DTLB store miss duration*, and *L1 ITLB miss duration*

measure the time in CPU cycles to find the memory address. For these counters, Likwid distinguishes between the data TLB (DTLB) and instruction TLB (ITLB). For DTLB on Broadwell and Cascadelake, there is also differed between misses that occur loading and storing reading data.

4.4.3 Overhead of Measurement

To estimate what impact our measurements have on the performance of parallel applications, we measured the execution time of applications while using the different tools. We use the tools perf, PAPI, and Likwid. We compare the execution time with measurements to the execution time without any measurements. We call this native execution later on. This native execution has no additional overhead besides the original application. With these measurements, we do not compare the overhead of the different tools because we do not measure the same things. It is hard to measure the exact same things, with different tools.

The following graphs show the execution time (on the y-axis) for the applications Mandelbrot and stream. They are executed with different scheduling techniques, with and without expert chunk parameters. For each of these parameters, we have measurements for native executions, without any additional measurements, as well as executions with PAPI, perf, and Likwid measurements. The red dotted line is the median execution time without measurements. This is our baseline. On top of the graph, there is the overhead in percent that the measurement tools introduce. All measurements were executed on Intel Broadwell CPUs and repeated 20 times. To avoid any differences between the nodes on miniHPC all experiments were conducted on node 8. The time measurement comes from the walltime measured during the execution of the application.

In figure 4.3, we see that in most cases the execution time of the application Mandelbrot with no measurements is the lowest. This is as expected because there is no additional work in these executions. The overhead that the tools introduce is between 0% and 5%. With expert chunks, there is less difference between the scheduling techniques. Especially the execution time with dynamic and RandomSel is similar to the other results. We see a very consistent overhead of the measurement tools. PAPI has an overhead between 0.2% and 1%, except for the executions with GAC where the overhead is 2.6%. The overhead of perf is around 0.5% and Likwid around 1.2%.

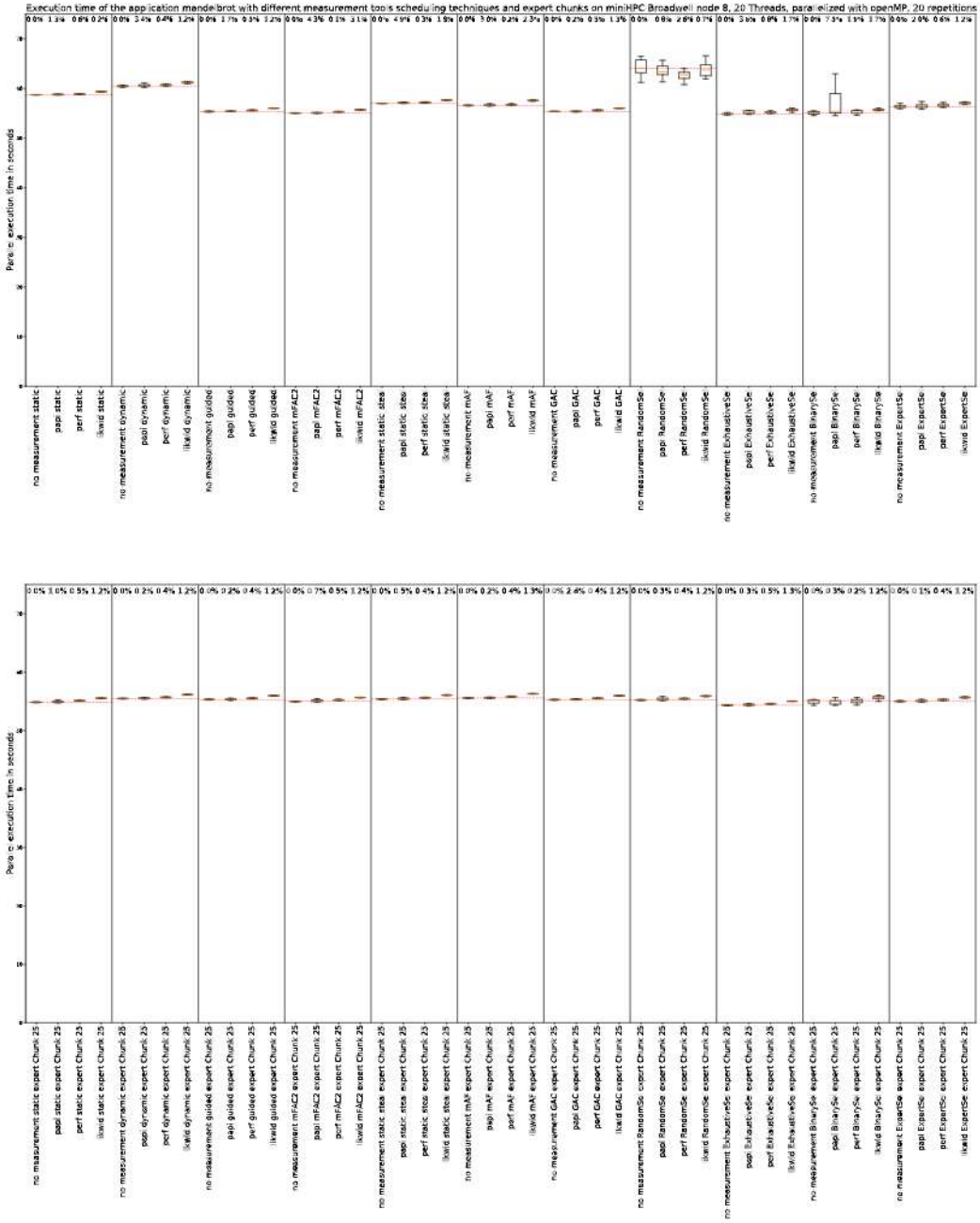


Figure 4.3: Overhead of the measurement tools PAPI, perf, and Likwid with the application Mandelbrot executed on Broadwell. On the top Graph with expert chunks disabled and on the bottom one with expert chunks enabled. The number on top of each graph is the average overhead introduced by the measurement compared to executions without measurements.

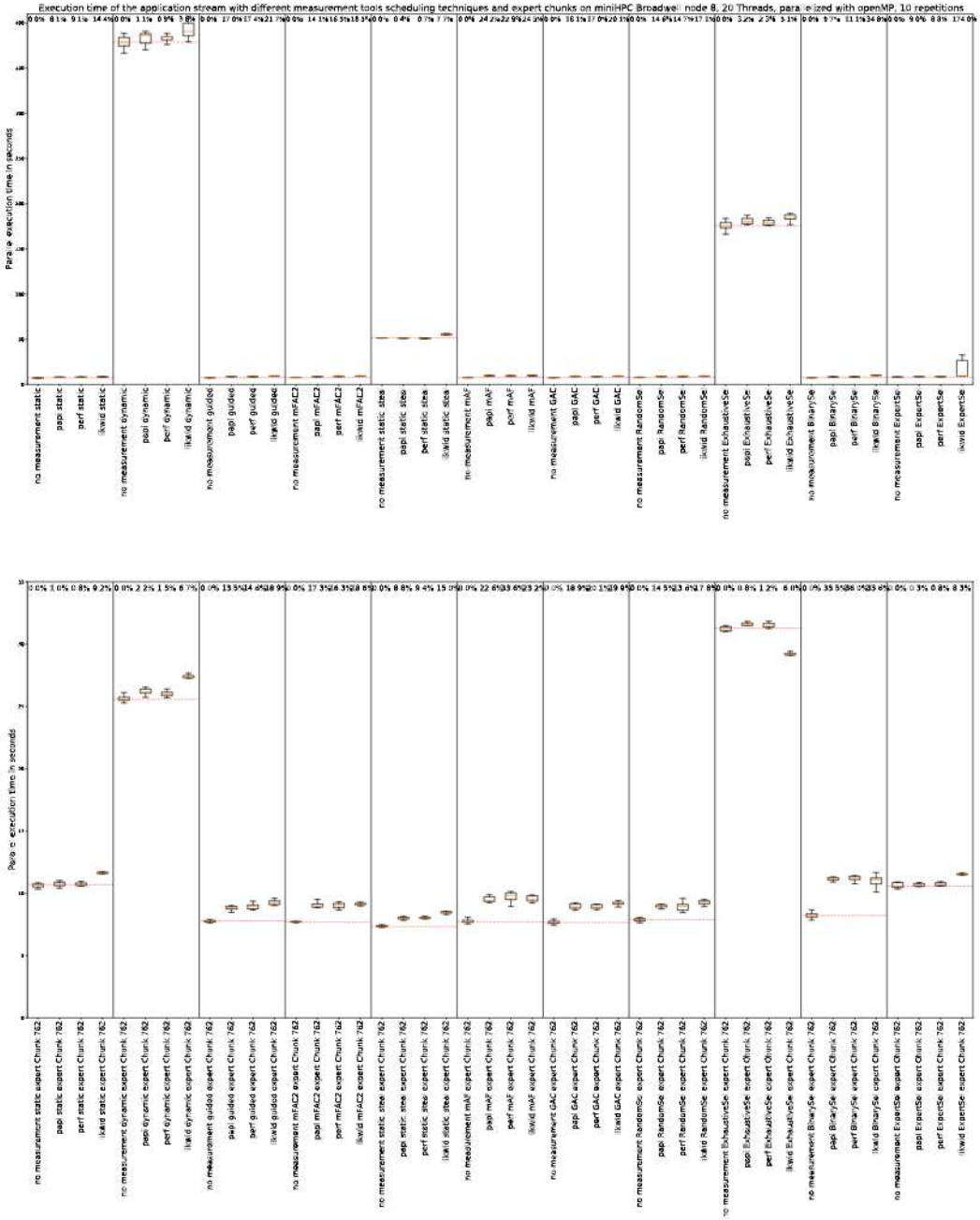


Figure 4.4: Overhead of the measurement tools PAPI, perf, and Likwid with the application Stream. On the top Graph with expert chunks disabled and on the bottom one with expert chunks enabled. The number on top of each graph is the average overhead introduced by the measurement compared to executions without measurements.

In contrast to Mandelbrot which is compute-bound, Stream is memory-bound. Memory-bound applications are more affected by the measurements, as we can see in figure 4.4. The measurement tools read the hardware counter and periodically store this data in the memory. This uses some of the available memory bandwidth. The application can not use the memory bandwidth fully. Therefore, the measurement tools introduce a much larger overhead compared to the executions with Mandelbrot. The scheduling techniques dynamic

and ExhaustiveSell are much worse than the other techniques. Expert chunks do not reduce the overhead as much as with the application Mandlebrot. The overhead depends more on the used scheduling technique. The lowest overhead is measured with the scheduling techniques static and ExpertSel.

The execution time with PAPI and perf measurements is also not much higher. In contrast Measurements with Likwid introduces an overhead of up to 20%. The reason for this difference is that the measurement with Likwid is more extensive than with perf or PAPI. We measured fewer counters with perf and PAPI than with Likwid. We tried to use similar measurements, but Likwid's counter groups contain more counters than those of PAPI. A fair comparison is not easy. Small measurements do not influence the performance much. In some cases, the execution time is even smaller with the measurements with PAPI. But this is probably caused by random deviation because there is only a very small difference between the execution time of the native execution and the execution with PAPI.

5

Evaluation of OS Scheduler Events

In this chapter, we present the results of the experiments we conducted on miniHPC. We measured the executions of the applications Mandelbrot, Stream, SPH_EXA, and Merge on Broadwell, Cascadelake, and KNL nodes. In the heat maps, we compare different thread scheduling techniques with and without expert chunks on the x-axis with applications, computing systems, thread configurations, and parallelization methods on the y-axis. We repeated the experiments on Broadwell and Cascadelake five times, on KNL only three times, because of time constraints. Table 5.1 summarizes the factors of our experiments. We explained these factors in chapter 4. We split up the data into two parts to make it a bit clearer. In section 5.1 we evaluate the results for different thread configurations and in section 5.2 are the result for the different parallelization methods. In both sections, we compare the different applications, computing systems, and thread level scheduling techniques. In section 5.3 we summarize and discuss the measurements.

5.1 Results for Thread Configurations

In this section we present the data for the different thread configurations, with the applications Mandelbrot, Stream, SPH_EXA, and Merge, executed on Broadwell, Cascadelake, and KNL. We also compare different thread-level scheduling techniques with and without expert chunks.

In this section, we show the result for the measurements with different thread configurations. The first configuration is not particularly labeled. It is a normal execution on all available cores on a given computing system. With one idle core, the application is executed with one thread less than with the thirst configuration. With two idle cores, the application is executed with two cores less. And with one socket idle, the application executes on only one socket. On KNL the configuration is a bit different because there is only one socket. We described this in section 4.3

First we show the results measured with perf for the thread migration events 5.1.1, then the context switch events 5.1.2 and the idle time events 5.1.3. After this, we present the data for the measurements with Likwid for the cache and memory performance in 5.1.4. We end this section with a summary of what we learned about the tread configurations 5.3.4

Table 5.1: Design of Factorial Experiments

Factors		Values	Properties
Applications	Mandelbrot	Broadwell	maxiter=1000; pixels=512; x0=0; y0=0; size=0.5
		Cascadelake	maxiter=1000; pixels=1000; x0=0; y0=0; size=0.5
		KNL	maxiter=100; pixels=512; x0=0; y0=0; size=0.5
	Stream	Broadwell	Array size=2,000,000,000
		Cascadelake	Array size=2,000,000,000
		KNL	Array size=500,000,000
	SPH_EXA	Broadwell	-n 100 -s 10 -w 100
		Cascadelake	-n 100 -s 10 -w 100
		KNL	-n 100 -s 1 -w 100
	Merge compute-b. & memory-b.	Broadwell	maxiter=1000; pixels=512; x0=0; y0=0; size=0.5; Array size=2,000,000,000
		Cascadelake	maxiter=1000; pixels=1000; x0=0; y0=0; size=0.5; Array size=2,000,000,000
Thread level scheduling techniques	OpenMP standard	static	Straightforward parallelization
		dynamic, guided	Dynamic and non-adaptive
	LLVM OpenMP RTL	static_steal	An extension of static
		GAC	A variant of guided self-scheduling
	LB4OMP	mFAC2	Dynamic and non-adaptive self-scheduling
		mAF	Dynamic and adaptive self-scheduling techniques
	Auto4OMP	RandomSel, ExhaustiveSel, BinarySel, ExpertSel	Automated DLS algorithm selection across application loops and time-steps application loops and time-step
Expert Chunk		Yes	Automatically calculated
		No	Naive size 1
Computing nodes		miniHPC-Broadwell	Intel Broadwell E5-2640 v4 (2 sockets, 10 cores per socket)
		miniHPC-GPU	Intel Cascadelake (2 sockets, 28 cores per socket)
		miniHPC-KNL	Intel Xeon Phi KNL 7210 (1 socket, 64 cores)
Thread Configuration	All threads pinned	All systems	OMPPROCBIND="close",OMP_PLACES="cores"
	One core idle	Broadwell	OMP_PLACES="{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18}"
		Cascadelake	OMP_PLACES="{0,...,55}"
		KNL	OMP_PLACES="{0,...,62}"
	Two cores idle	Broadwell	OMP_PLACES="{0,1,2,3,4,5,6,7,8, 10,11,12,13,14,15,16,17,18}"
		Cascadelake	OMP_PLACES="{0,...27.29,...,55}"
		KNL	OMP_PLACES="{0,...,31,33,...,62}"
	One socket idle	Broadwell	OMP_PLACES="{0,1,2,3,4,5,6,7,8,9}"
		Cascadelake	OMP_PLACES="{0,...,27}"
		KNL	OMP_PLACES="{0,...,32}"
Parallelization		OpenMP	OMPPROCBIND="close",OMP_PLACES="cores"
		MPI	1 Process per available Core
		Hybrid: OpenMP& MPI	2 MPI Ranks, with halve of the available cores each

Metrics	Memory and cache performance	L1 request rate L1 miss rate L1 miss ratio L2 bandwidth [MBytes/s] L2 data volume [GBytes] L2 request rate L2 miss rate L2 miss ratio L3 bandwidth [MBytes/s] L3 data volume [GBytes] L3 evict bandwidth [MBytes/s] L3 evict data volume [GBytes] L3 load bandwidth [MBytes/s] L3 load data volume [GBytes] L3 request rate L3 miss rate L3 miss ratio Memory bandwidth [MBytes/s] Memory data volume [GBytes] Memory read bandwidth [MBytes/s] Memory read data volume [GBytes] Memory write bandwidth [MBytes/s] Memory write data volume [GBytes] L1 DTLB load misses L1 DTLB load miss rate L1 DTLB load miss duration L1 DTLB store misses L1 DTLB store miss rate L1 DTLB store miss duration L1 ITLB misses L1 ITLB miss rate L1 ITLB miss duration L1 ITLB miss duration
	Thread migration	wait time [ms] scheduler delay [ms] run time [ms] Number of events
	Context switches	wait time [ms] scheduler delay [ms] run time [ms] Number of events
	Idle time	wait time [ms] run time [ms] Number of events

5.1.1 Thread Migration

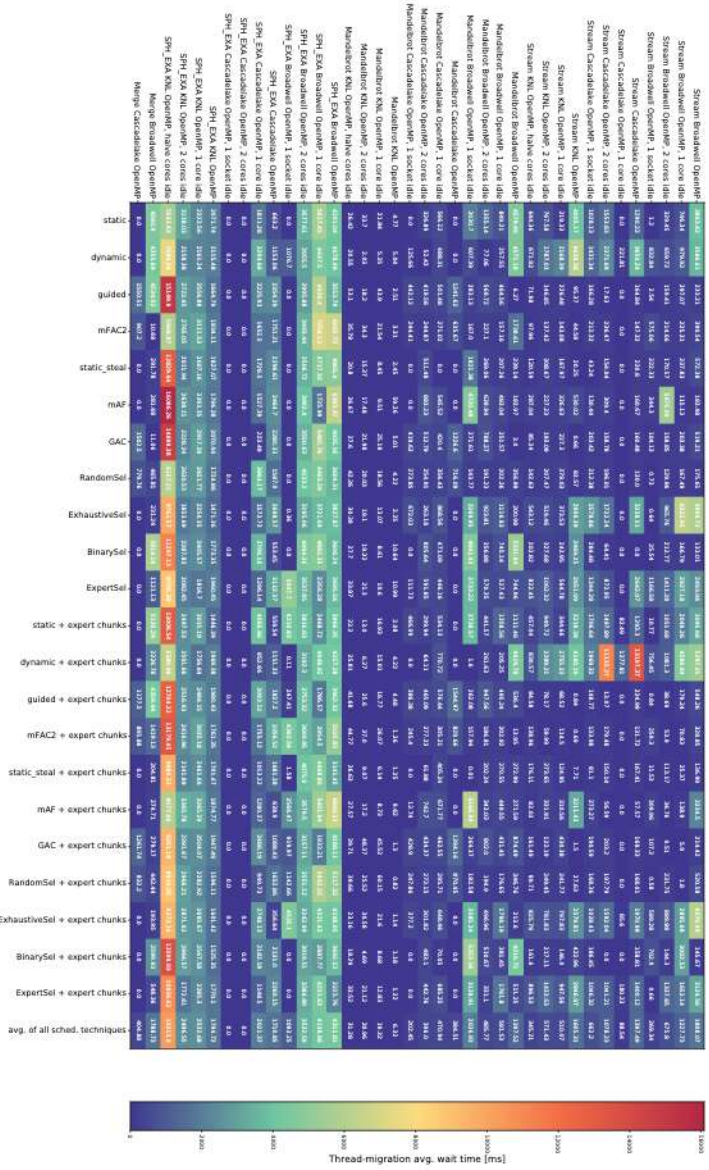


Figure 5.1: Average wait time for thread migration events for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

In figure 5.1, we have the average wait time for thread migration events. This is the time spent waiting until the scheduler lets the application execute. We observe that SPH_EXA has the highest wait times. These high numbers are caused by some outliers, which have a reported wait time of several seconds. For the application Stream, some scheduling techniques lead to higher wait times. For static and dynamic there is a higher wait time with expert chunks than without. On KNL idle cores lead to higher wait times, this is opposite to

Broadwell. On Broadwell, the wait time is lower when one or two cores are idle, compared to executions with all cores. On Cascadelake a loot of zero values is reported.

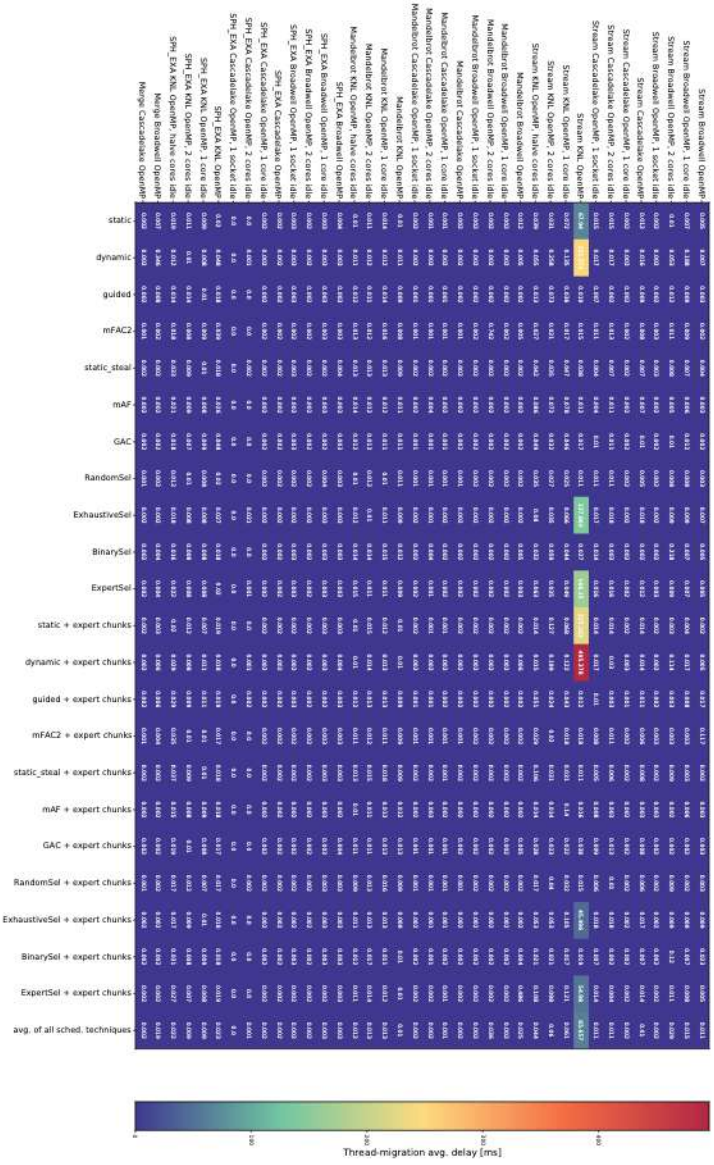


Figure 5.2: Average scheduler delay for thread migration events for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

In figure 5.2 we see the average scheduler latency. The thread migration delay is mostly near zero, except for Stream on KNL with all cores. There are some outliers for the scheduling techniques static, dynamic, ExhaustiveSel, and ExpertSel with and without expert chunks. Except for these outliers, there are no values above 1 ms.

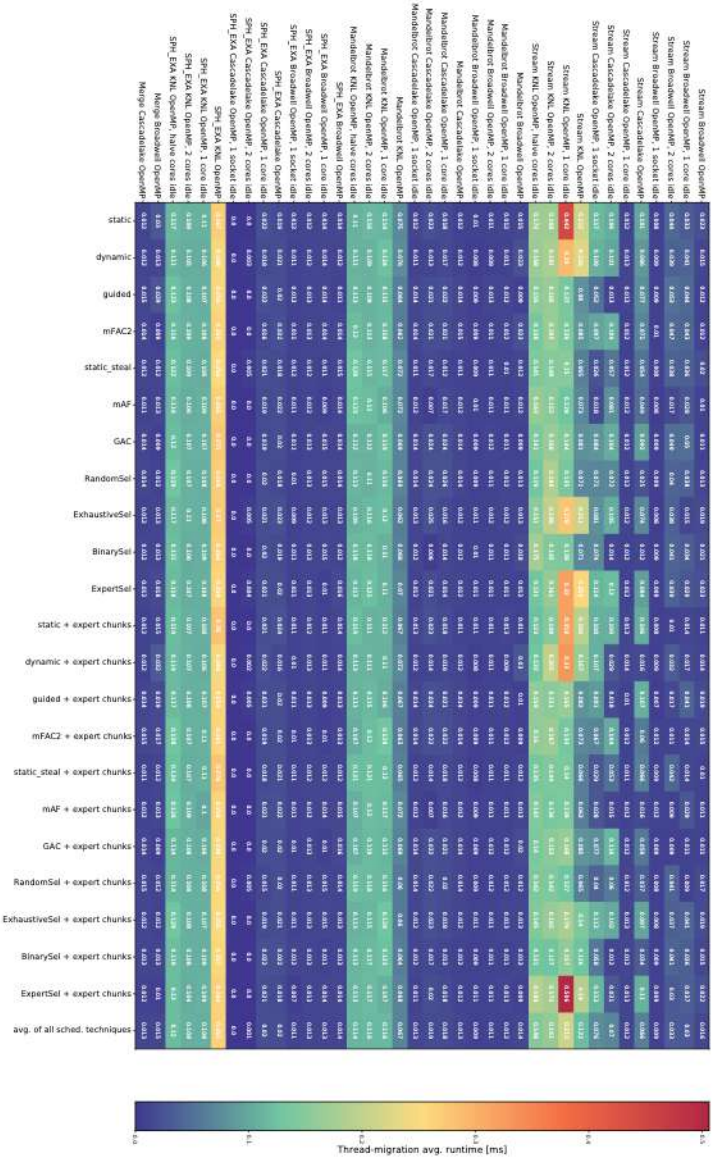


Figure 5.3: Average runtime for thread migration events for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

In figure 5.3 we see the average duration of a migration event. Between the different applications, thread configurations and scheduling techniques is no big difference. Migrations on KNL take longer than on the other systems. So the time for thread migration depends mostly on the computing system. Some measurements on Cascadelake were reported as 0.000 ms. We do not think that these are the correct times. Why perf reports zero values for some thread configurations and not for others, needs to be explored in future work.

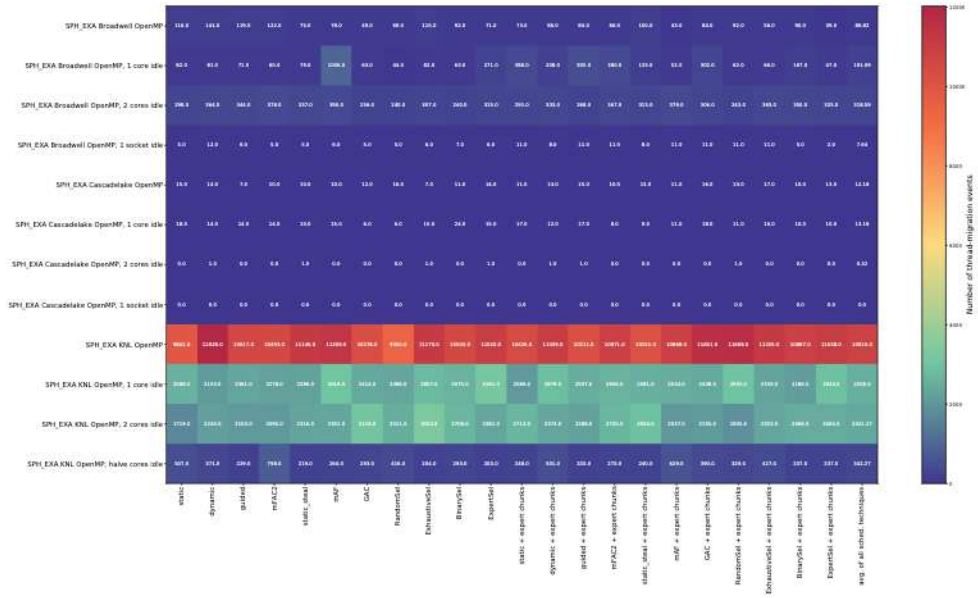


Figure 5.4: The Number of thread migration events for the application SPH_EXA on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

Figure 5.4 shows the number of thread migration events for SPH_EXA. The results for the other applications are in the next plot (figure 5.5). We divided this plot because the data for SPH_EXA on KNL dominates the heatmap.

We see that the highest number of thread migration events occur for SPH_EXA on KNL (figure 5.4). SPH_EXA has the longest execution time of all applications and KNL is the slowest system in our set. Therefore, SPH_EXA on KNL had the longest execution time. We expected that there are more migrations when the application executes longer because there are periodical interrupts by the OS. The longer an application executes the more interrupts occur.

On KNL for all applications, with more idle cores fewer thread migrations take place. We think that this indicates, that the OS uses these idle cores. Since the threads of the applications are pinned most of the migrations should be for other processes. The processes of the OS execute on the idle cores and are not migrated as much. The data for Broadwell does not support this hypothesis. On Broadwell, more thread migrations are reported with one or two idle cores than with all cores utilized. But with one idle socket, there are only very few migrations for Stream and SPH_EXA.

For SPH_EXA on Cascadelake with thread configurations, two idle threads, and an idle socket, there are many results with zero migrations. This is implausible. On Cascadelake Mandelbrot has much more thread migrations with idle cores than when using all cores. Stream has the fewest thread migration on Cascadelake with one idle core. So it might be highly dependent on the computing system which thread configuration leads to fewer thread migrations.

For Mandelbrot and Stream different scheduling techniques have more thread migrations. For Mandelbrot it is guided, mFAC2,GAC, and RandomSel. For Stream, it is static, dynamic, and ExhaustiveSel. The expert chunk does not make a difference.

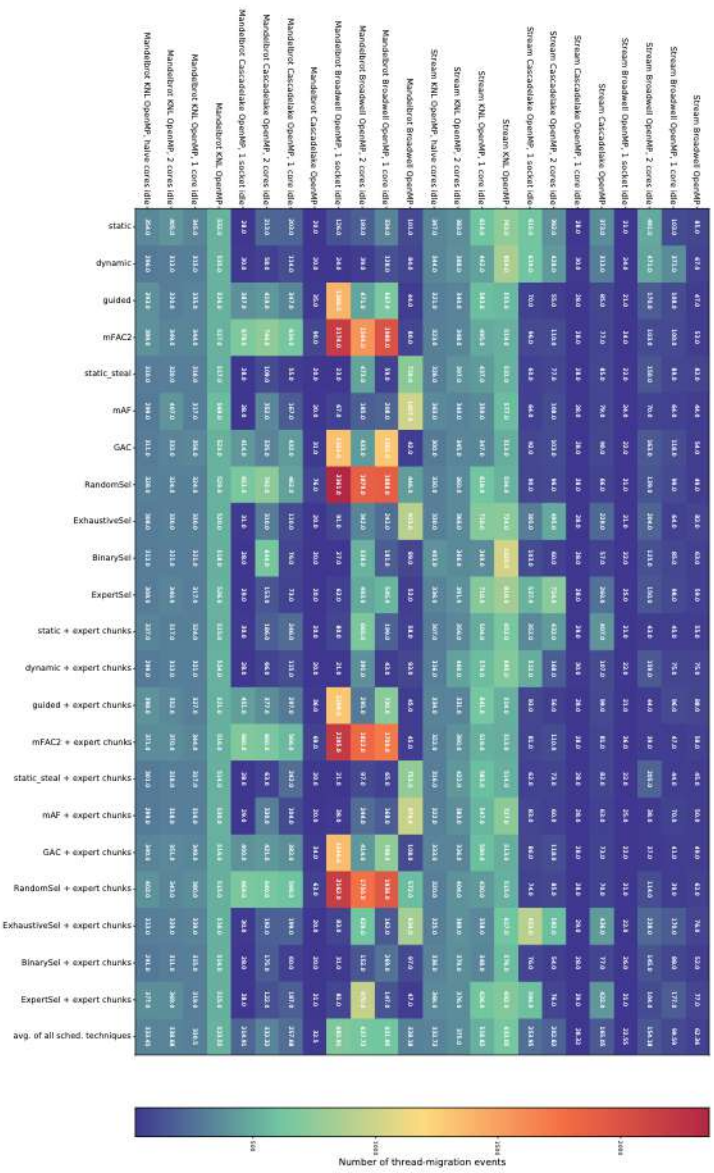


Figure 5.5: The Number of thread migration events for the applications Mandelbrot and Stream on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

5.1.2 Context Switches

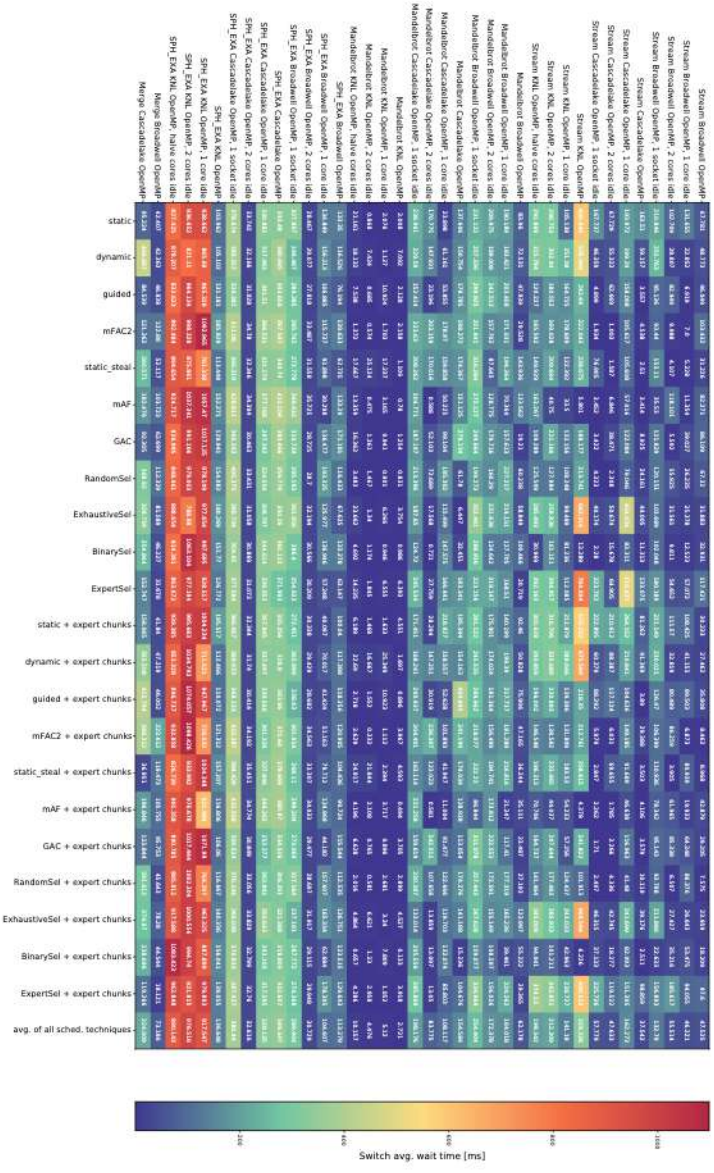


Figure 5.6: Average wait time for context switches events for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

In figure 5.6 we see the average time between context switches. The average wait time for context switches for the applications Stream and SPH_EXA is higher on KNL than on the other nodes. For Mandelbrot, this is the other way round. This is similar to the switch runtime, see figure 5.8.

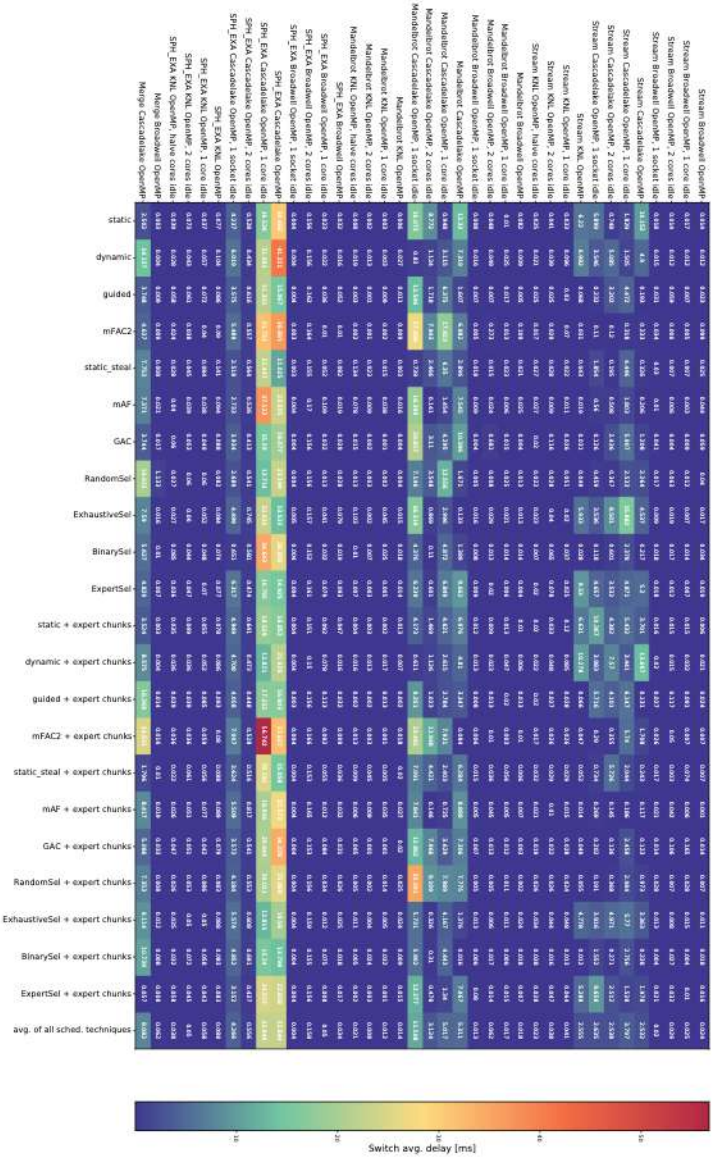


Figure 5.7: The average delay for context switches events for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

The average switch delay (figure 5.7) is the highest on Cascadelake compared to the other systems. The scheduling technique mFAC2 with and without expert chunks has the highest OS scheduler latency. It seems that for different applications different scheduling techniques perform well.

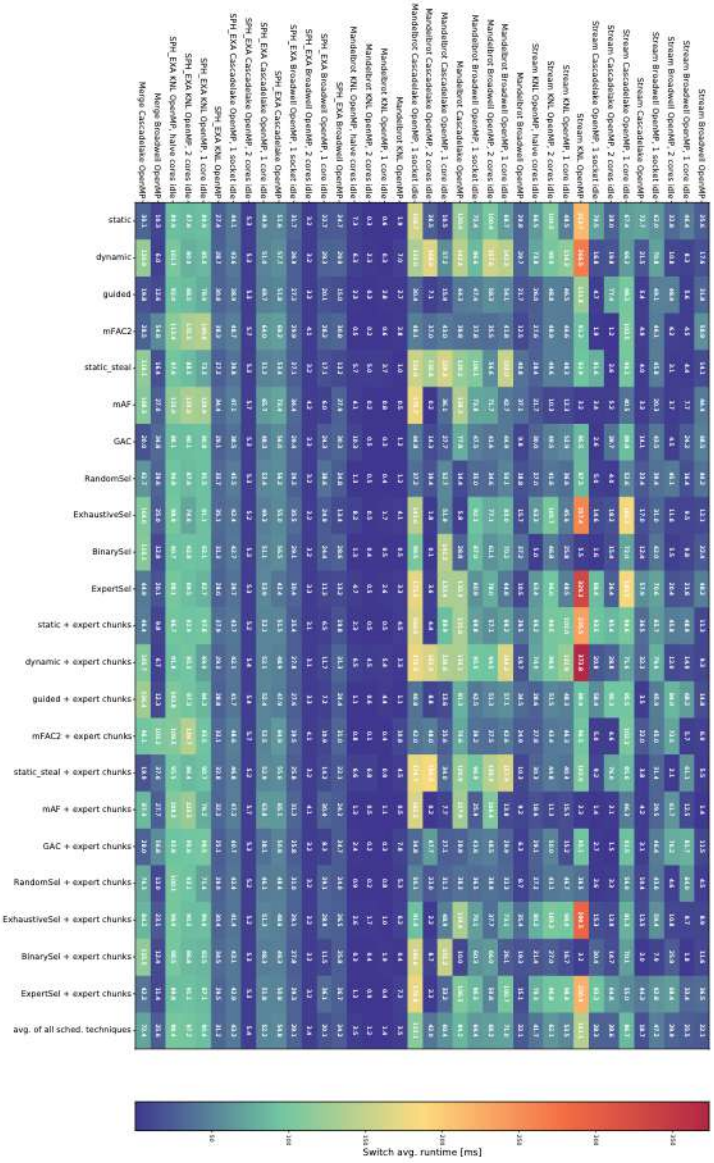


Figure 5.8: Average runtime for context switches events for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

The runtime of context switches (figure 5.8) is, as the wait time (figure 5.6) for the applications Stream and SPH_EXA higher on KNL than on the other systems. Opposite to Mandelbrot, which has the lowest switch runtimes on KNL We are still investigating the reason why the context switches on KNL perform so differently for different applications.

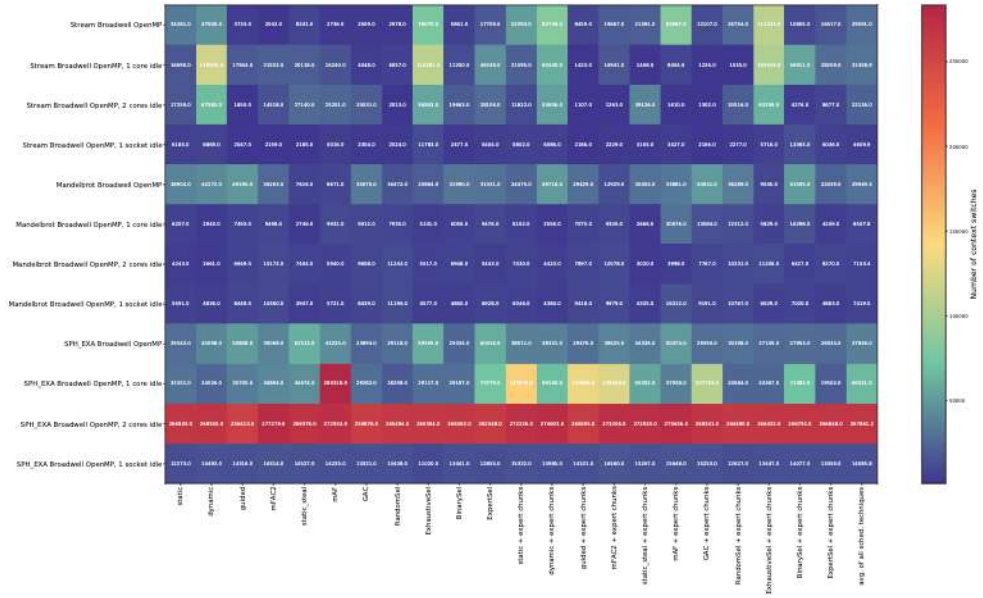


Figure 5.9: The number of context switch events for the different applications on Broadwell. With different thread configurations and scheduling techniques.

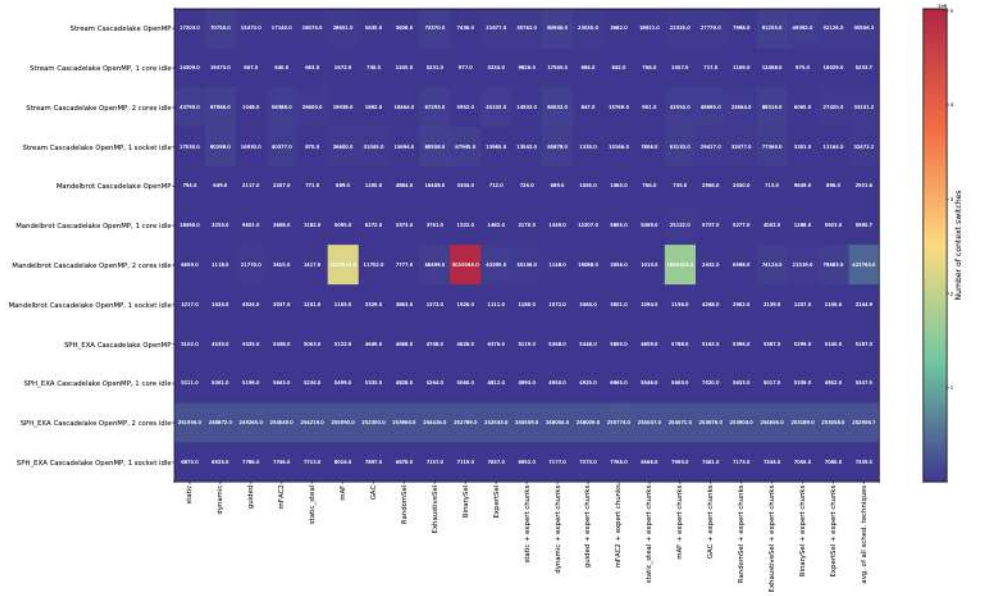


Figure 5.10: The number of context switch events for the different applications on Cascadelake. With different thread configurations and scheduling techniques.

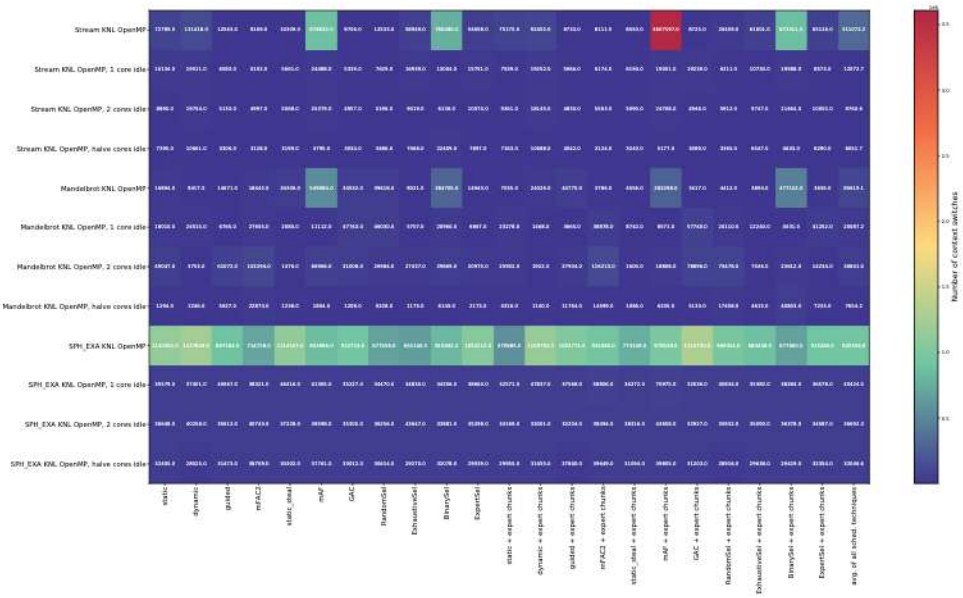


Figure 5.11: The number of context switch events for the different applications on KNL. With different thread configurations and scheduling techniques. Here we removed the outliers that dominate the plot above.

In figures 5.9 - 5.11 we have the number of context switches. There are two outliers, one with Mandelbrot on Cascadelake and the other with Stream on KNL. These two outliers have much more switches than the other measurements. On KNL there are fewer context switches when there are some idle cores. This is similar to the number of thread migrations on KNL (figure 5.5). The other systems behave differently. On Broadwell, the fewest context switches are reported for executions with one idle socket. On Cascadelake there are much more Switches with two idle cores than with other thread configurations. So the different systems behave differently for different thread configurations regarding the number of switches.

There is some difference between the applications. SPH_EXA executed for a longer time than the other application. Context switches, similar to thread migrations, happen over time because the OS is executing some processes. We expect that applications with longer execution times have more context switches. But Stream has more context switches than Mandelbrot. Although the execution time of Stream is shorter. A possible explanation for this behavior is that Stream waits for data and at this time another process is scheduled to execute. There is no big difference between the scheduling techniques.

5.1.3 Idle Time

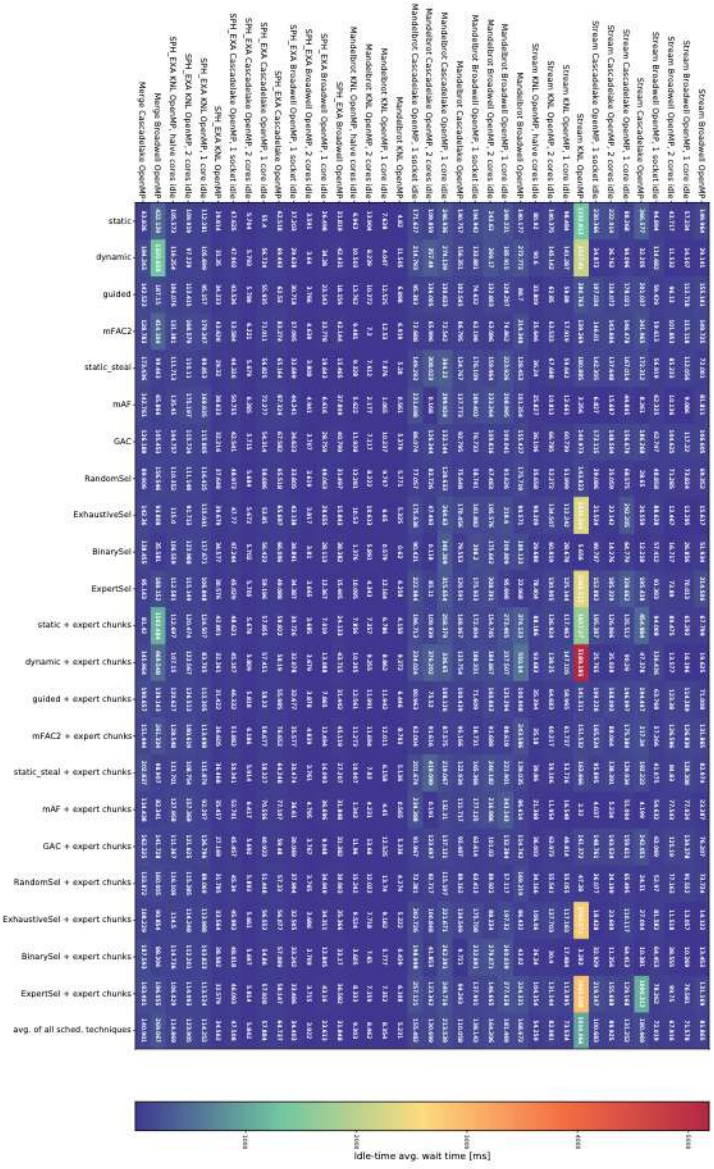


Figure 5.12: Average wait time for idle time events for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

The average wait time (figure 5.12) for idle events is the time between two idle events. This should be high for good performance. It is good when fewer idle events occur. There are some outliers for Stream on KNL and one on Cascadelake. This is surprising, we expected that Stream had relatively frequent idle time, to wait for data. For a better overview of the other data, we have a separate plot without the outliers (figure 5.13). There we observe that Mandelbrot has a very low wait time between idle events on KNL, while SPH_EXA has a high wait time on KNL. On Broadwell and Cascadelake this is the other way round.

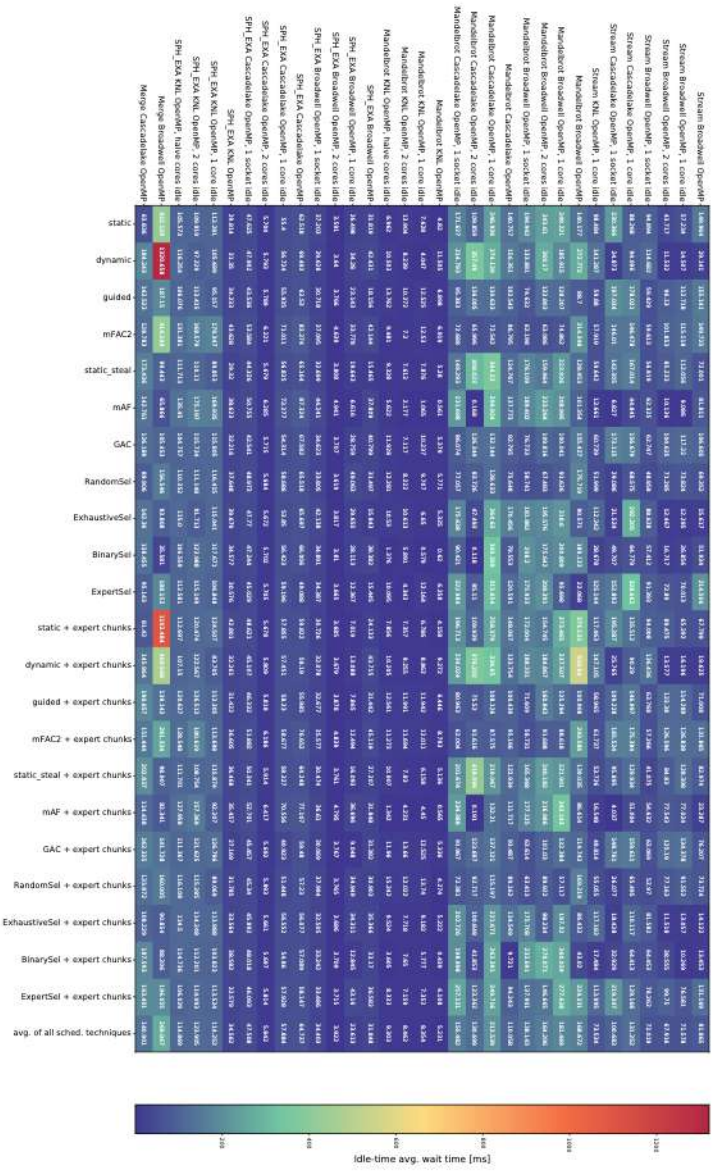


Figure 5.13: Average wait time for idle time events. Some of the outliers are not shown, to have a better overview of the rest of the data.

It is not necessary to show the plot for the average delay for idle time events. It is always zero. Idle time is not a process that requests time on a CPU. Therefore it does not wait until the scheduler allocates computing time to it.

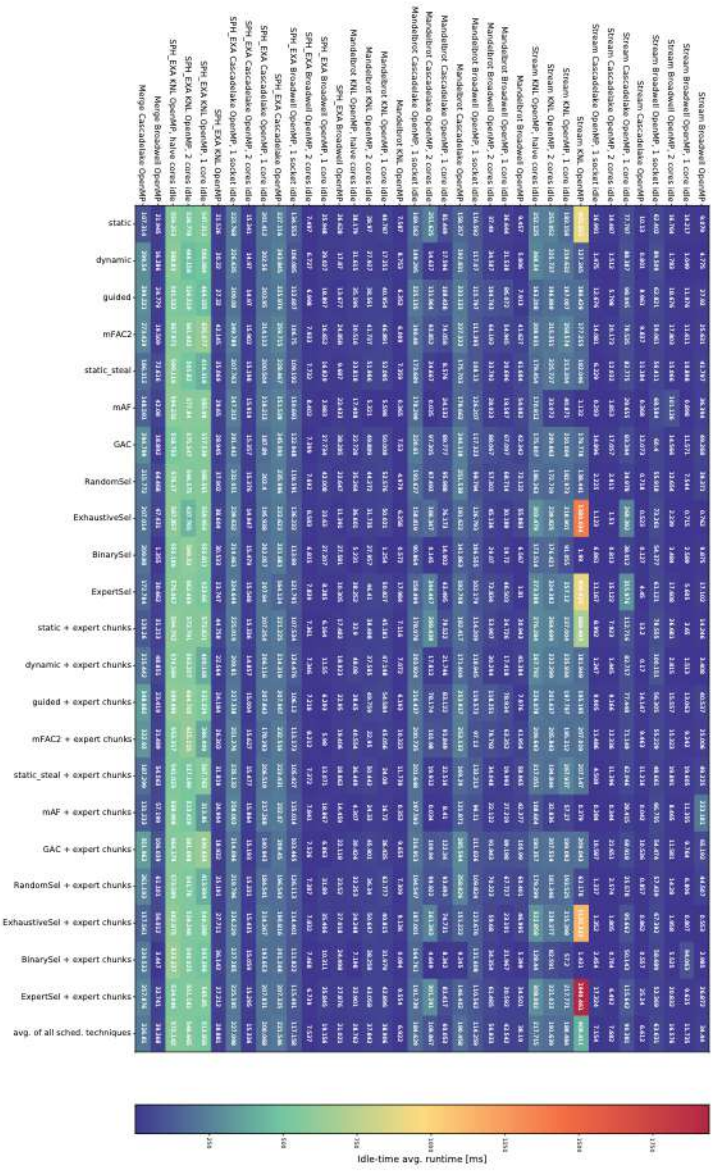


Figure 5.14: Average runtime for idle time events for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

The average runtime for idle events (figure 5.14) is the average time a CPU was idle for a recorded idle event. It shows outliers for Stream on KNL, similar to the average wait time (figure 5.12). This indicates that Stream on Cascadelake has fewer but longer idle events. Compared to these outliers, the other executions have more frequent, but shorter idle events.

On Broadwell, the average runtime of an idle event is longer when one socket is idle. But this is not the case on other systems.

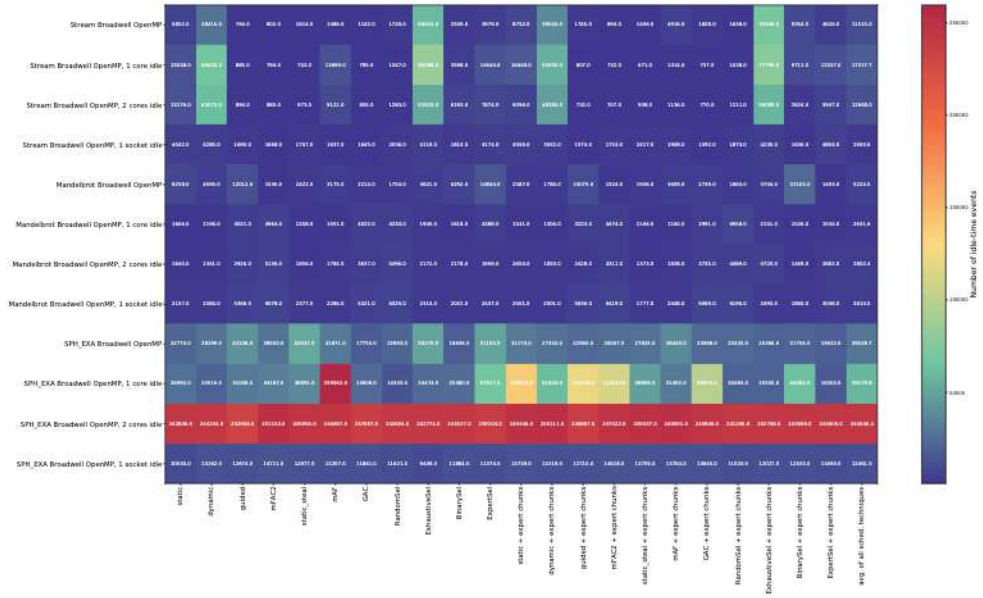


Figure 5.15: The number of idle time events for the different applications on Broadwell. With different thread configurations and scheduling techniques.

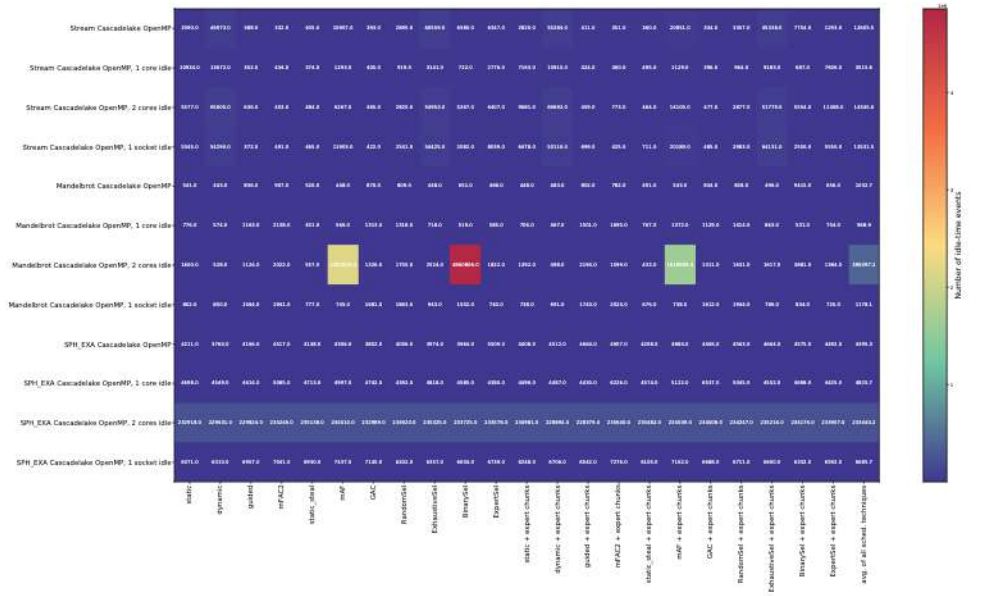


Figure 5.16: The number of idle time events for the different applications on Cascadelake. With different thread configurations and scheduling techniques.

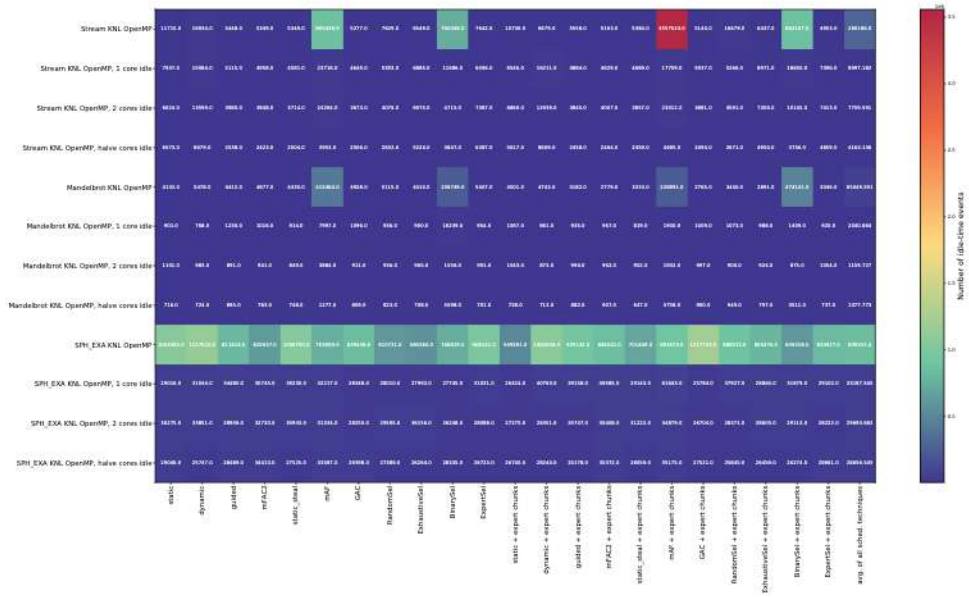


Figure 5.17: The number of idle time events for the different applications on KNL. With different thread configurations and scheduling techniques.

The number of idle time events (figures 5.15 - 5.17) has the same pattern as the number of context switches (figure 5.9). We see outliers with the same configurations.

On Broadwell SPH_EXA with one or two idle cores has more idle times. We expected to see this. The OS should not fully use the idle cores, but only from time to time. The idle cores have a lot of idle time events. But with one idle socket, there are fewer idle events than when all cores are used. With the average idle runtime, see figure 5.14, for SPH_EXA on Broadwell, we see that these idle events took much longer. So with one idle socket, there are fewer idle events that took on average much longer, compared to more utilized cores. This shows that the unused cores on one socket were indeed idle for most of the time. The average runtime of idle events for the other applications Mandelbrot and Stream on Broadwell are also longer when one socket is idle. But for the other systems, this is not the case. On Cascadelake the thread configuration with the shortest idle time events differ between the applications. Also, the difference between the number of idle events is not that big compared to other systems. On KNL all applications have more idle events when all cores are utilized. The average runtime of these idle events on KNL for the applications Mandelbrot and SPH_EXA are longer when all cores are used, compared to the measurements with some idle cores. But Stream on KNL has the longest and the idlest events when all cores are used.

5.1.4 Memory and Cache Performance

With Likwid we measured the performance of memory and cache. Not all counter groups of Likwid are available on every system. Therefore, not all comparisons are possible.

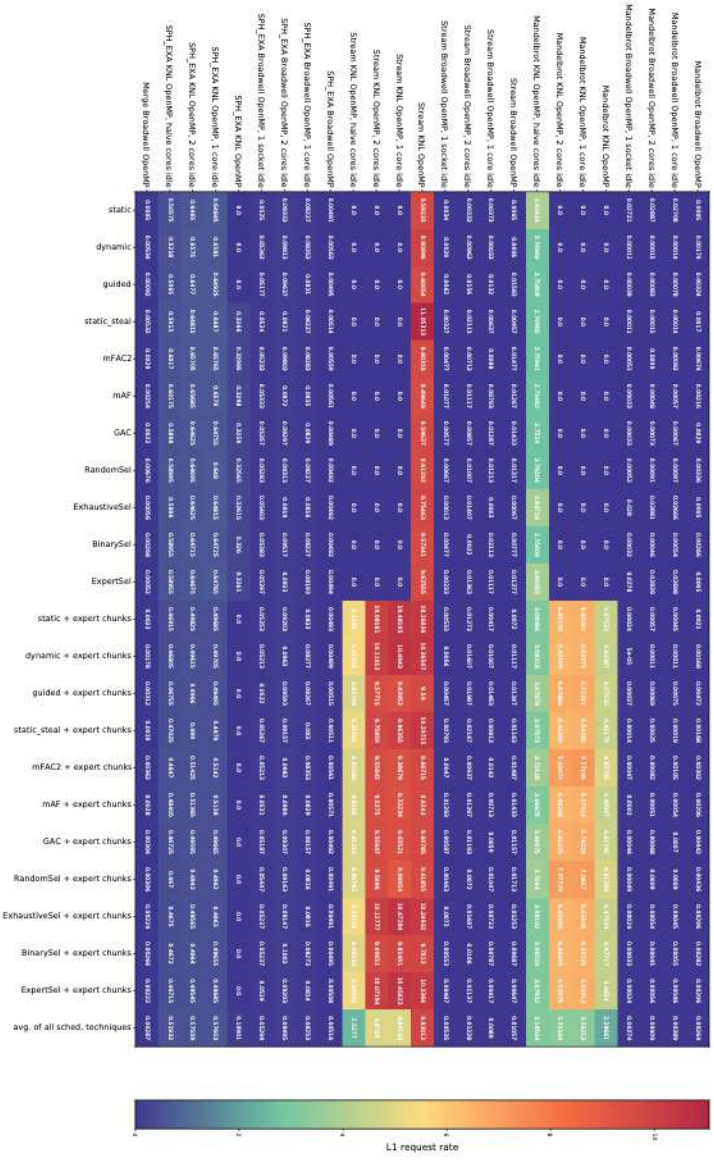


Figure 5.18: L1 request rate for the different applications on Broadwell and KNL. With different thread configurations and scheduling techniques.

In figure 5.18 we see the L1 request rate. L1 request rate is the rate of instructions that access the L1 cache. This rate should be higher for applications that read more memory. As we can see for the same configurations, the L1 request rate is lower for Mandelbrot than for Stream. These two applications have a much higher L1 request rate on Broadwell than on KNL. This is not the case for SPH_EXA. For SPH_EXA there is not a big difference between the two systems. The results from KNL seem strange because for executions without expert chunk parameters the measured L1 request rate is often 0. It is much higher with expert chunks. We do not have an explanation for this.

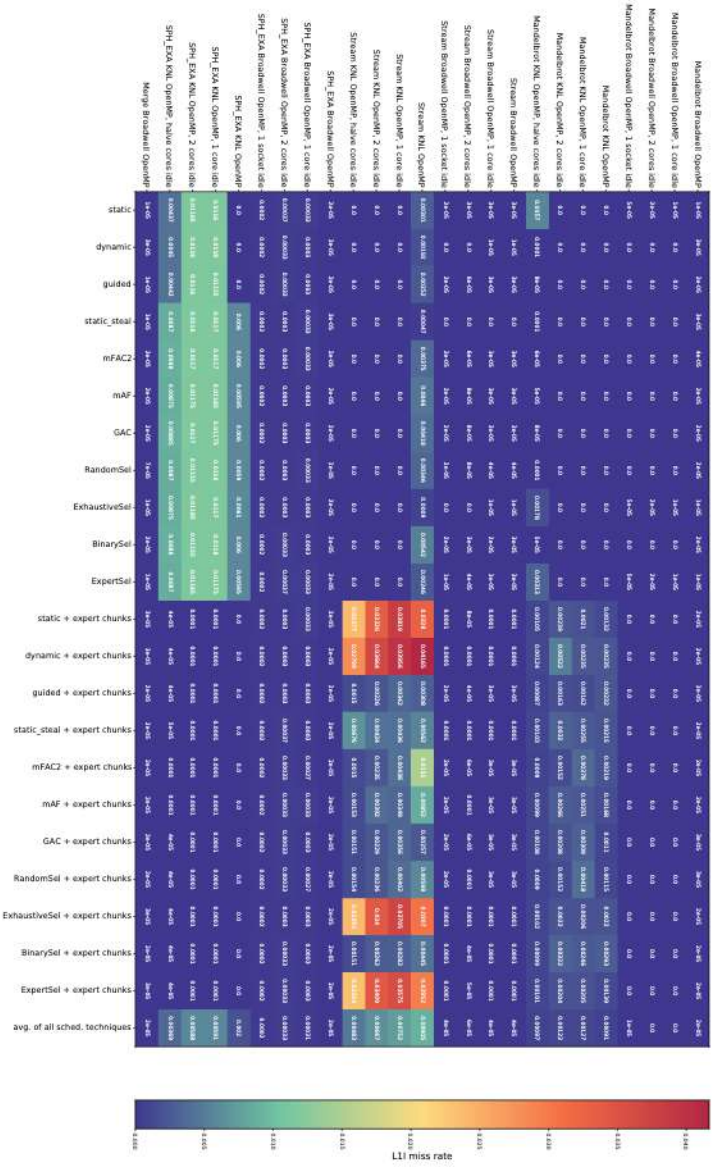


Figure 5.19: L1 miss rate for the different applications on Broadwell and KNL. With different thread configurations and scheduling techniques.

L1 miss rate (figure 5.19) is the fraction of L1 cache misses by the number of instructions. A low miss rate is good. But for a memory-bound application, we expected some cache misses. When comparing Mandelbrot and Stream this is the case. Mandelbrot has very low values, often zero on Broadwell. Stream has higher values, especially on KNL. As in previous plots, the results for Mandelbrot and Stream on KNL are zero without expert chunks. This is not the case for SPH_EXA which has higher values for the executions without expert chunks. This shows that expert chunks lead to better scheduling where fewer cache misses occur. We can also observe that for Stream on KNL, the scheduling techniques static,

dynamic, ExhaustiveSel, and ExpertSel with expert chunks have higher miss rates than other scheduling techniques.

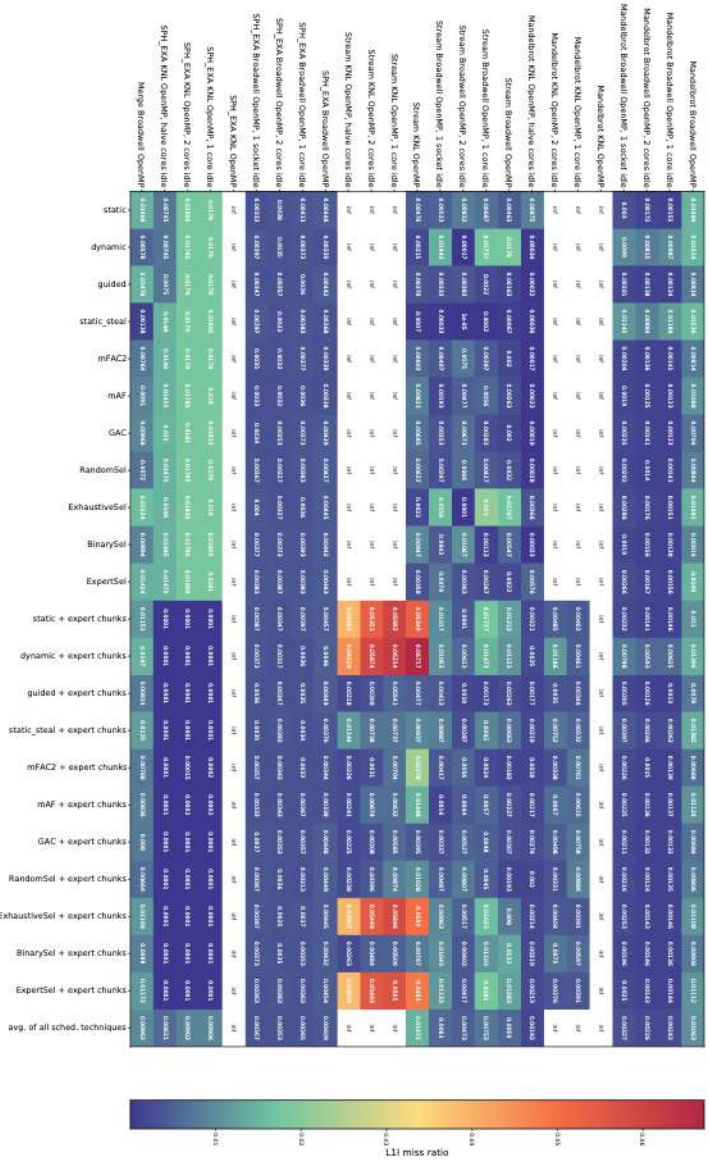


Figure 5.20: L1 miss ratio for the different applications on Broadwell and KNL. With different thread configurations and scheduling techniques.

The L1 miss ratio (figure 5.20) is the fraction of L1 misses to the number of L1 accesses. The result looks very similar to the L1 miss rate. Mandelbrot has a lower miss ratio than Stream. SPH_EXA performs better with expert chunks, and for Stream, on KNL the same scheduling techniques show the worst results. In contrast to the miss rate, here are a loot of *inf* values. This is probably because the number of counted L1 accesses is zero. When calculating the miss ratio, Likwid divides by zero. We assume that the reason for the strange

values for the L1 miss rate and ratio on KNL is, that some counters are zero.

For the L1 cache misses we do not see much difference between the thread configurations. So an idle core does not lead to fewer L1 cache misses. On Broadwell and KNL each core has its own L1 cache. Likwid does not measure the core where no application is executing, because of the pinning configuration.

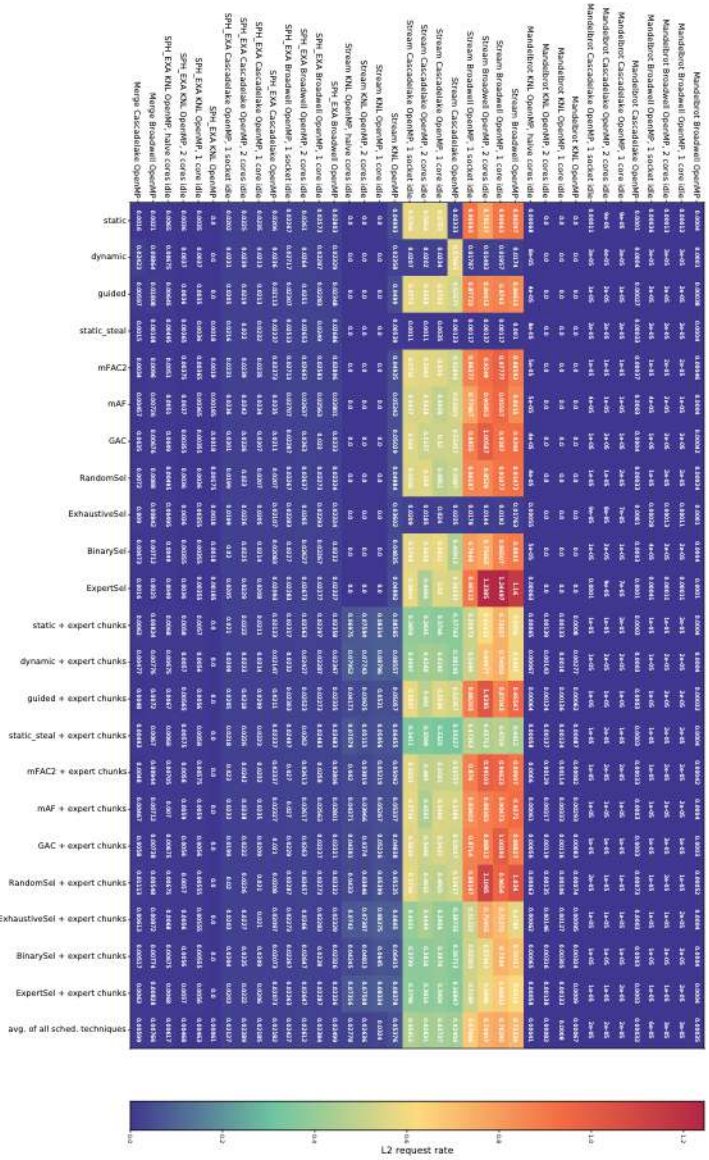


Figure 5.21: L2 request rate for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

The L2 request rate (figure 5.21) looks similar to the L1 request rate (figure 5.18). Applications with higher memory needs have a higher request rate. Stream requests more than SPH_EXA, which requests more than Mandelbrot. And as for the L1 measurements, the

data for Stream and Mandelbrot on KNL does not look correct. All executions without expert chunks have a reported L2 request rate of zero.

Remarkable is the difference between the different scheduling techniques for Stream. Dynamic, Static_Steal, and ExpertSel without expert chunks have much lower L2 request rates. With expert chunks, the difference to other scheduling techniques is lower. The scheduling technique with the highest request rate is ExpertSel without expert chunks. The thread configuration for Mandelbrot on Broadwell and Cascadelake is interesting. With all cores used the L2 request rate is higher than when some cores are idle.

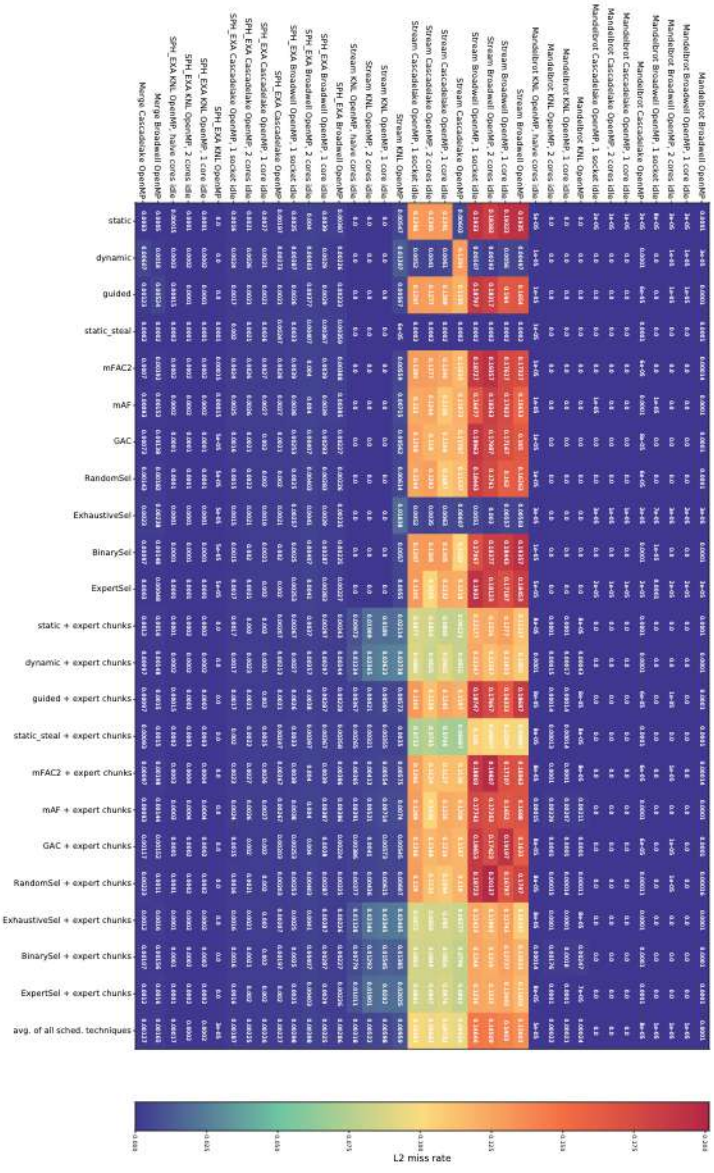


Figure 5.22: L2 miss rate for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

The L2 miss rate (figure 5.22) is similar to the L2 request rate (figure 5.21). For Stream with the scheduling techniques dynamic, static_steal, and ExhaustveSel the recorded cache request and miss rate on L2 is much lower than with other scheduling techniques. This is caused by the longer execution time of the application with these scheduling techniques. The data arrives in time on the cache when the application executes slower.

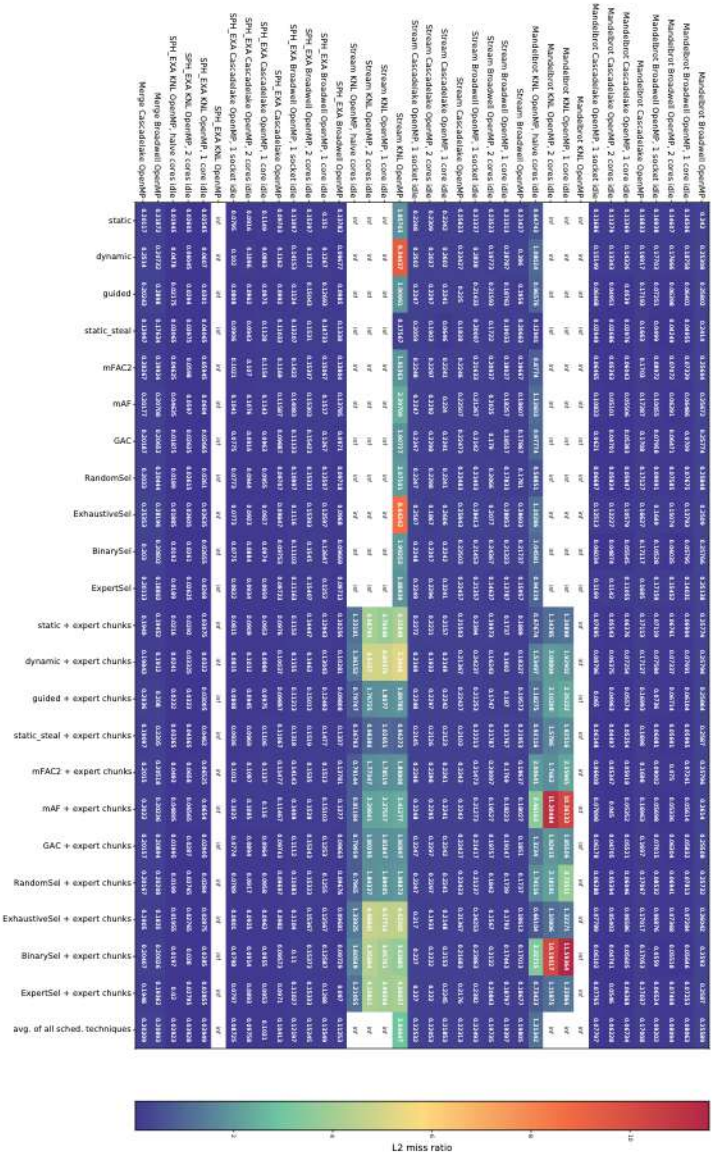


Figure 5.23: L2 miss ratio for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

For the L2 miss rate (figure 5.22) and request ratio (figure 5.23) the data from KNL is often zero for the miss rate and inf for the miss ratio. The miss rate is higher for applications with high memory usage. The high miss ratio on KNL, where there are no inf values, is

probably caused by small denominators. Because this plot is dominated by the outliers on KNL and the miss ratio seems similar for all applications, we have a separate plot without the results from KNL (figure 5.24). There we can see the L2 request ratio on Broadwell and Cascadelake. We can see that Mandelbrot has a much lower miss ratio when some cores are idle. This is not the case for SPH_EXA on Broadwell. SPH_EXA with two idle cores has a higher request ratio than with all threads used. But this is not the case on Cascadlake.

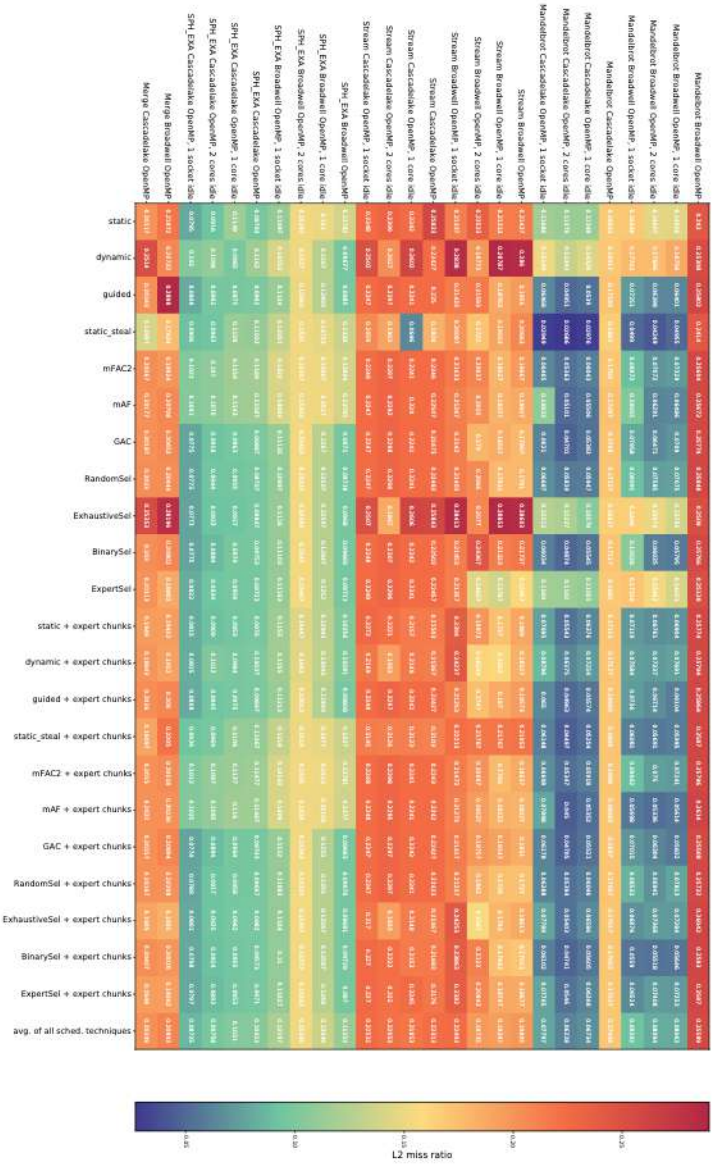


Figure 5.24: L2 request ratio only on Broadwell, Cascadelake for the different applications. With different thread configurations and scheduling techniques.

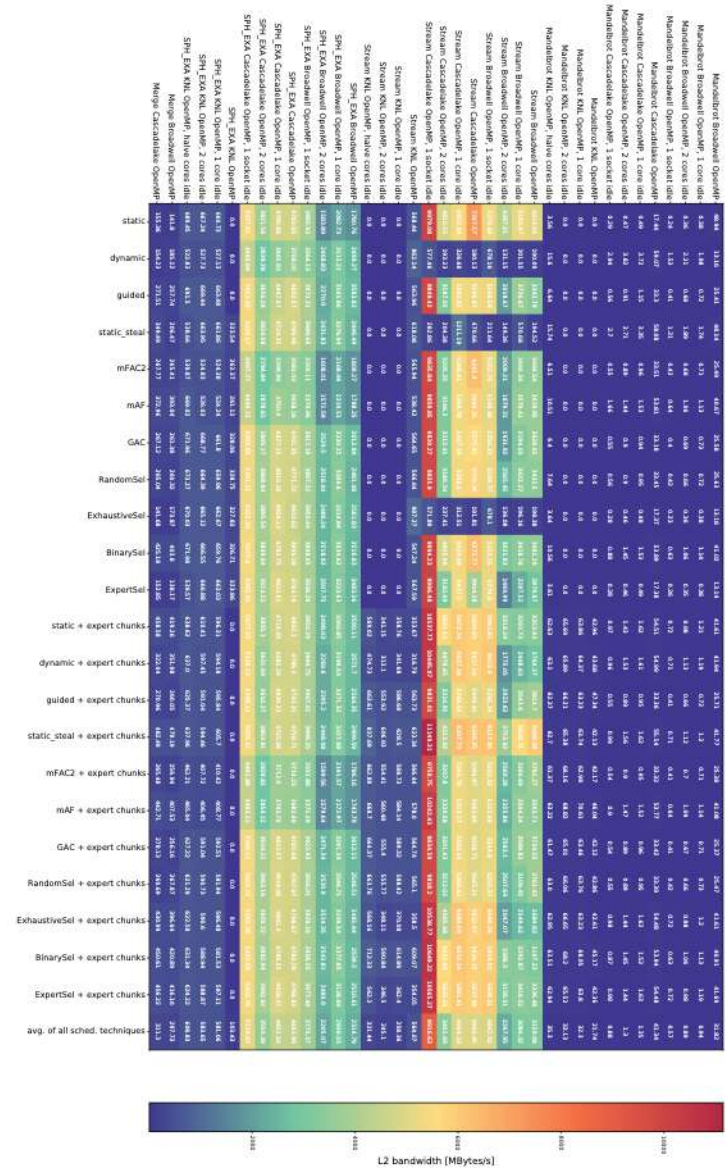


Figure 5.25: L2 bandwidth for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

The measured L2 bandwidth (figure 5.25) for Mandelbrot and Stream is often zero without expert chunks. The bandwidth between L2 and L1 cache is more used by Stream and SPH_EXA than Mandelbrot. But there is an interesting difference between those applications. Mandelbrot has a higher bandwidth when all cores are used and a lower bandwidth when some cores are idle. This is the opposite of the results for Stream and SPH_EXA. These applications have the highest bandwidth when one socket is idle. Also, both applications have higher bandwidth on Cascadelake than on Broadwell. For Stream, the scheduling techniques dynamic, static_steal, and ExhaustiveSel have the lowest bandwidth.

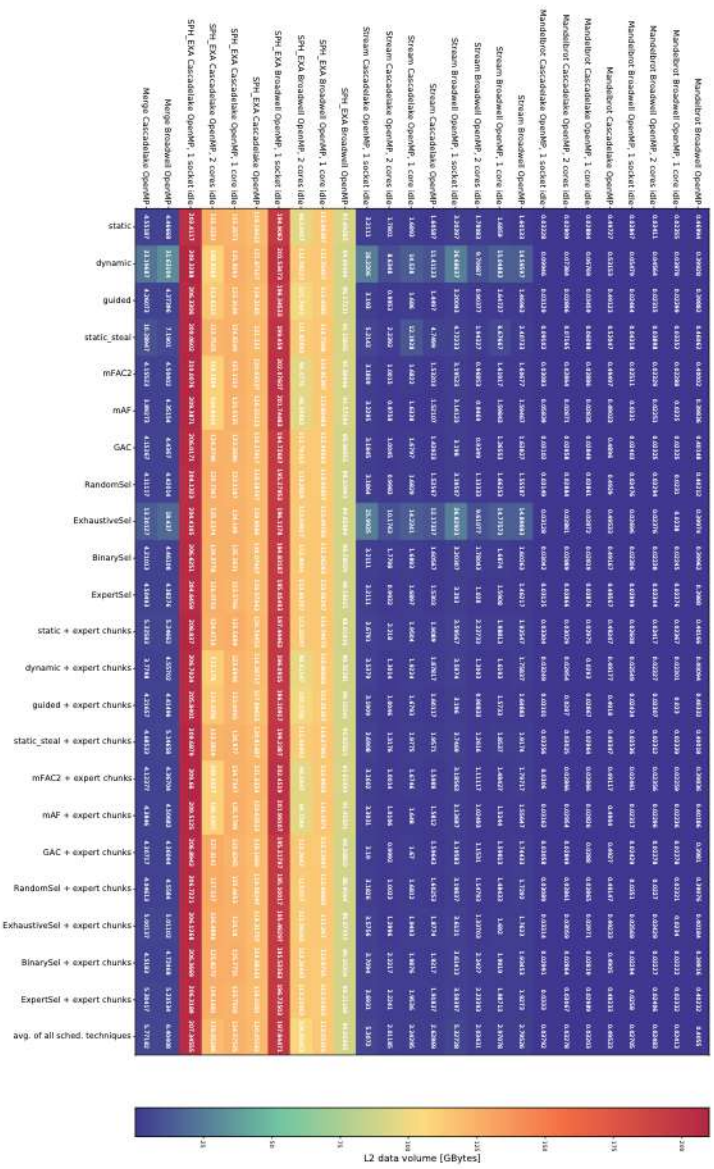


Figure 5.26: L2 data volume for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

For the L2 data volume (figure 5.26), we excluded the data from KNL, because it does not look correct. On KNL the reported datavolume was too high. The data volume for the different applications is quite different. Mandelbrot uses very little data volume between L1 and L2, Stream a bit more. The highest data volume is recorded for SPH_EXA. There we can also observe that with one idle socket the needed data does not fit in the L1 cache as well as with much more available cores. So a lot more data is loaded from L2. Stream uses the same data only once. Perhaps this is the reason for the difference between Stream and SPH_EXA. The scheduling techniques dynamic and ExhaustiveSel without expert chunks

have a higher bandwidth than the other techniques for Stream.

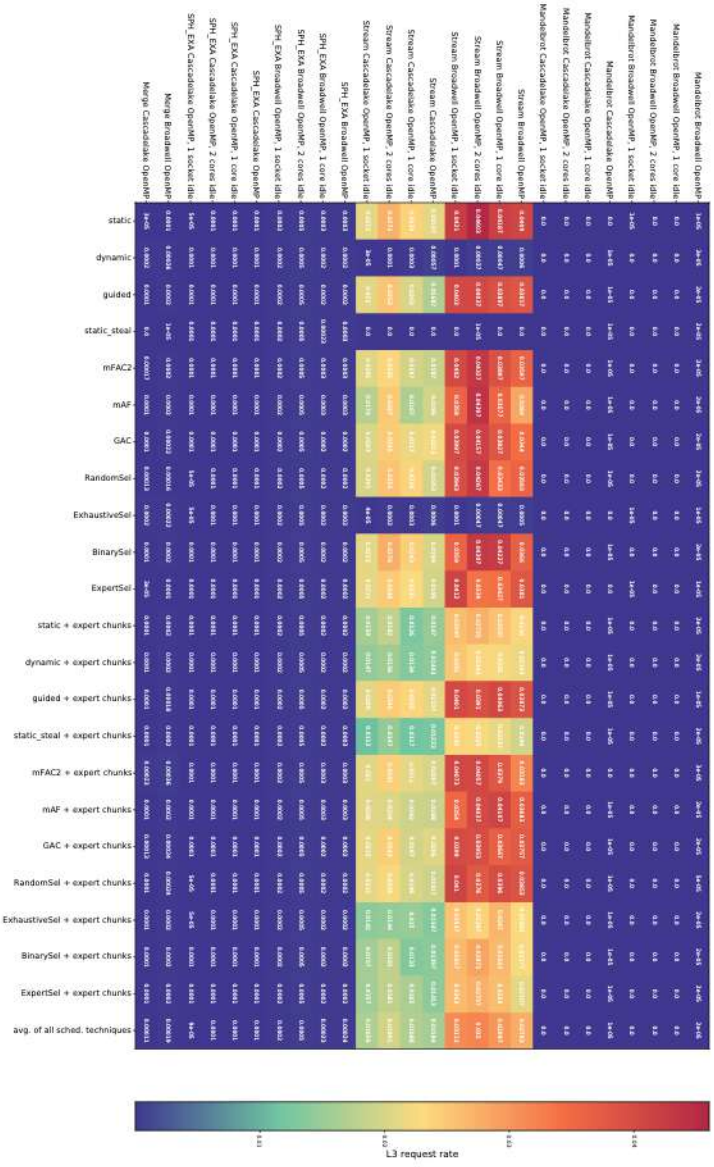


Figure 5.27: L3 request rate for the different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

Since there is no L3 cache on KNL there are no measurements for this system. The L3 request rate (figure 5.27) for Mandelbrot and SPH_EXA is very low on Broadwell and Cascadelake. The difference between Broadwell and Cascadelake for the application Stream is quite high. This difference between the computing systems is also visible in the L2 request rate (figure 5.21). Broadwell has probably a higher request rate because the cache is a bit smaller than on Cascadelake. The scheduling techniques dynamic, static_steal and ExhaustiveSel have a much lower L3 request rate for Stream.

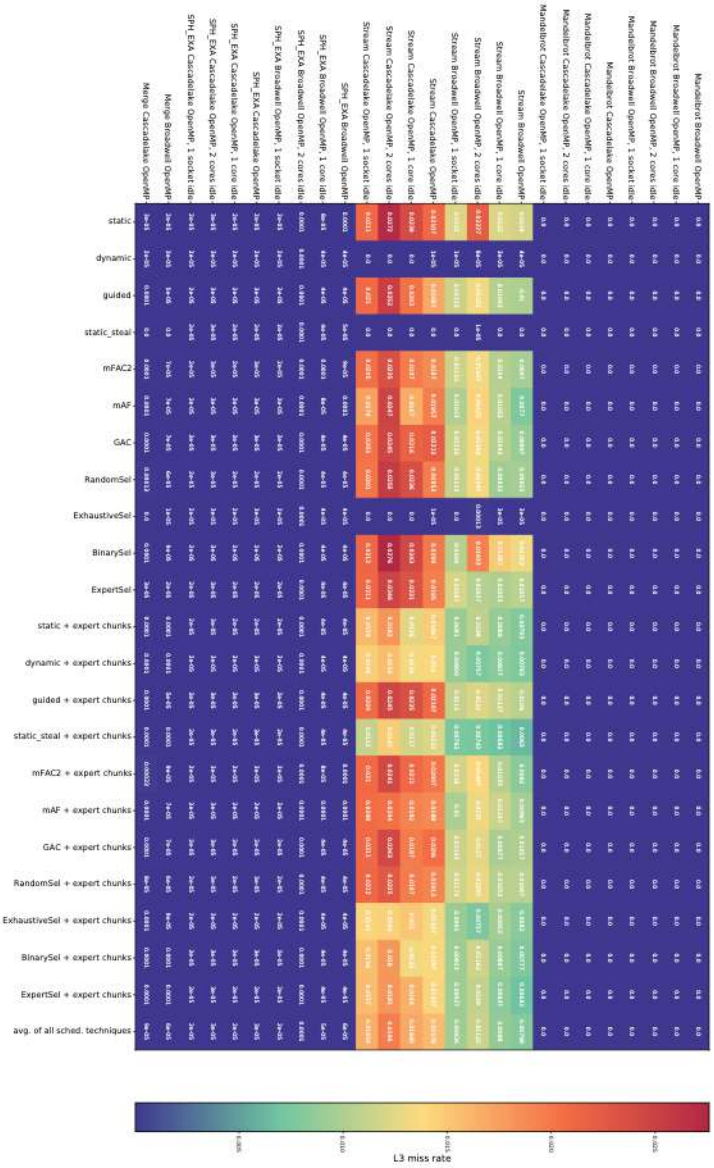


Figure 5.28: L3 miss rate for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

For the application Stream, the L3 miss rate (figure 5.28) on Cascadelake is higher than on Broadwell. The L2 miss rate (figure 5.22) is lower on Cascadelake than on Broadwell. This highlights that the same application on different systems can have different limitations. The L3 miss rate for the other applications is quite low.

For Stream with the scheduling techniques dynamic, static_steal, and ExhaustveSel the recorded cache misses on L3 is much lower than with other scheduling techniques. This is also similar to the L2 miss rate (figure 5.22). When comparing this with the memory bandwidth on L3 (figure 5.30), we see that these scheduling techniques have the lowest

bandwidth. The application requested less data per second. Therefore a higher rate of this data arrived in time.

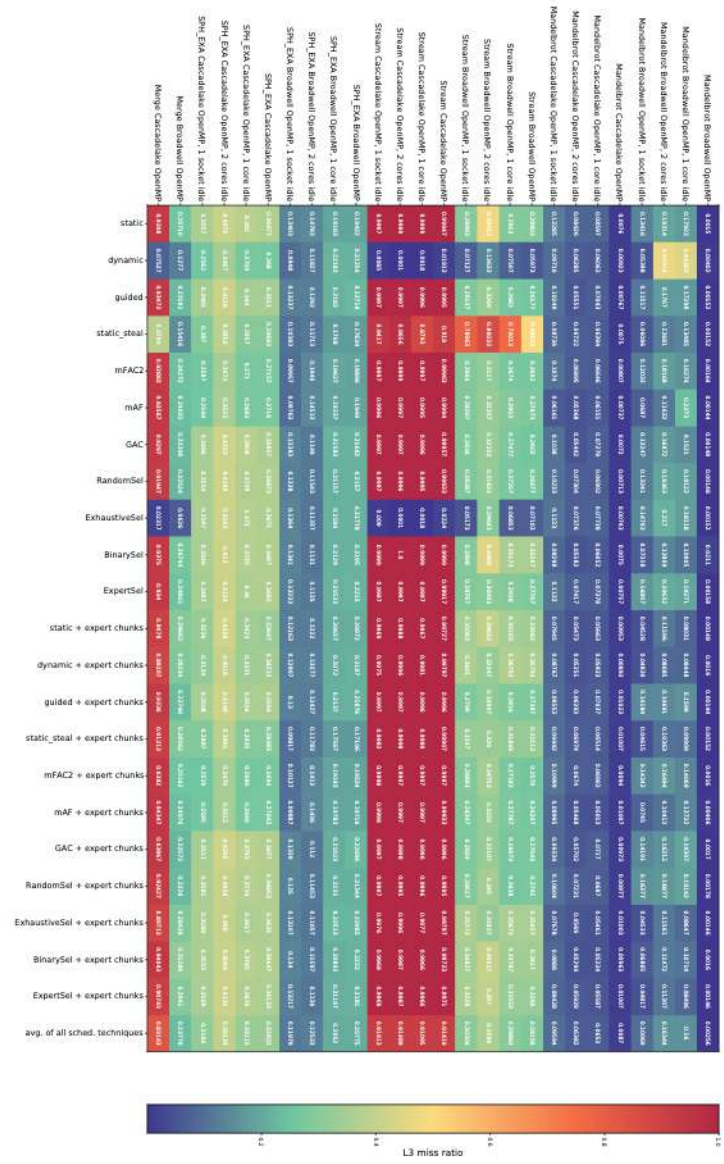


Figure 5.29: L3 miss ratio for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

The L3 miss ratio (figure 5.29) is higher on Cascadelake than on Broadwell for all applications except Mandelbrot. The results for Stream and Merge are interesting. Except for the scheduling techniques dynamic and ExhaustiveSel without expert chunks, nearly all L3 accesses were missed. When comparing the L3 miss ratio to the L3 data volume (figure 5.31), we see that these two scheduling techniques have a much higher L3 data volume than other techniques. It seems that these two scheduling techniques load a lot of data from the

L3 cache. This data is loaded in time. For other scheduling techniques, which load only a small amount from L3 the data is not loaded in time which leads to a very high miss ratio.

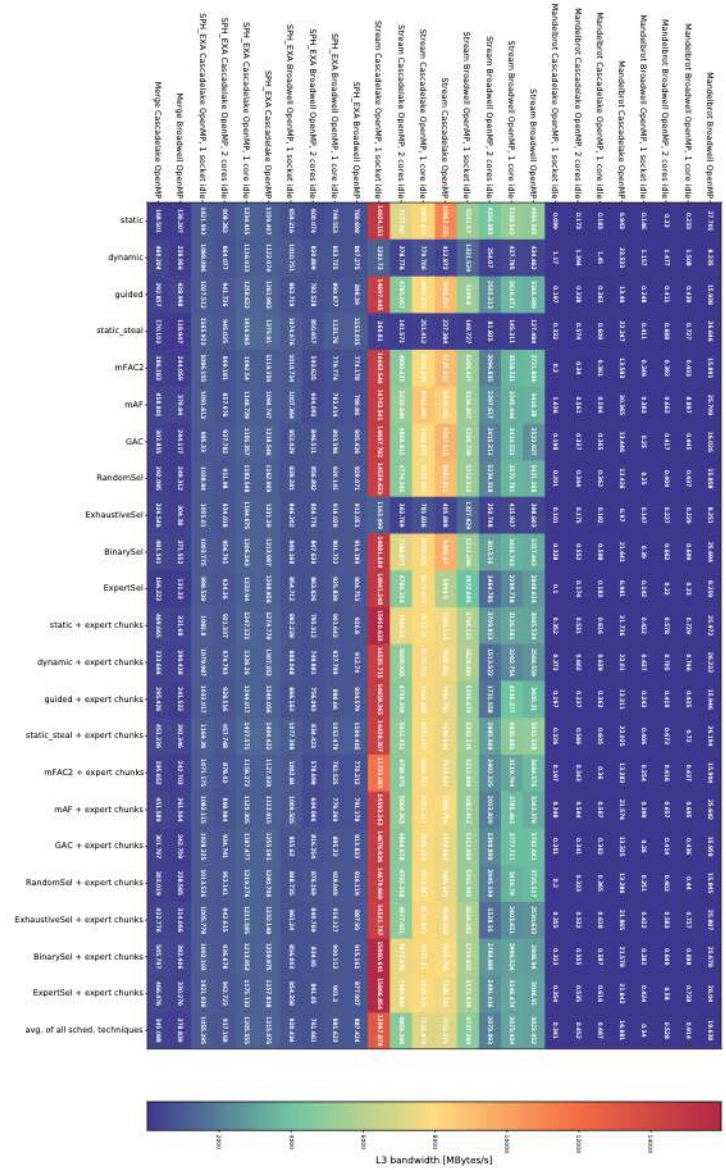


Figure 5.30: L3 bandwidth for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

The bandwidth between L3 and L2 cache (figure 5.30) for the applications Mandelbrot and Stream is similar to the bandwidth between L2 and L1 (figure 5.25). For Mandelbrot the L2 bandwidth is between 0.5 and 40 MBytes/s, L3 bandwidth is between 0.3 and 20 MBytes/s. The bandwidths for Stream are between 2200 and 9000 MBytes/s for L2 and 2000 to 14,000 MBytes/s for L3. For SPH_EXA the L2 and L3 bandwidths are different. The L2 bandwidth is between 2200 and 5000 MBytes/s, the L3 bandwidth is between 800 1200 MBytes/s. We

still investigate why SPH_EXA has such a different bandwidth for the cache levels. For Stream, the same scheduling techniques as previously show much lower L3 bandwidth than the others.

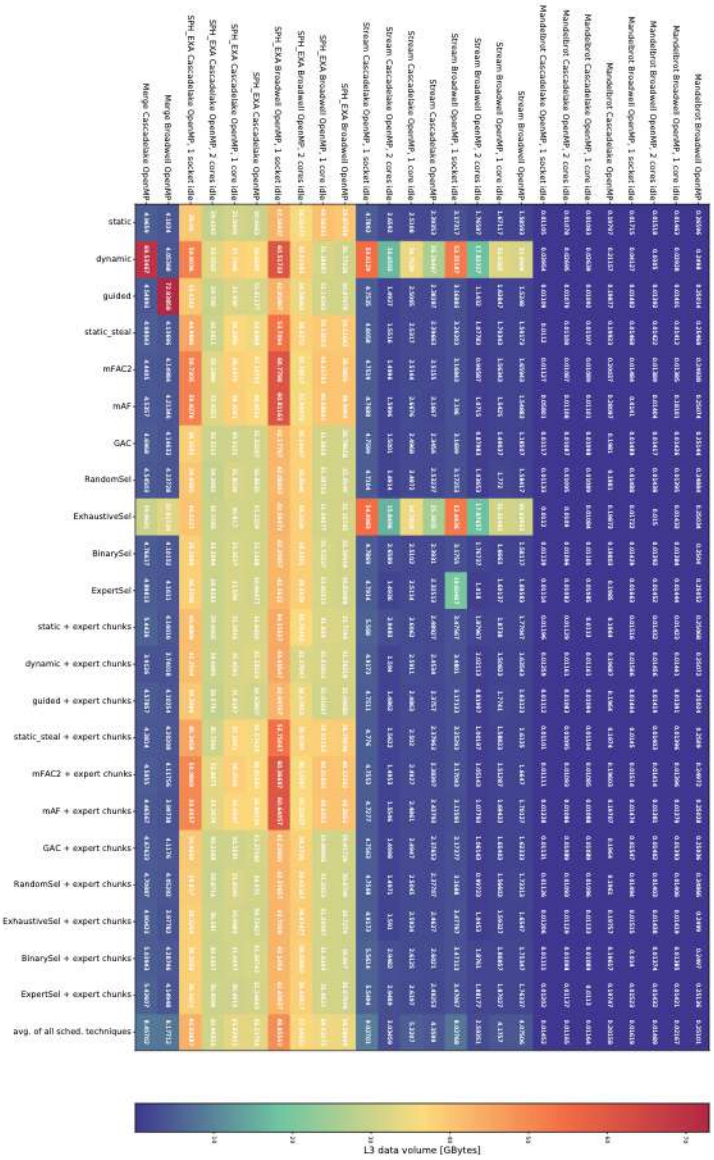


Figure 5.31: L3 data volume for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

The L3 data volume (figure 5.31) is similar to the L2 data volume (figure 5.26). SPH_EXA has the highest data volume compared with the other applications. So is the data volume higher for executions with one idle socket than when more cores are utilized. As already mentioned in the paragraph about the L3 miss ratio (figure 5.29), the scheduling techniques dynamic and ExhaustiveSel have much higher data volume than other scheduling techniques.

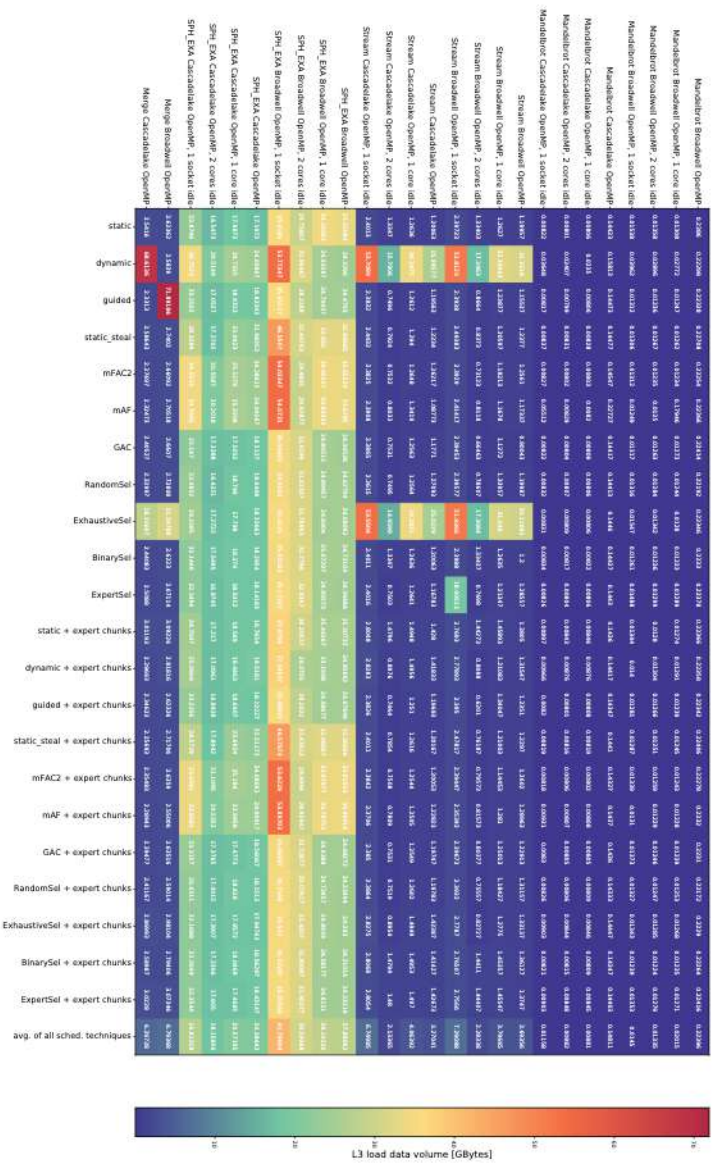


Figure 5.32: L3 load data volume for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

The loaded data volume (figure 5.32) has the same pattern as the total data volume (figure 5.31). The same scheduling techniques, applications, and thread configuration show higher loaded data volumes than others.

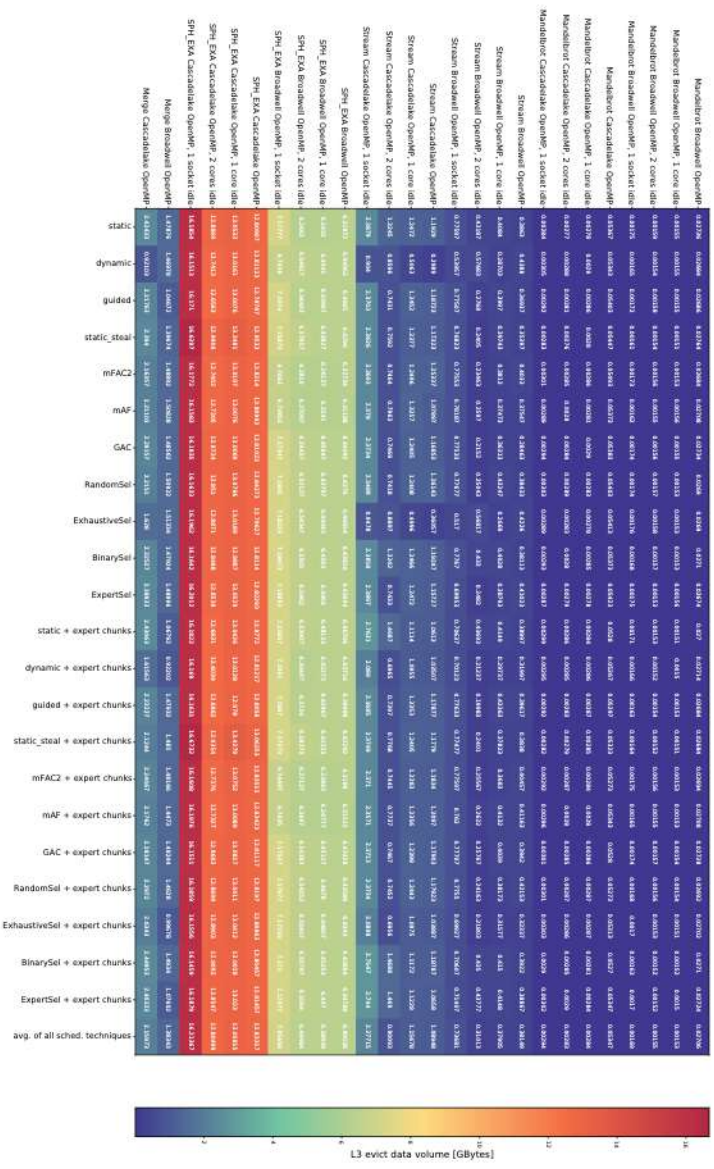


Figure 5.33: L3 evict data volume for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

The evicted data volume (figure 5.33) does not differ much between different scheduling techniques. There is no big difference between one and two idle cores because the L2 cache size remains. The idle cores are on different sockets. So they use different L2 cache groups. For Stream and SPH_EXA the executions with one idle socket have a higher evicted data volume than with more used cores. This is probably because the usable L2 cache is smaller for one socket.

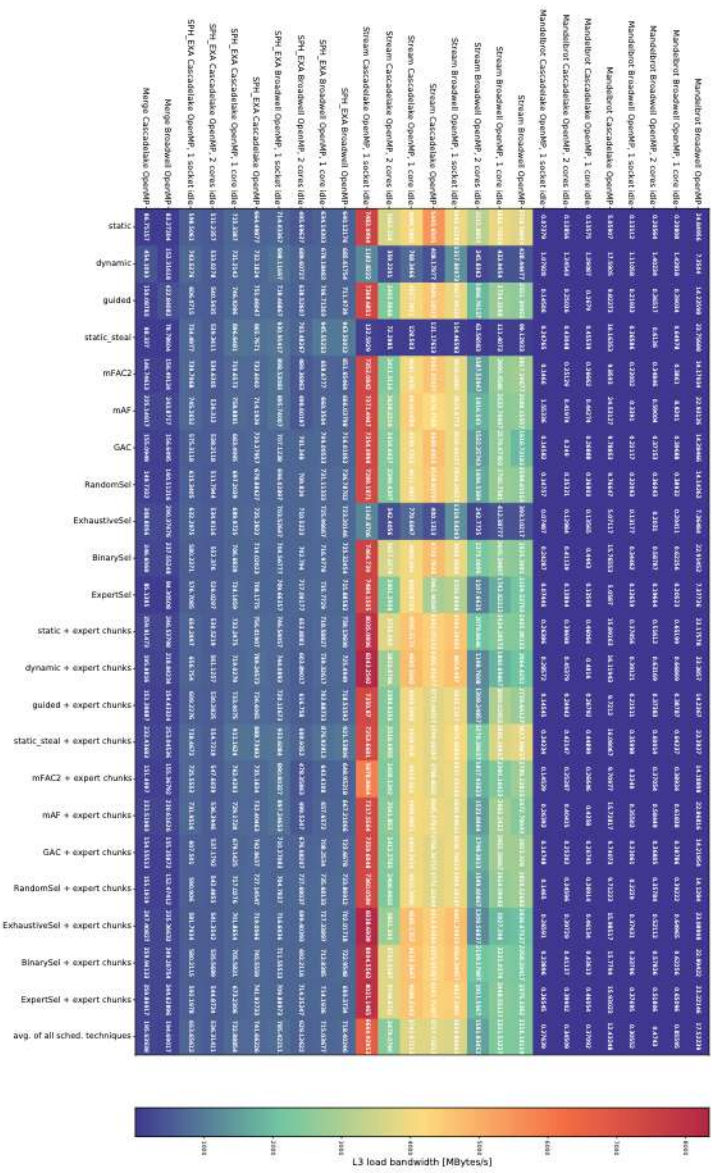


Figure 5.34: L3 load data bandwidth for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

The load bandwidth for L3 (figure 5.34) is highest for the application Stream. SPH_EXA loads a bit of data and Mandelbrot much less. Stream with one idle socket loads more data because the available L3 cache is much smaller than when the second socket is used. More data is loaded and evicted (figure 5.35) This difference between the thread configurations is higher on Cascadelake than on Broadwell.

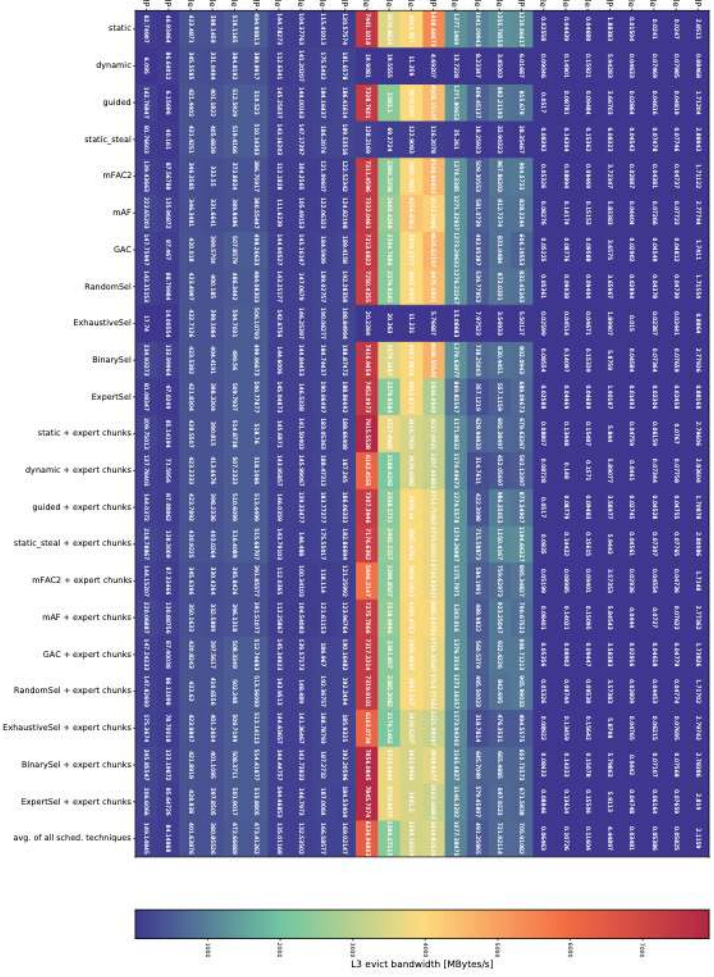


Figure 5.35: L3 evict data bandwidth for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

Both the load (figure 5.34) and evict data bandwidth (figure 5.35) do not differ much from the total L3 bandwidth (figure 5.30) We observe the same pattern on all three graphs. The same scheduling techniques have the lowest bandwidth. Also, the bandwidth is much higher for Stream than for other applications.

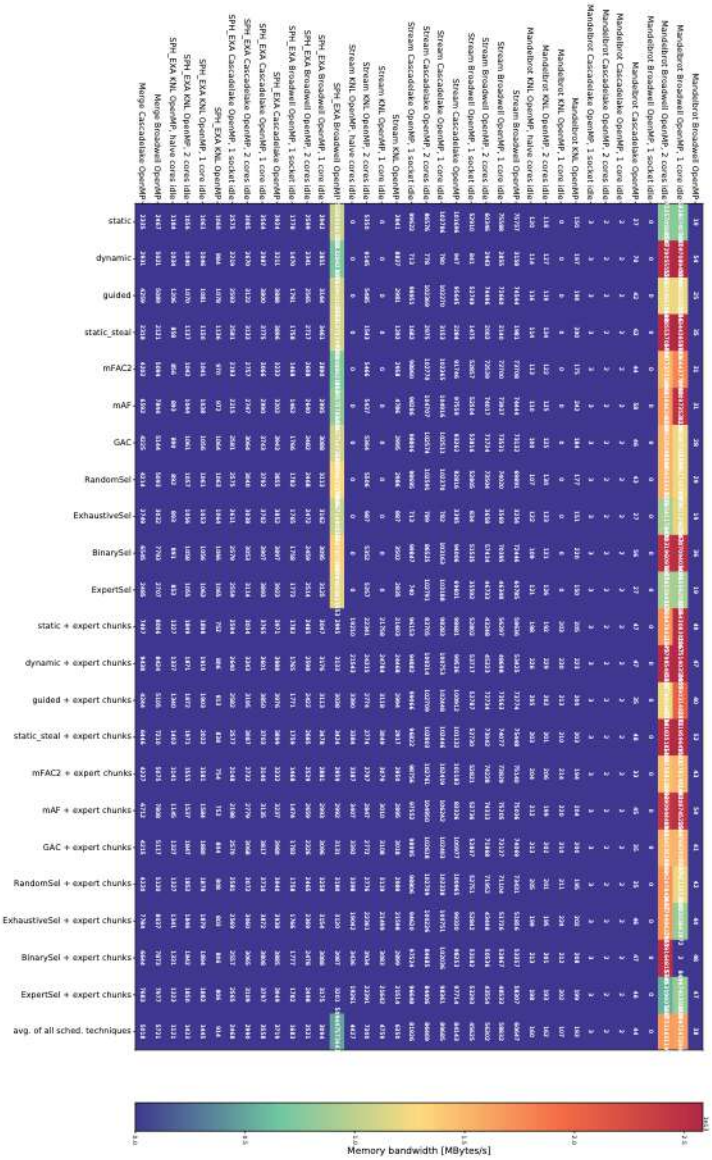


Figure 5.36: Memory bandwidth for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

According to the data in figure 5.36, Mandelbrot has on Broadwell with one or two idle cores a bandwidth of about 16,000,000,000,000 MBytes/s. Also, the measurements for SPH_EXA on Broadwell with all threads report up to 5,859,771,276,596 MBytes/s. This can not be correct. The other more realistic measurements for Mandelbrot are between 2 and 200 MBytes/s.

In figures 5.37 and 5.38 there is the memory bandwidth divided between the memory read and write. This shows similar values for Mandelbrot and SPH_EXA.

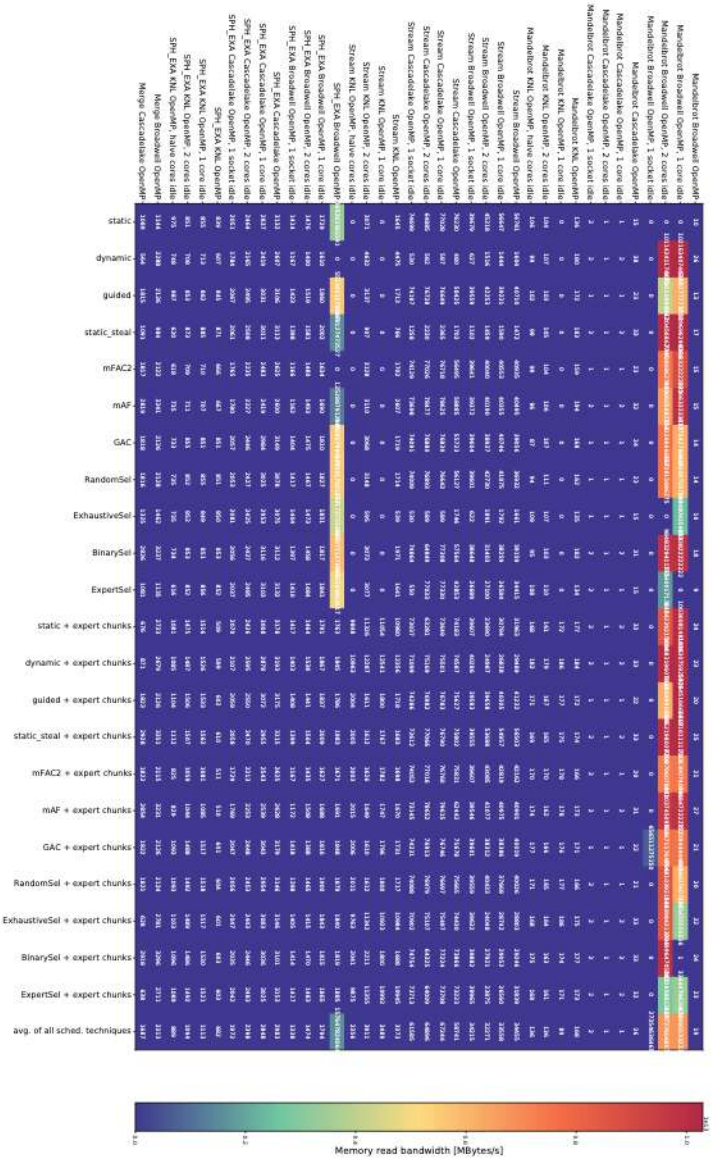


Figure 5.37: Memory read bandwidth for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

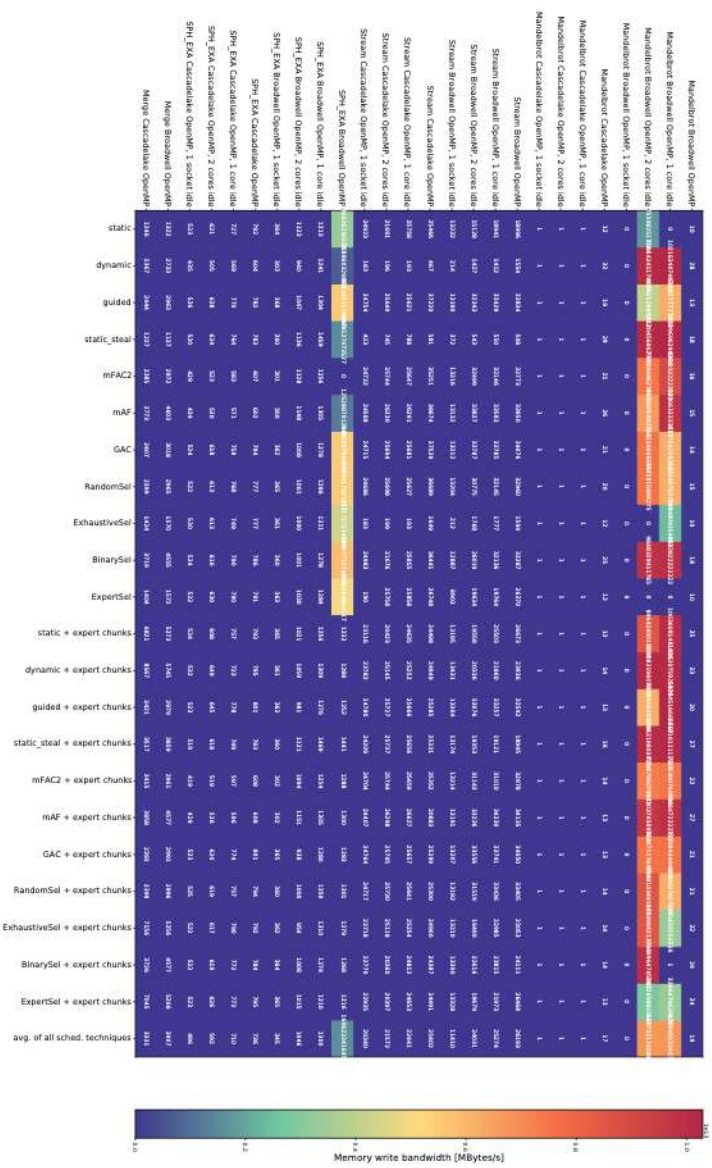


Figure 5.38: Memory write bandwidth for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

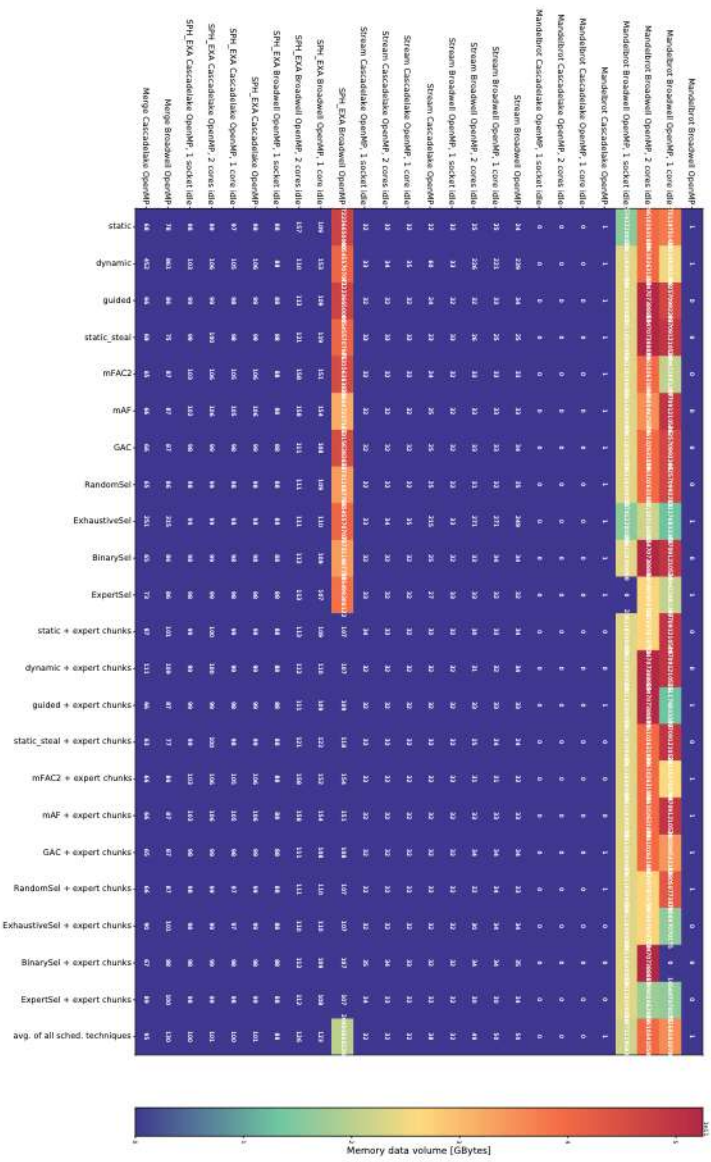


Figure 5.39: Memory data volume for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

In figure 5.39 is the total Memory volume. In figures 5.41 - 5.40 is the data volume divided between reads and writes. This data shows similar to the memory bandwidth (figure 5.36) values that are too high for Mandelbrot and SPH_EXA on Broadwell. We leave it to future work to investigate why Likwid reports these values with some applications and thread configurations.

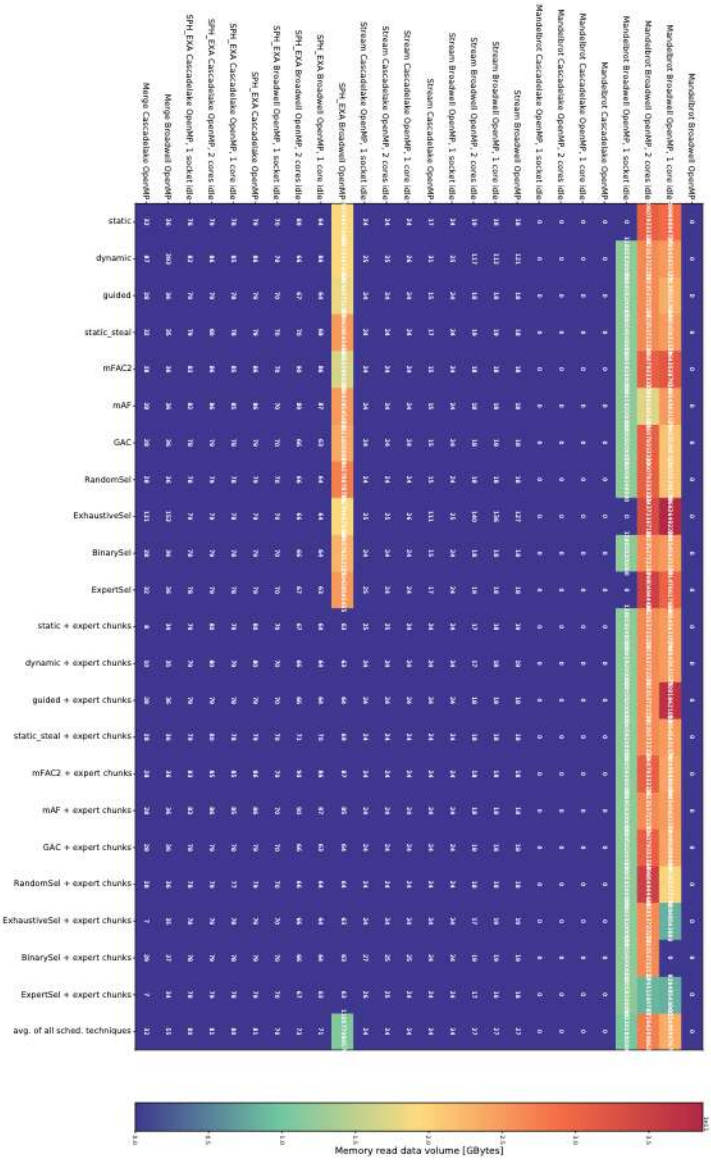


Figure 5.40: Memory read data volume for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

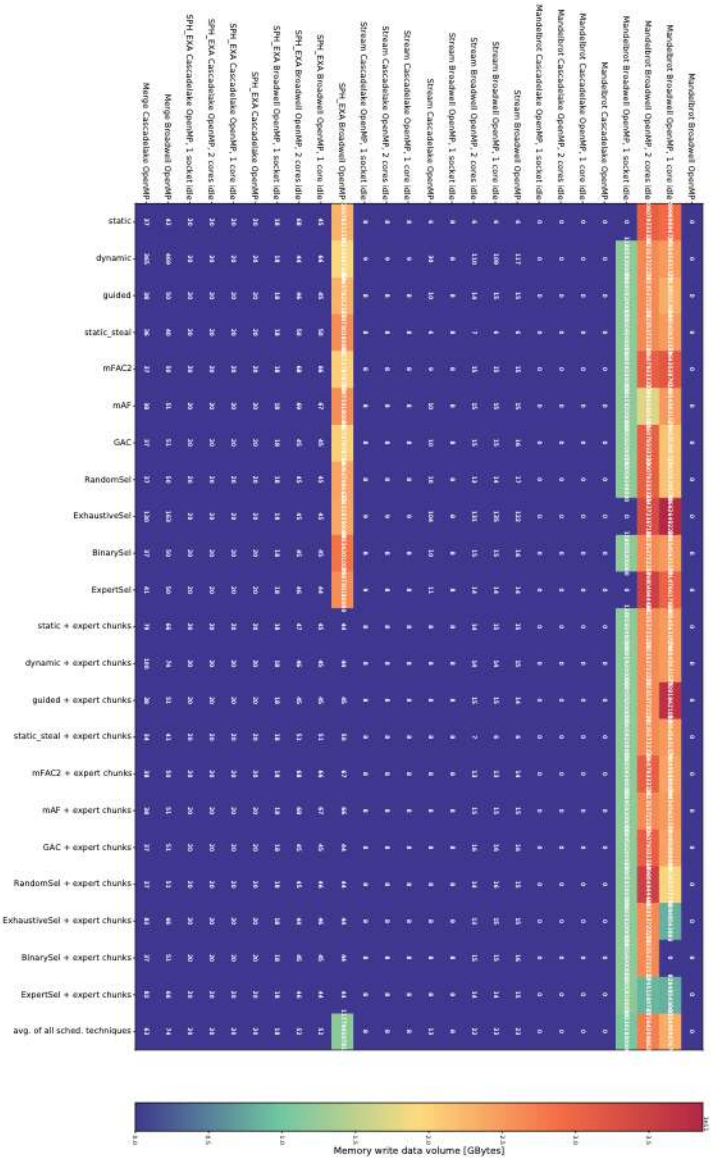


Figure 5.41: Memory write data volume for the different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

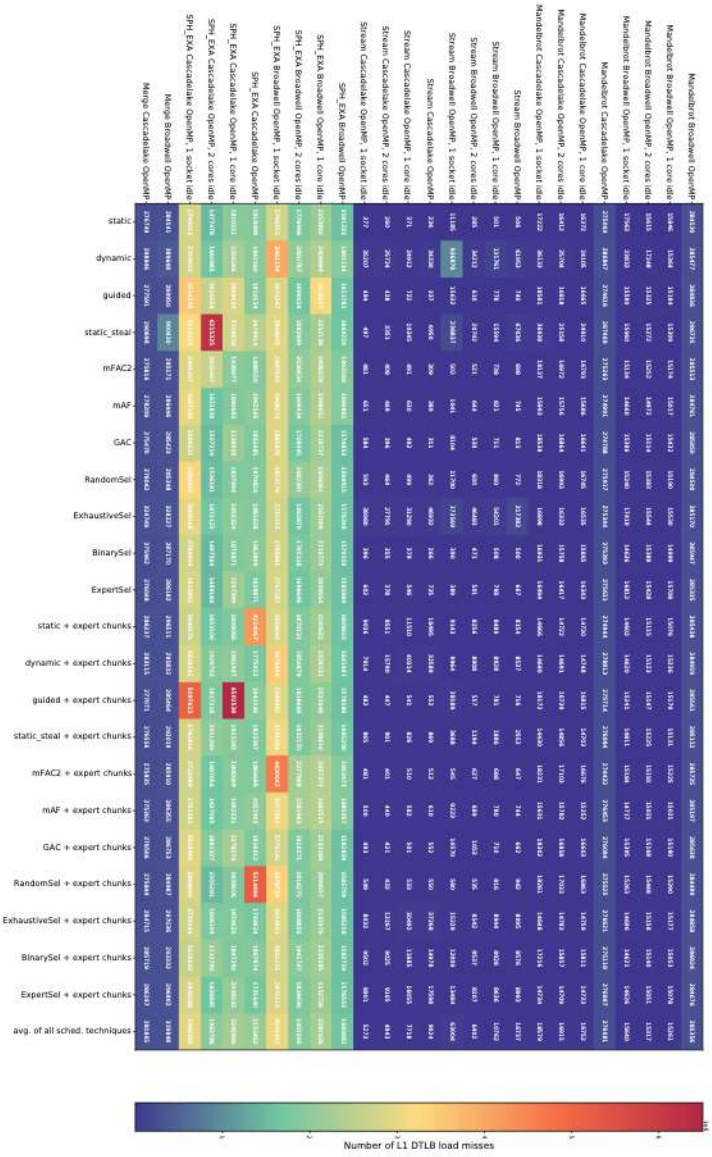


Figure 5.42: L1 DTLB load misses for different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

As with other Likwid counter groups, the data from KNL is not good. Therefore, we leave the results out for the analysis of the translation lookaside buffer (TLB). For the TLB misses in figure 5.42, 5.43, and 5.44 we see that SPH_EXA has the most TLB misses compared with other applications. Mandelbrot has very few data accesses, and the data access of Stream is not random. Stream accesses three matrices. So the OS can probably predict which data is accessed next. SPH_EXA is not that predictable, which leads to much more TLB misses. Surprising is the difference in ITLB misses of SPH_EXA on Broadwell and Cascadelake. On Broadwell, SPH_EXA suffers much fewer ITLB misses than on Cascadelake. It is also

interesting that some extreme outliers dominate the heat-map, especially for DTLB store misses.

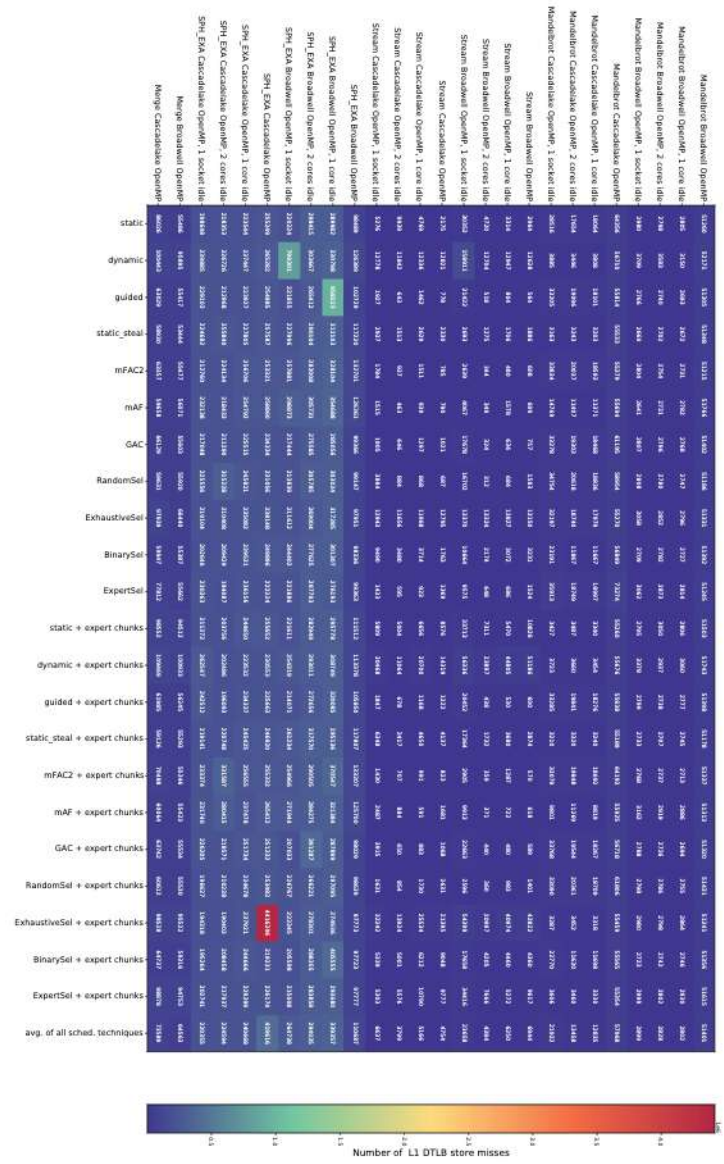


Figure 5.43: L1 DTLB store misses for different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

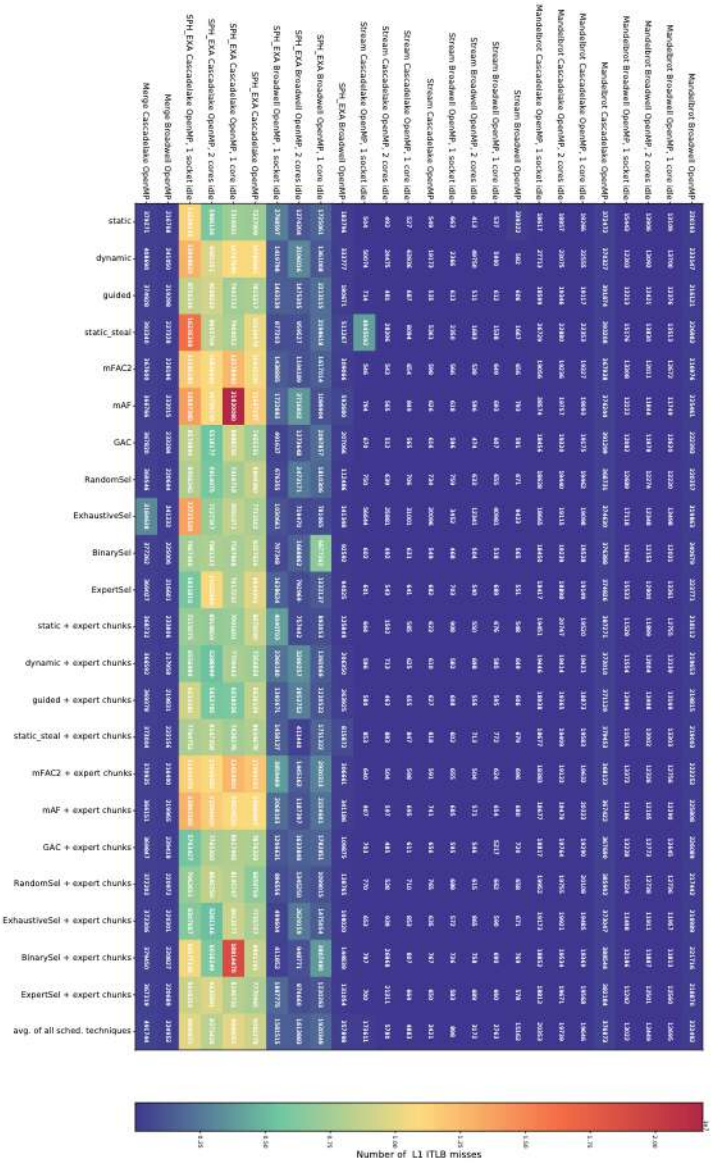


Figure 5.44: L1 ITLB misses for different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

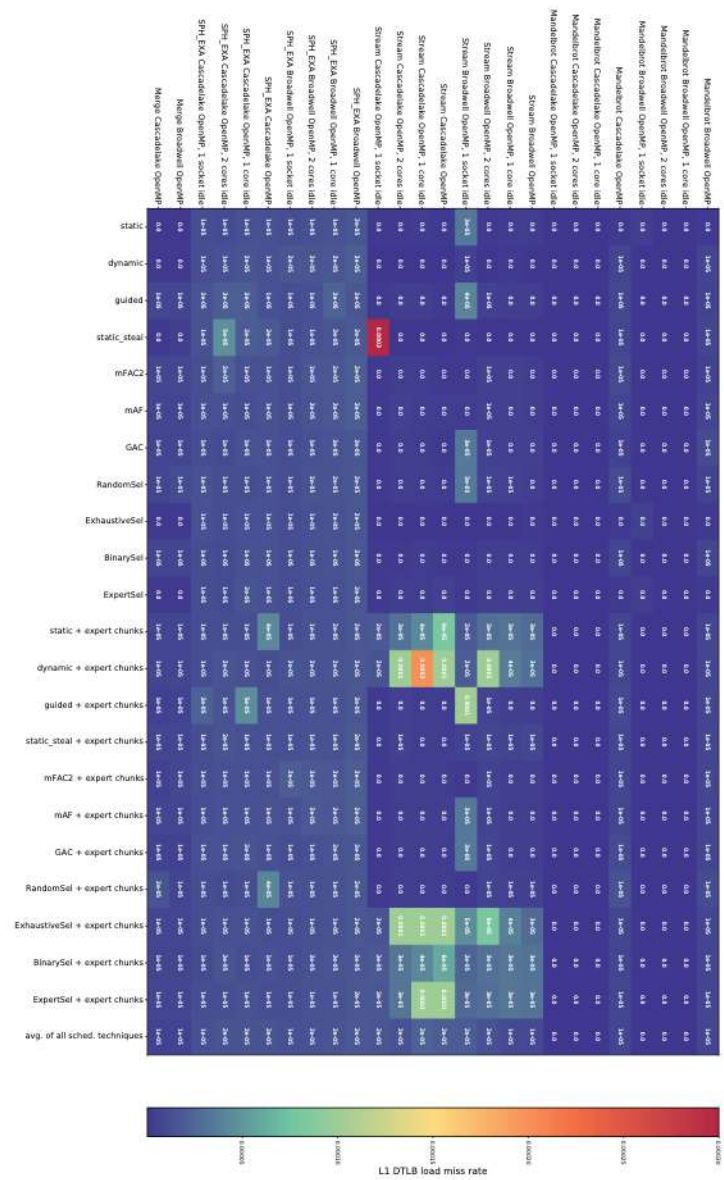


Figure 5.45: L1 DTLB load miss rate for different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

The TLB miss rate, see figures 5.45, 5.46, and 5.47, should be low for high performance. As we can see this is the case for all applications and systems. The instruction TLB misses are a bit higher for SPH_EXA than for the other applications. Since SPH_EXA does more complex calculations than Mandelbrot or Stream this higher ITLB miss rate sims right. Notable is the difference for SPH_EXA on Broadwell and Cascadelake. There is a lower ITLB miss rate on Broadwell. Stream has the highest data TLB store miss rate, especially with the scheduling techniques dynamic and ExhaustveSel with expert chunks.

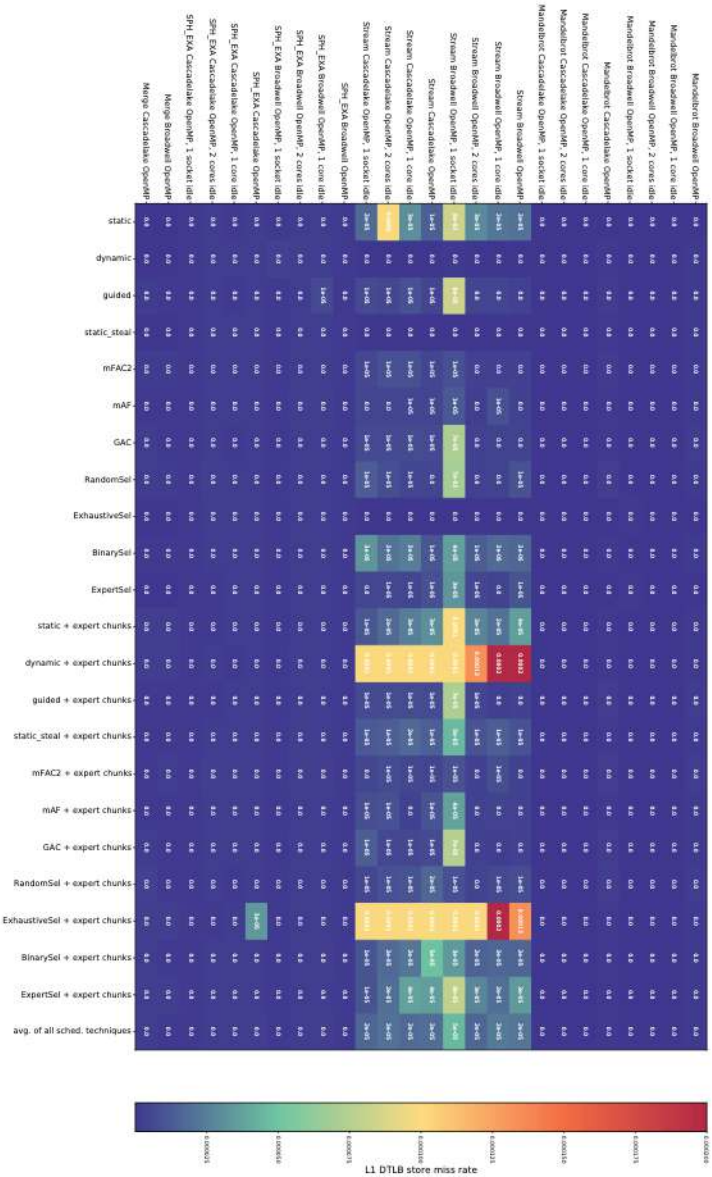


Figure 5.46: L1 DTLB store miss rate for different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

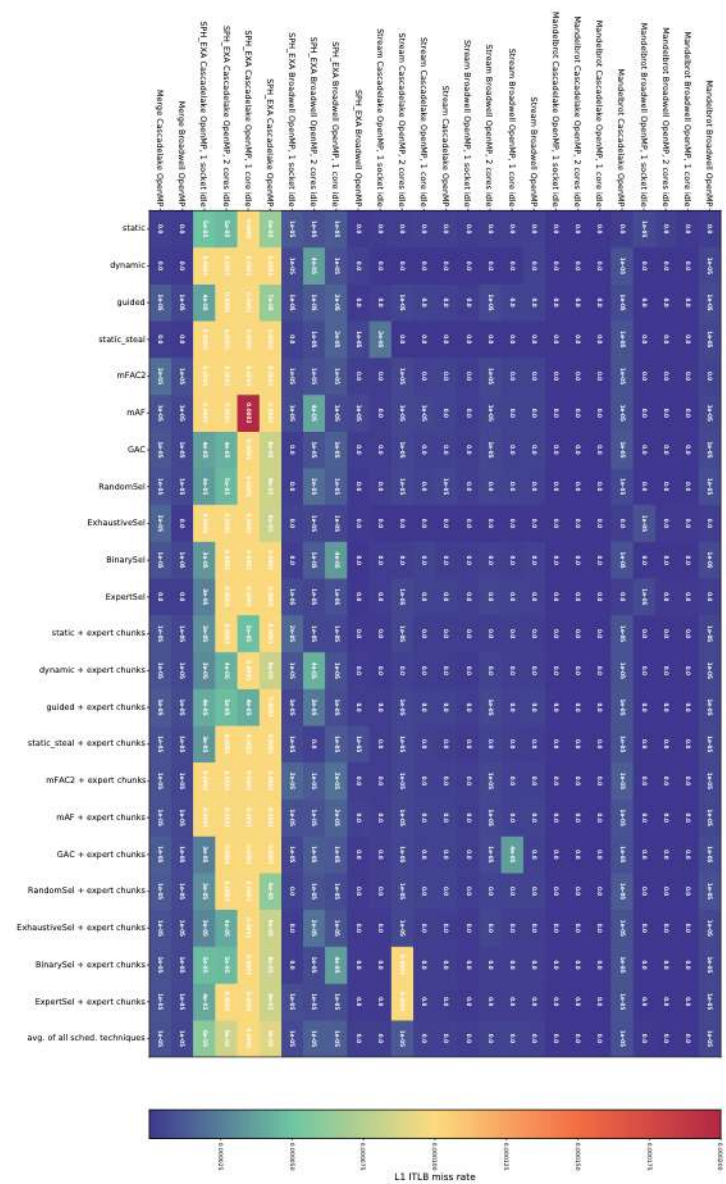


Figure 5.47: L1 ITLB miss rate for different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

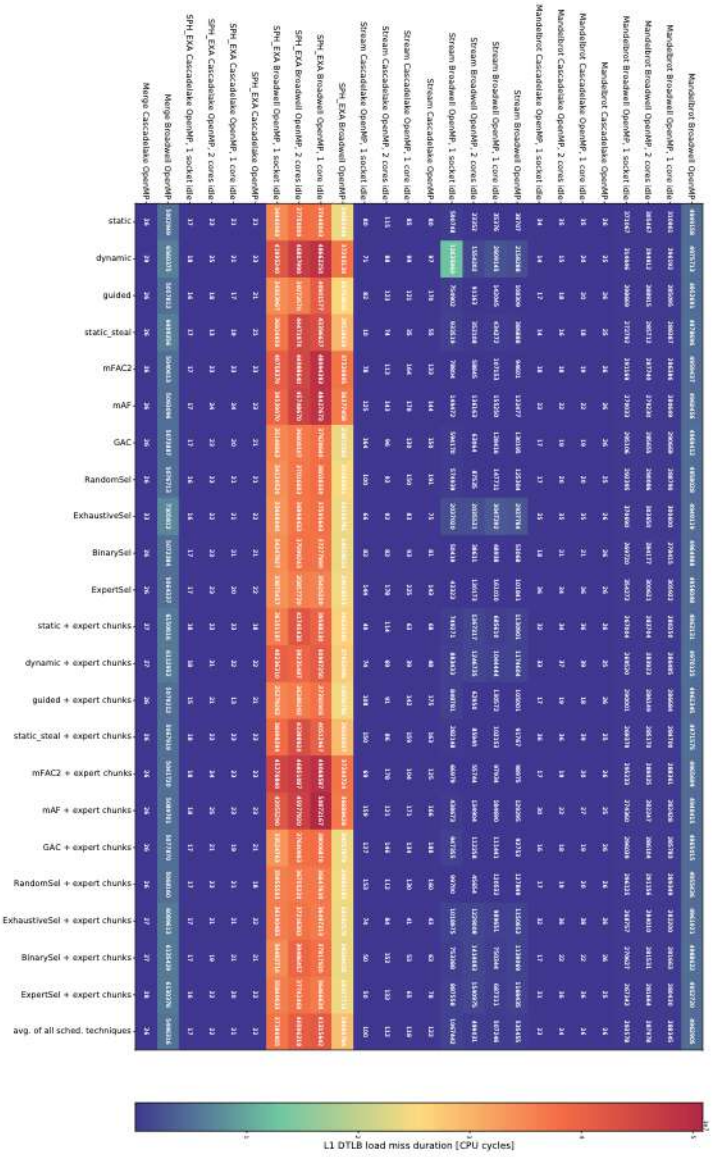


Figure 5.48: L1 DTLB load miss duration for different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

The TLB miss duration is in figures 5.48, 5.49, and 5.50. We observe a big difference between the Systems. We assume that this is caused by different time units. The documentation of these counters states that the miss duration is measured in cycles. Perhaps this is not correctly converted on some systems.

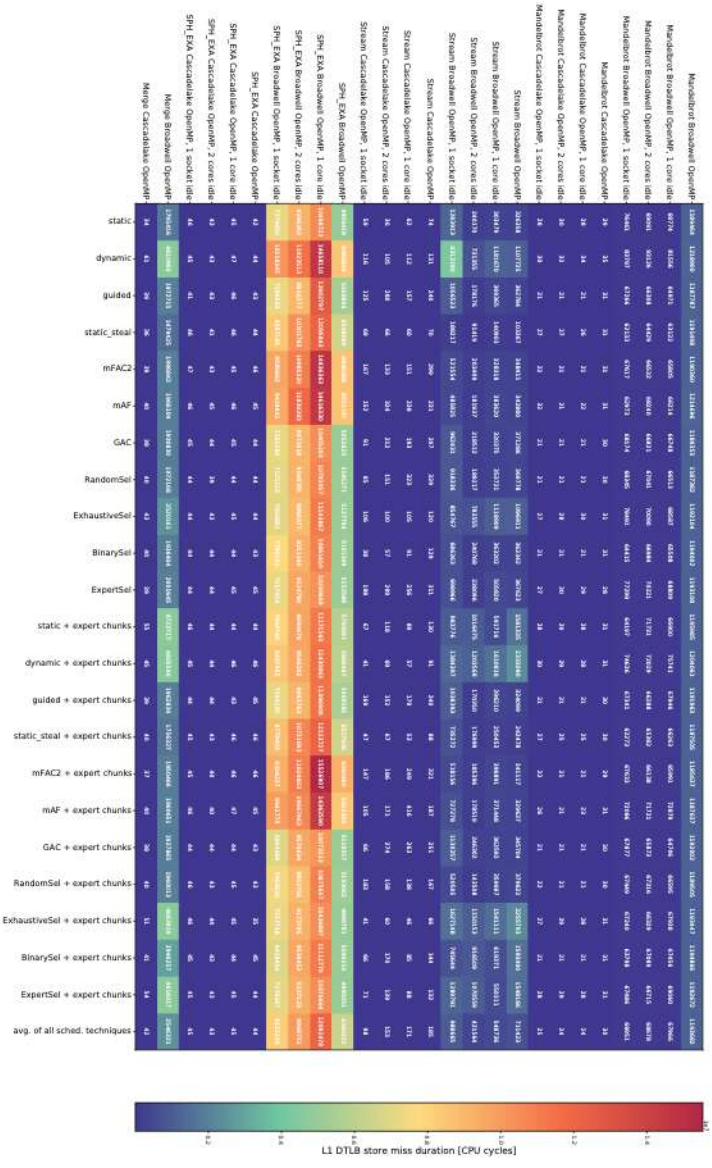


Figure 5.49: L1 DTLB store miss duration for different applications on Broadwell and Cascadelake. With different thread configurations and scheduling techniques.

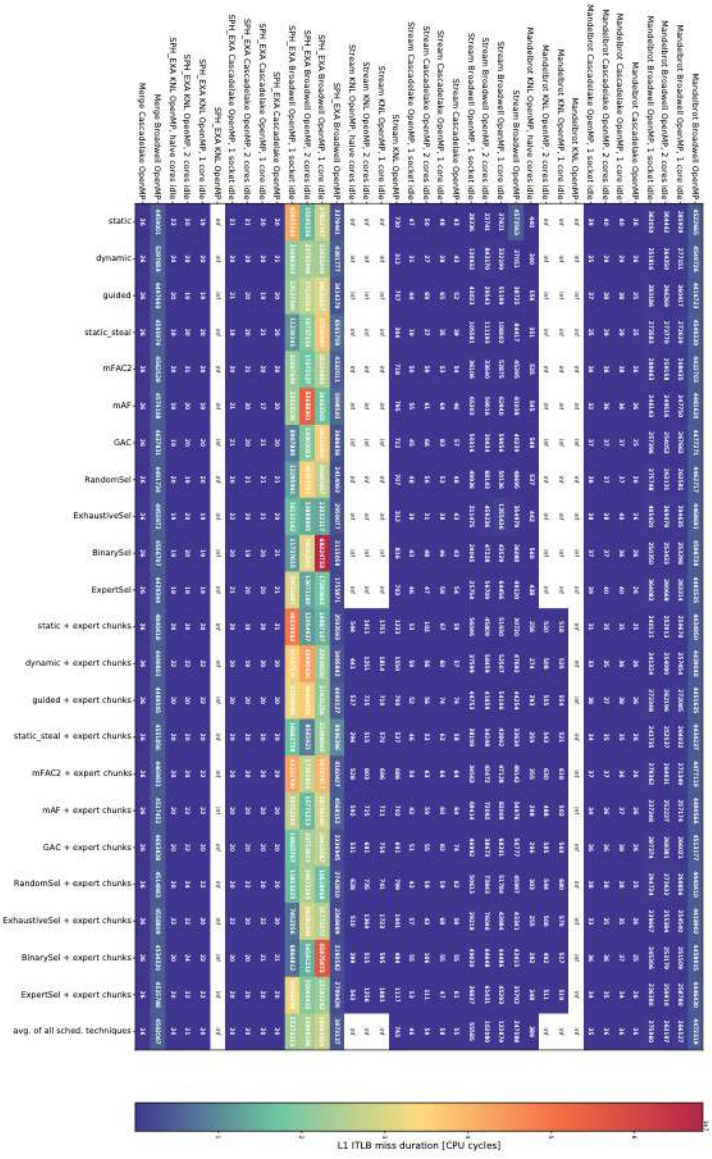


Figure 5.50: L1 ITLB miss duration for different applications on Broadwell, Cascadelake, and KNL. With different thread configurations and scheduling techniques.

5.2 Results for Parallelization Methods

In this section, we compare the parallelization methods MPI and OMP. All executions used all available cores on the respective computing system. The goal was to find out how the OS influence differs between MPI processes and OpenMP threads. Executions with only OpenMP. We also compare executions with hybrid MPI and OpenMP. Due to time constraints, we could perform these experiments only for SPH_EXA on Broadwell and

Cascadelake. We could only perform the measurements with perf. Likwid provides a tool to monitor MPI applications. Due to technical problems and time constraints we did not perform the measurements about the memory and cache. In section 5.2.1 we present the results for thread migration events. The measurements for the context switches are shown in section 5.2.2 and the idle time events in section 5.2.3.

5.2.1 Thread Migration

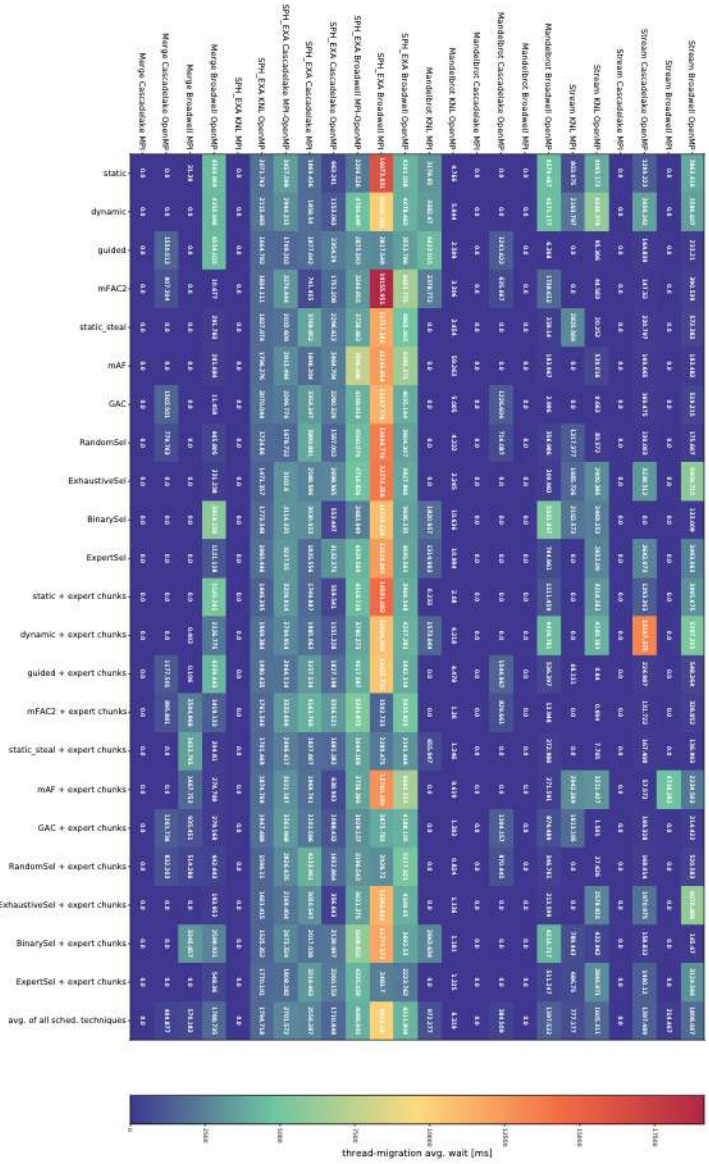


Figure 5.51: Average wait time for thread migration events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

A loot of measured thread migration events have a wait time of zero milliseconds (figure 5.51). The number of migration events is often very low, as we can see in figure 5.54, This means that on most cores at most one migration event occurs. The wait time for the first migration events is zero because there was no previous migration process to measure the elapsed time between the two events. In most cases, the execution with MPI has shorter wait times between the thread migrations. Some exceptions are Mandelbrot on KNL, SPH_EXA on Broadwell, and Cascadelake. On Broadwell, the hybrid MPI and OpenMP versions have higher wait times compared to the OpenMP version. On Cascadelake there is only a small difference between these Parallelizations.

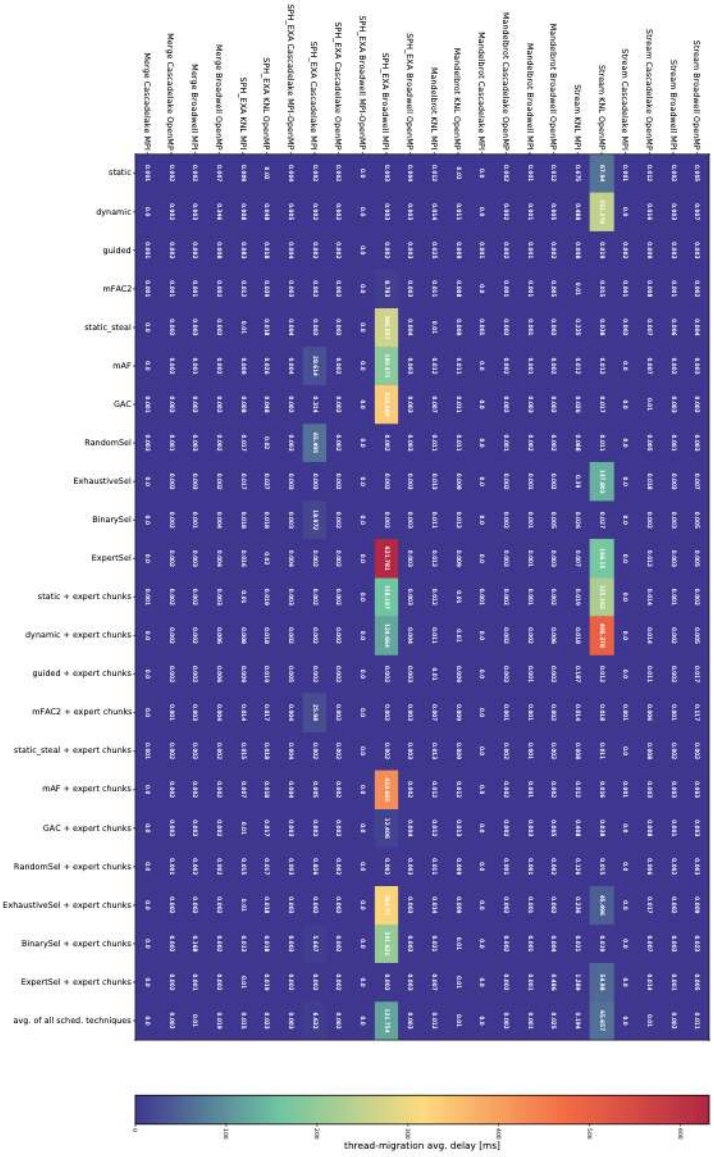


Figure 5.52: Average delay for thread migration events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

The average scheduler delay (figure 5.52) that thread migrations experience is in most cases very low. With only a few migrations the core the process migrates to is most likely idle. So there is probably a low delay. There are very few recorded exceptions to this for SPH_EXA on Broadwell and Stream on KNL. We are still investigating why this outliers occur. The delay for the hybrid version is similar to the OpenMP version.

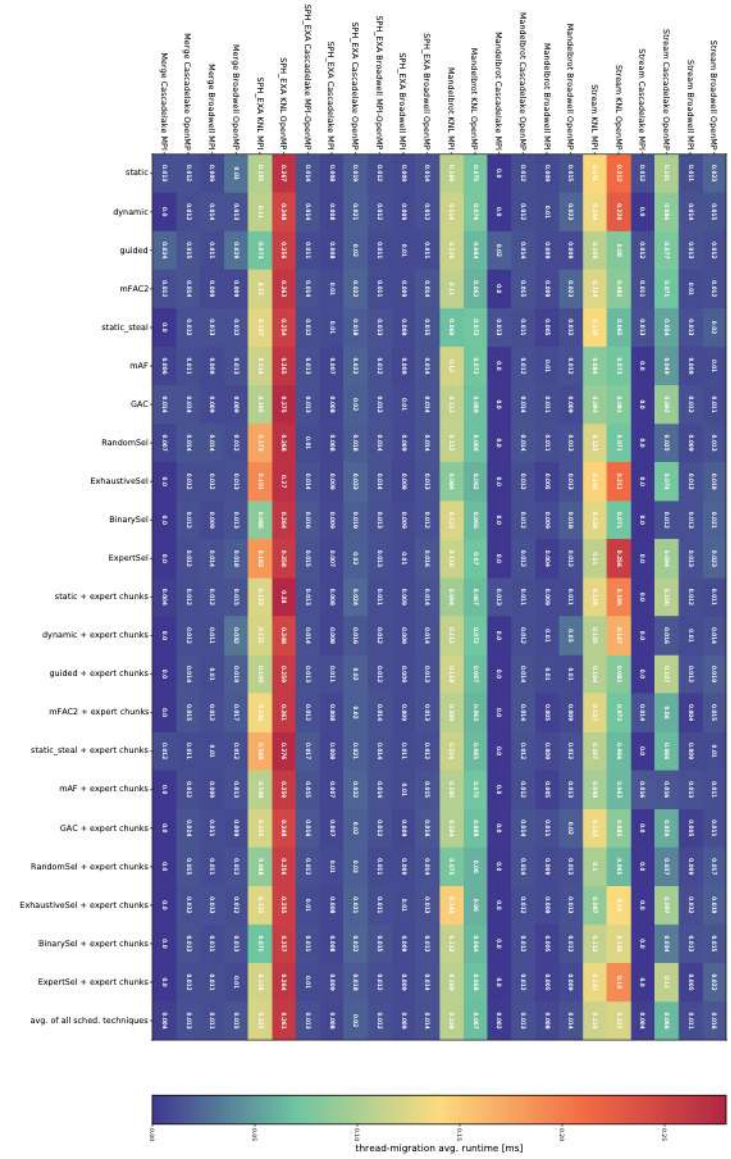


Figure 5.53: Average runtime for thread migration events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

The runtime of a thread migration event (figure 5.53) depends a lot on the computing system, we have already discussed this for the thread migrations in figure 5.3. On Broadwell and Cascadelake thread migrations with OpenMP take much longer than with MPI. This is not what we expected. We expected that the migration with MPI processes takes longer than with OpenMP threads. Because threads are smaller than processes. On Cascadelake the runtime for thread migrations is higher for the hybrid SPH_EXA version than for the OpenMP or MPI version.

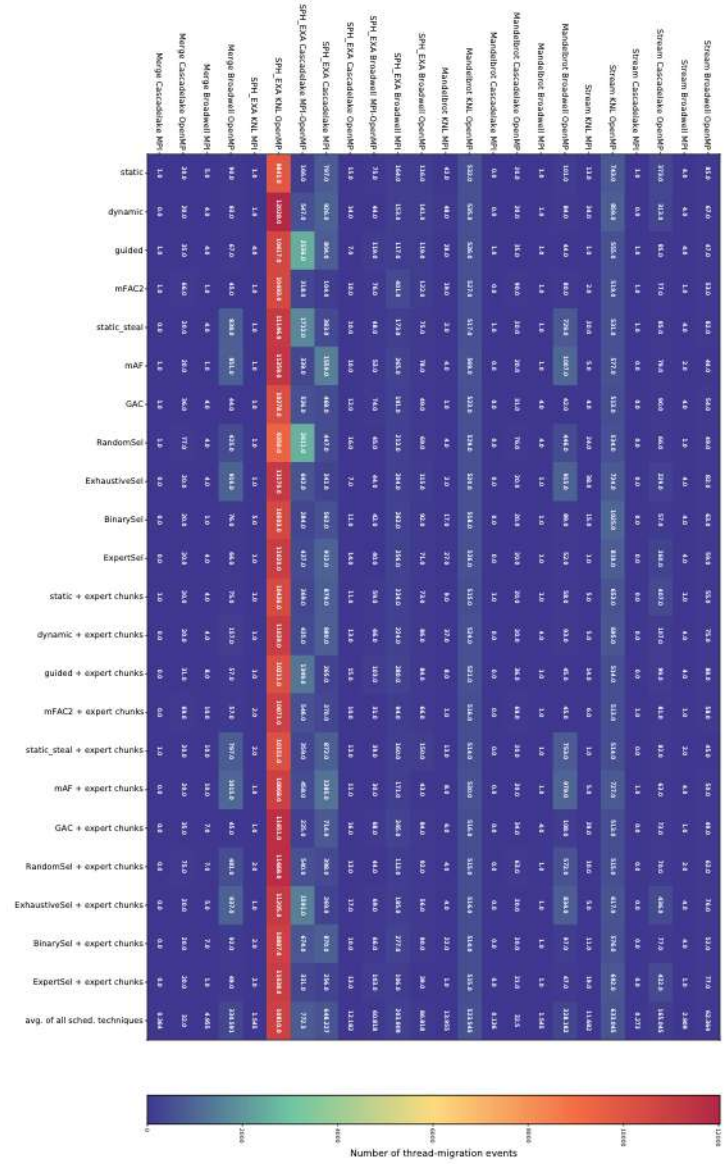


Figure 5.54: The number of thread migration events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

The number of thread migration events (figure 5.54) is much higher for executions with OpenMP than with MPI. The only exceptions for this are SPH_EXA on Broadwell and Cascadelake. Hybrid SPH_EXA on Broadwell has fewer thread migrations than with only MPI or OpenMP. On Cascadelake the number of thread migration is between OpenMP and MPI.

5.2.2 Context Switches

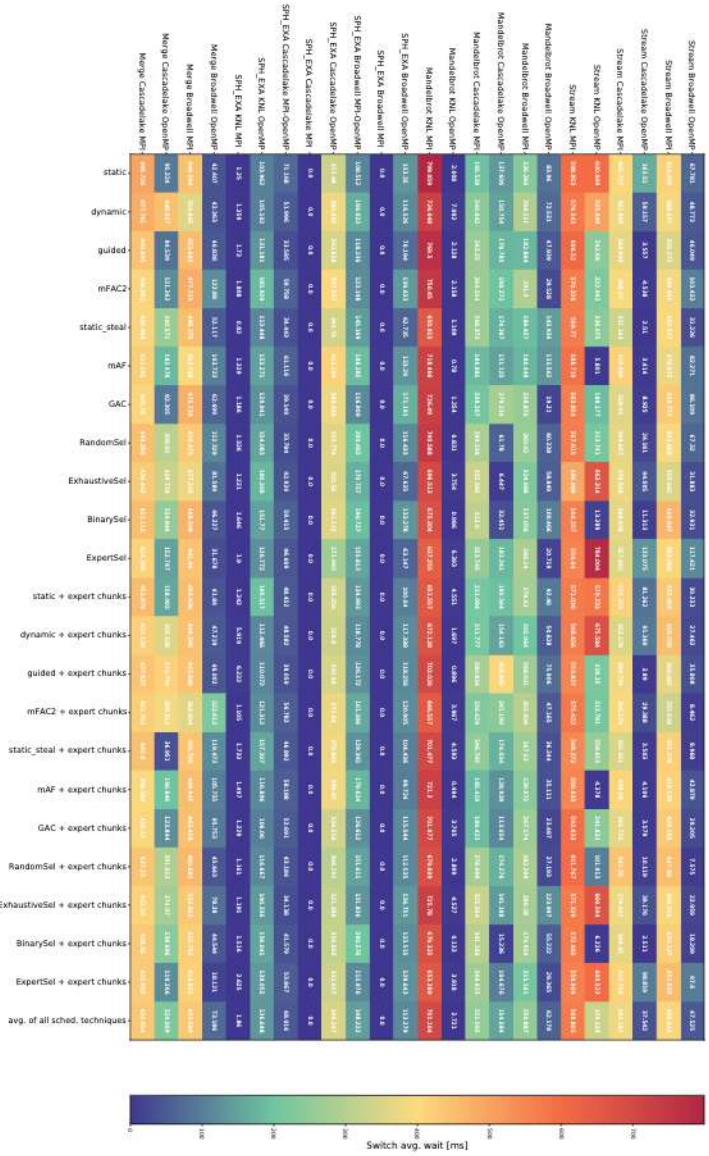


Figure 5.55: The average wait time for context switches events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

Except for SPH_EXA all recorded wait times for context switch events (figure 5.55) took longer for application parallelized with MPI than with OpenMP. This means that with MPI context switches are less frequent. For SPH_EXA with MPI on Broadwell and Cascadelake, the wait time is zero. This means that at most one context switch is recorded. The hybrid SPH_EXA versions perform similar to the versions parallelized with OpenMP.

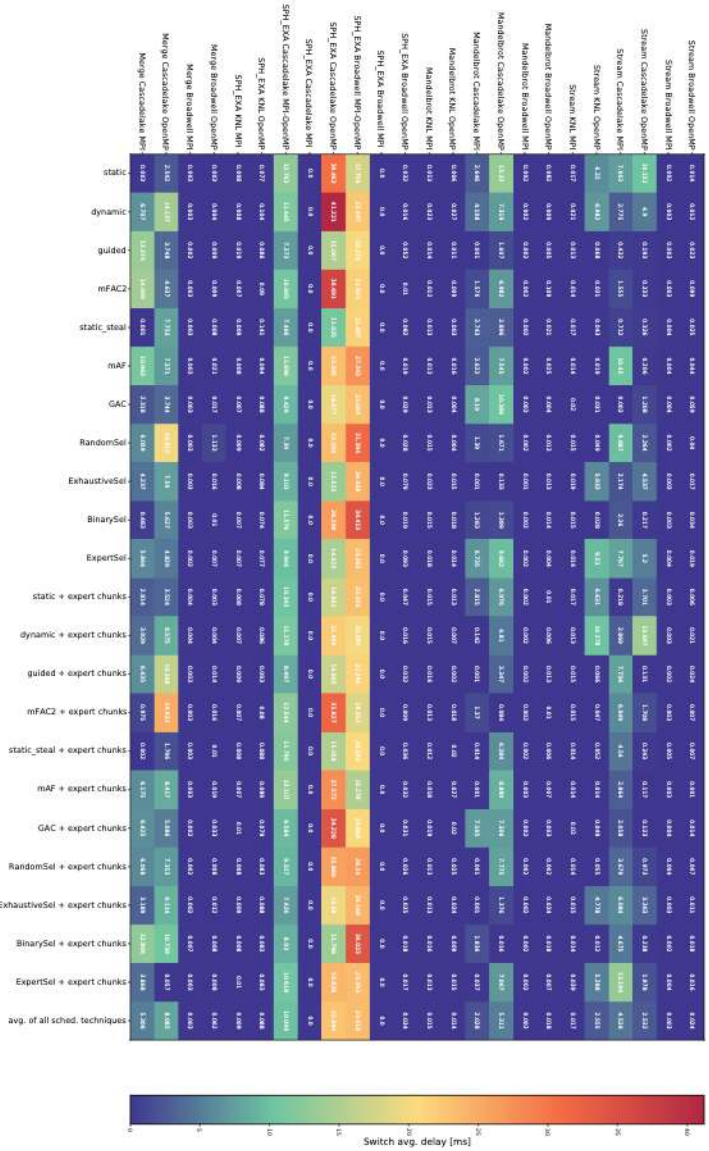


Figure 5.56: The average delay for context switches events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

Except for Stream on Cascadelake, the average delay for a context switch event (figure 5.56) is shorter for executions with MPI. So the scheduler has a lower latency for context switches for applications parallelized with MPI. As for the wait time of context switches (figure 5.55) there are no recorded context switches for SPH_EXA on Broadwell and Cascadelake with MPI. The delay for the hybrid SPH_EXA version on Broadwell is much higher than with OpenMP or MPI. ON Cascadelake this value is between MPI and OpenMP.

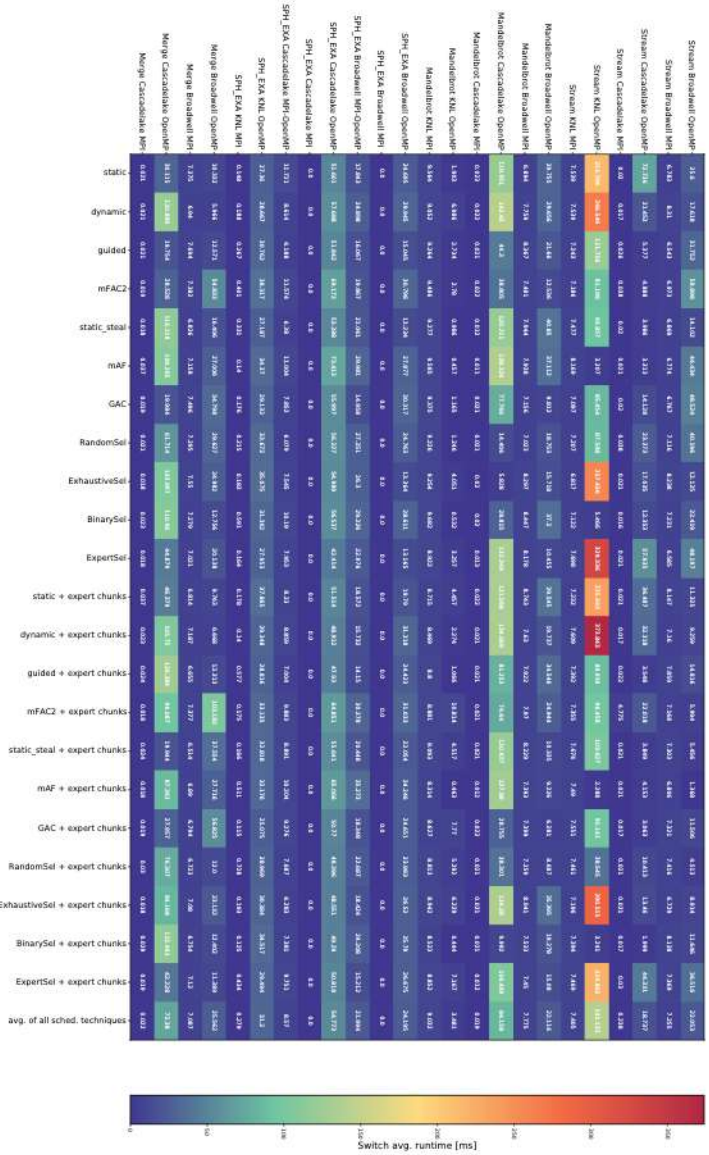


Figure 5.57: Average runtime for context switches events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

The average duration of a context switch (figure 5.57) is shorter with MPI than with OpenMP. The only exception to this is Mandelbrot on KNL. Similar to the wait time (figure 5.55) and delay (figure 5.56) SPH_EXA with MPI has no recorded context switches.

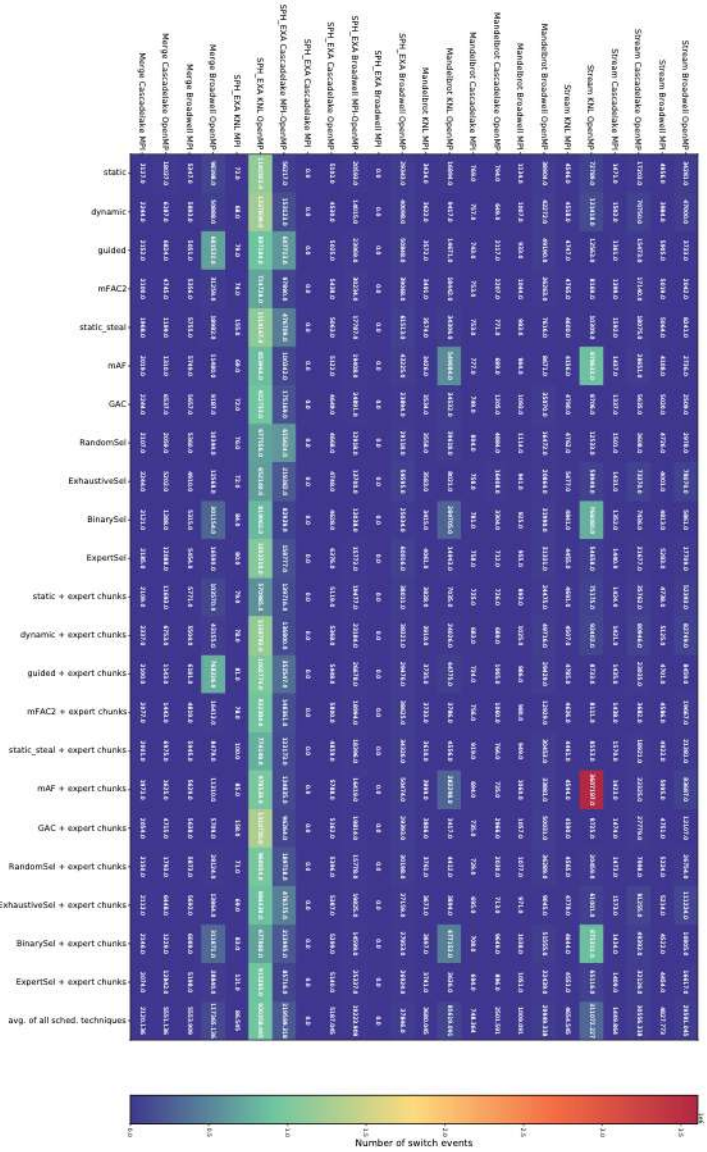


Figure 5.58: The number of context switch events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

For all applications and on all systems there are fewer context switches with MPI than with OpenMP (figure 5.58). For SPH_EXA with MPI on Broadwell and Cascadelake, there were no context switches recorded. An explanation for this is that the executions with MPI were a bit shorter. On Cascadelake the SPH_EXA hybrid version experienced more context switches than the MPI or OpenMP version.

5.2.3 Idle Time

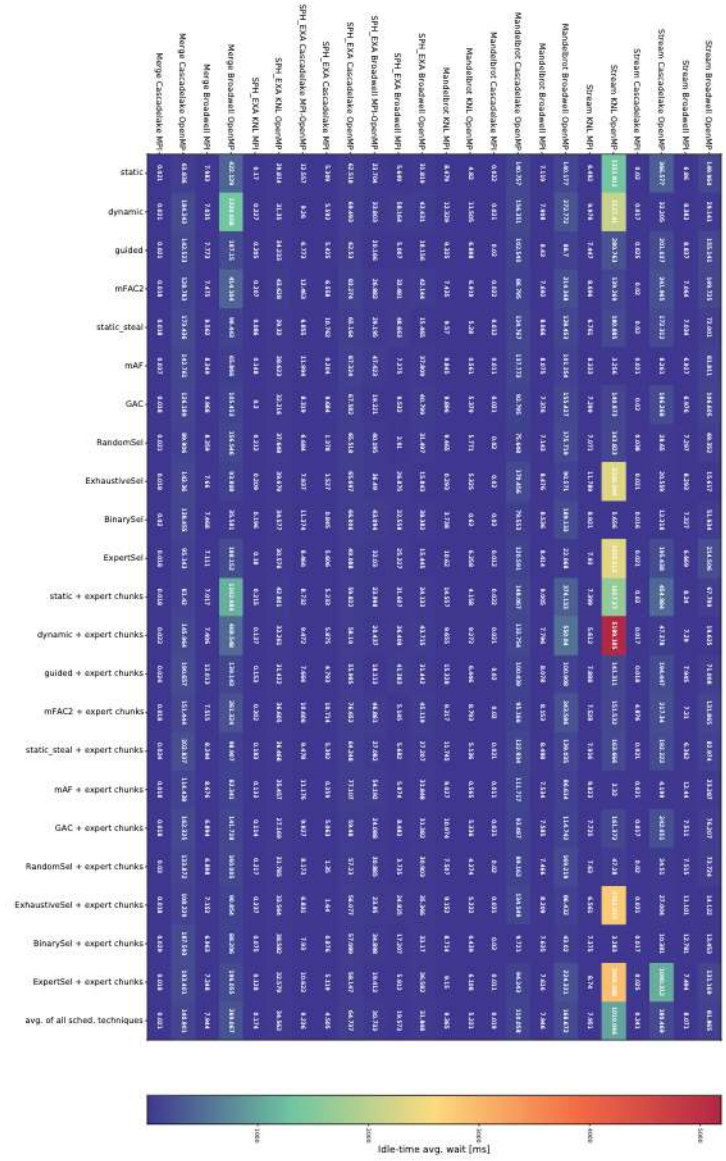


Figure 5.59: Average wait time for thread migration events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

The wait time of idle events (figure 5.59) is shorter for applications with MPI. The only exception to this is Mandelbrot on KNL. The hybrid version of SPH_EXA has values between the versions with OpenMP and MPI.

The delay of idle time events is always zero. No process needs execution time on the CPU. Therefore, the idle time does not wait until it is scheduled.

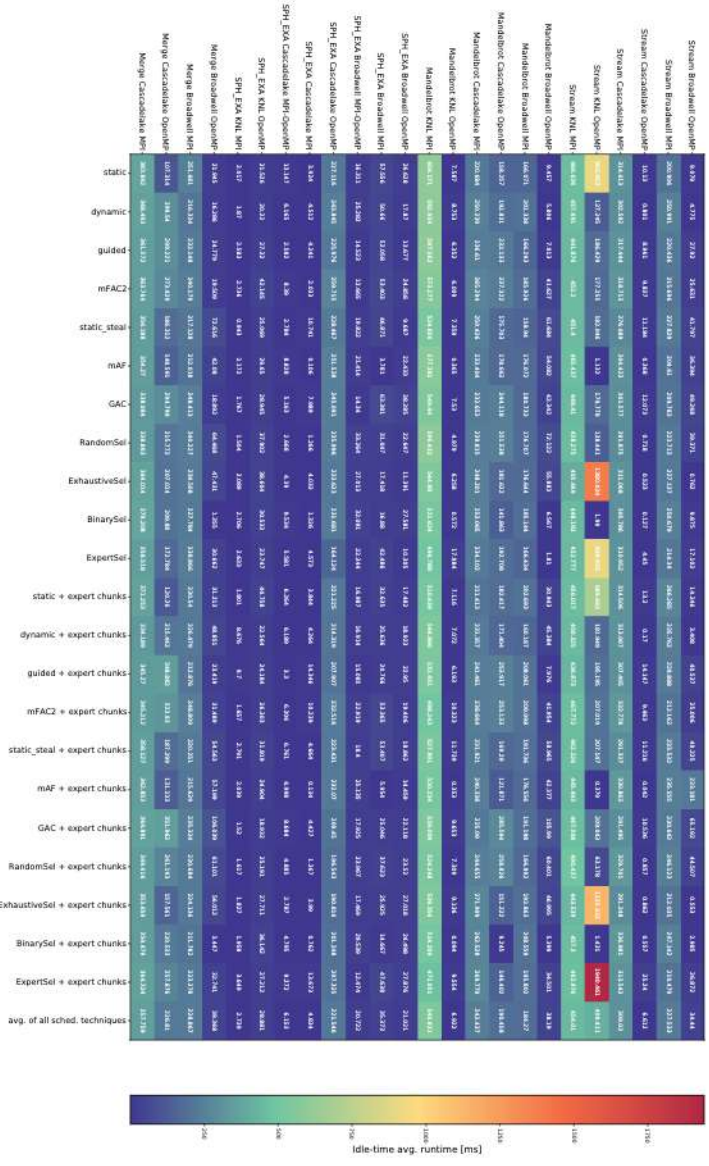


Figure 5.60: Average runtime for thread migration events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

The runtime of idle events (figure 5.60) is longer for executions with MPI, except for SPH_EXA on Cascadelake and KNL. The hybrid SPH_EXA version on Broadwell had shorter idle time events than with MPI or OpenMP.

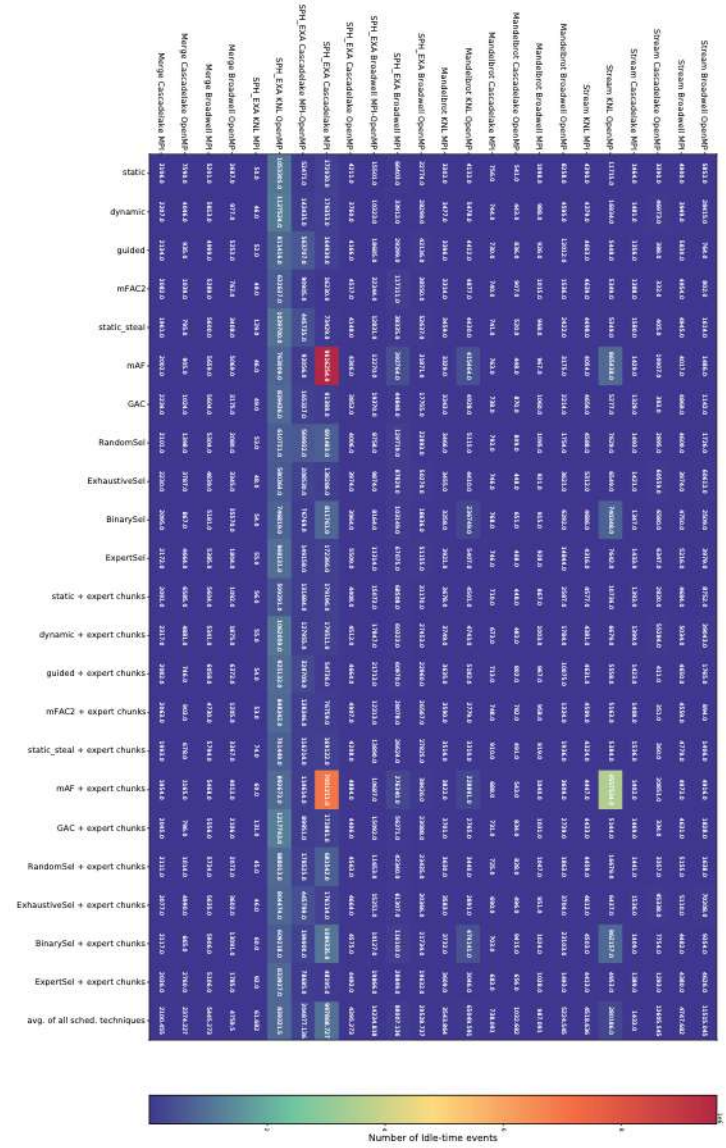


Figure 5.61: Number of thread migration events for the different applications on Broadwell, Cascadelake, and KNL with different thread scheduling techniques. Parallelized with MPI or OpenMP.

The highest number of idle time events is reported for all applications on KNL with OpenMP. Except for the two outliers for SPH_EXA with MPI on Cascadelake. The number of idle events and their duration suggests that with MPI there are fewer idle events, but they take longer. The hybrid SPH_EXA version on Broadwell has fewer idle time events. This events have also a shorter runtime (figure 5.60). So the processor on Broadwell has less idle time with the hybrid version of SPH_EXA than with the other versions. This is not the case on Cascadelake. There the hybrid version has values between those of MPI and OpenMP.

There is often no big difference between the execution with OpenMP and the hybrid MPI and OpenMP. A reason for this might be that there are only two MPI ranks and many OpenMP threads. We leave it for future research to investigate whether more MPI ranks change these measurements.

5.3 Discussion

In this section we summarize and discuss the results, that we presented in sections 5.1 and 5.2. We highlight some measurements that are distributed among the plots.

5.3.1 Application

We have four different applications with different properties. We presented the data in sections 5.1 and 5.2. Mandelbrot is compute-bound, Stream is memory-bound. Merge is a combination of Mandelbrot and Stream with compute- and memory-bound phases. The last application is SPH_EXA which is memory-bound on Broadwell and Cascadelake and compute-bound on KNL. The performance of the application depends on the computing system on which they are executed.

Also important is the execution time of the application. SPH_EXA has the longest execution time. Mandelbrot executed longer than Stream. The execution time of Merge is those of Mandelbrot and Stream added. If an application executes longer there will be more scheduling events during this time. Because the scheduler interrupts the application periodically to let other processes execute. So we expected that SPH_EXA has more context switches, thread migrations, and idle times than the other applications.

Mandelbrot has more thread migrations than Stream and SPH_EXA on Broadwell and Cascadelake (figure 5.5). Mandelbrot has a higher degree of load imbalance. This requires more load balancing by the scheduler. On KNL, SPH_EXA has much more thread migration events than the other applications and Stream slightly more than Mandelbrot (figure 5.4). The number of thread migrations for Merge is similar to those of Mandelbrot. But we expected, that this number is closer to the number of thread migration of Mandelbrot and Stream added up.

The number of context switches on Broadwell is lower for Mandelbrot than Stream. SPH_EXA has the more context switches then the other applications (figure 5.9-5.11). On Cascadelake, Mandelbrot has fewer context switches than the other applications. Stream and SPH_EXA have roughly the same amount of context switches. On KNL, Mandelbrot and Stream have roughly the same amount of context switches and SPH_EXA has more. The number of context switches depends more on the computing system and the thread configuration than on the application.

Mandelbrot has the least amount of idle time events on Broadwell and Cascadelake (figure 5.15-5.17), followed by Stream. SPH_EXA has the most idle time events. The difference between the applications can vary widely depending on the computing system. These idle events on Broadwell and Cascadelake for Mandelbrot take between 30 and 190ms (figure 5.14). For Stream, they have an average runtime between 10 and 60 and for SPH_EXA

between 10 and 220ms. So Mandelbrot has the least amount of idle time events, which on average are a bit longer than for Stream. SPH_EXA has the most idle time events which have a longer runtime. Because Mandelbrot is compute-bound this is what we expected. A compute-bound application, like Mandelbrot, should spend most of its time computing. The idle time of a system should be short. We observe that Stream has more idle time events. Memory-bound applications spend more time idle while waiting for data.

Cache request rate for the different cache levels and applications

Application \ Cache Request Rate	System	L1	L2	L3
Mandelbrot	Broadwell	0.0036	0.0009	0
	Cascadelake	-	0.008	0
	KNL	3.0365	0.0006	-
Stream	Broadwell	0.0093	0.7254	0.0302
	Cascadelake	-	0.4357	0.0172
	KNL	5.5316	0.0475	-
SPH_EXA	Broadwell	0.0589	0.0248	0.0014
	Cascadelake	-	0.0219	0.0008
	KNL	0.4459	0.0041	-

Table 5.2: These values are averages for all thread scheduling techniques

All applications have a much higher L1 request rate on KNL than on Broadwell. The application Stream has the highest cache request rate on all cache levels and systems. This is what we expected from a memory-bound application. On KNL Mandelbrot has more L1 cache accesses per instructions than SPH_EXA, but less on L2. On Broadwell, SPH_EXA has a higher request rate on all cache levels. On KNL SPH_EXA has fewer L1 accesses per instructions than Mandelbrot, but more on L2 and L3. This indicates that the data locality for SPH_EXA is not as good as for Mandelbrot, and therefore more data has to be loaded from higher cache levels.

Bandwidth between the cache levels and main memory [MBytes/s]

Application \ Bandwidth between	System	L1 - L2	L2 - L3	L3 - Memory
Mandelbrot	Broadwell	31.82	19.638	38
	Cascadelake	41.34	16.881	44
	KNL	21.74	-	193
Stream	Broadwell	3159.08	3022.052	60647
	Cascadelake	4945.95	7152.571	84143
	KNL	564.07	-	6350
SPH_EXA	Broadwell	2314.79	887.424	3180
	Cascadelake	4443.99	1215.075	3719
	KNL	103.43	-	914

Table 5.3: These values are averages for all thread scheduling techniques. Because there is not much difference between the thread scheduling techniques.

The memory-bound applications Stream and SPH_EXA have higher bandwidths than Mandelbrot. Interesting is that for Mandelbrot and SPH_EXA, the bandwidth between L2 and L3 cache is much lower than between L1 and L2 and between L3 and memory. This is not

the case for Stream. This shows that Stream reads and writes from all cache levels equally. Other applications use different cache levels differently.

5.3.2 Thread Level Scheduling

Some scheduling techniques struggle with Stream. Especially dynamic and ExhaustiveSel. For some metrics also static_steal and ExpertSel. Stream requests a lot of data. If the execution of Stream is divided into small intervals, then the data can not be preloaded. This happens with dynamic. Each thread does a small chunk of the multiplication and has then to wait for the data for the next data chunk. The L3 data volume (figure 5.31 and L2 data volume (figure 5.26) are much higher for dynamic and ExhaustiveSel. For most metrics, the expert chunks improve these bad-performing scheduling techniques.

For Mandelbrot and Stream different scheduling techniques have more thread migrations (figure 5.5). For Mandelbrot it is guided, mFAC2,GAC, and RandomSel. For Stream, it is static, dynamic, and ExhaustiveSel. The expert chunk does not make a big difference. Thread migration may show load imbalance. So these scheduling techniques may lead to more load imbalance than others. But this depends on the application.

Also the number of context switches is influenced by some scheduling techniques. For dynamic and ExhaustiveSel we measured more context switches than for the other scheduling techniques. The time between context switches (figure 5.6)is shorter for mAF and BinarySel for Mandelbrot and Stream. So these techniques experience more frequent context switches. Also the runtime of these switches is shorter than for other scheduling techniques.

In most measurements the scheduling techniques, that we analyzed, performed similar to each other. There is no scheduling technique that is less affected by the OS.

5.3.3 Computing Systems

We measured the executions of the applications on three different computing systems, Broadwell, Cascadelake, and KNL. In this section, we compare the performance of these three systems. For this, we take the average results over the different scheduling techniques because they perform similarly on all systems.

Over all applications, Cascade lake has the lowest number of thread migrations, followed by Broadwell, and KNL has much more. The average duration of a thread migration event is shortest on Broadwell (0.01ms) and takes a bit longer on Cascadelake (0.02ms). On KNL these events take much longer (0.13ms). The wait time of thread migrations is much lower on Cascadelake than on the other two systems, which have similar delays. So on Cascadelake, there are fewer thread migrations and they are less frequent. KNL has the highest number of thread migrations, which take more time. So there are quite some differences between the systems. The scheduling delay for migrations on Broadwell and Cascadelake are very similar, on KNL the delay is much higher.

Broadwell has on average the lowest number of context switches, Cascadelake has a bit more, and on KNL there are much more context switches. Also, the runtime of these context switches is lower on Broadwell. On Cascadelake and KNL they take roughly the same time. The frequency of these events is higher on Broadwell than on Cascadelake. It is much lower

on KNL. So the time between context switches on Broadwell is shorter compared to the other systems. The scheduling delay of context switches is much higher on Cascadelake, on average 7.29 ms. On KNL it is 0.26ms and on Broadwell 0.04ms. Broadwell nodes have fewer idle time events that are shorter than Cascadelake. KNL has much more idle time events which take even more time. Only the time between the idle events is similar on all systems.

5.3.4 Thread Configuration

We examined four thread configurations. The normal configuration is to use all available cores. We compare this to executions with one or two cores left idle. The last configuration is to leave one socket idle. The impact of the thread configuration depends on the used application and system.

On Broadwell, the fewest context switches are reported for executions with one idle socket (figure 5.9). On Cascadelake there are much more Switches with two idle cores than with other thread configurations (figure 5.10). The other configurations have similar amounts of context switches. On KNL there are fewer context switches when there are some idle cores (figure 5.11). This is similar to the number of thread migrations on KNL (figure 5.5). KNL has more context switches and thread migrations when all cores are used. So the OS interrupts the application less when there is at least one idle core. Cascadelake has also a bit fewer thread migrations with idle cores. But Broadwell it depends on the application which thread configuration leads to fewer migrations. The different systems behave differently for different thread configurations regarding the number of switches. Some can use this idle cores. This leads to fewer interrupts for the application.

The number of thread migration is not clearly influenced by the thread configuration (figure 5.5).Mandelbrot Cascadelake and KNL have in most cases a bit fewer thread migrations with idle cores. On Broadwell there are more thread migrations with one or two idle cores.

For the cache misses on all levels (figures 5.19,5.22, and 5.28) we do not see much difference between the thread configurations. On Broadwell and Cascadelake each core has its own L1 cache. So the available L1 cache per used core is the same. But several cores belong to the same cache group in L2 and L3. Therefore, we expected to see a lower miss rate.

Stream with one idle sockets loads more data (figure 5.34) because the available L3 cache is much smaller than when the second socket is used too. More data is loaded and evicted (figure 5.35) This difference between the thread configurations is higher on Cascadelake than on Broadwell. For the average idle runtime (figure 5.14) for SPH_EXA on Broadwell with one idle socket. We see that these idle events took much longer. So with one idle socket, there are fewer idle events that took on average much longer, compared to more utilized cores. This shows that the unused cores on one socket were indeed idle for most of the time.

The datavolume on L2 and L3 is higher for executions on one socket (figure 5.26, 5.31). At least for the memory intensive applications Stream and SPH_EXA. This can be

explained by the smaller available cache on one socket. The applications do the same work but have only half of the cache available. The other thread configurations do not show a big difference.

5.3.5 Limitations

With perf, we recorded thread migration, context switches, and idle time events. These events take a very short time, with some exceptions that we see in the plots in this chapter. Perf measures the wait time, scheduler delay, and runtime for these events in microseconds. A lot of these measurements are reported as 0.000ms. So the resolution of perf for this events is not always enough.

Migrations may depend on what is executed before. But we did not analyze thread migrations depending on what was executed, because we only recorded around 10-600 migrations for most configurations. With *perf latency* we see a list of the executed processes. Normally there are around 120 different processes that perf recorded. Most of these processes are kworker, also different kernel threads. It would be interesting to see if the number of migrations of the same Kernel threads increases with longer measurements. With enough recorded events it would be possible to analyze thread migrations depending on what kind of thread was migrated. The size of the perf.data files would probably increase significantly for this measurements.

Hardware counters are hard to interpret what they count. Not all systems use the same names for the same counters. Or they do not have these counters implemented. Perf uses these event aliases that map to predefined counter events and masks. PAPI and Likwid have commands to print the available events and metrics on a given system. The problem is that the names can change from system to system. It is not easy to verify that the same command measures the exact same metric. The hardware counters are defined by the manufacturer. But there is not a unified standard for these counters. So the available counters on our systems differ, even when they were produced by Intel. For example, on KNL much fewer hardware counters are available than on Broadwell.

6 Conclusion

To investigate the OS influence on the performance of parallel applications, we conducted many measurements on different systems. We compared applications with different properties with several thread-level scheduling techniques and expert chunks. We also compared the influence on different parallelization methods. With the tools PAPI, perf, and Likwid we measured scheduling events and memory and cache performance.

Our measurements show that the performance of an application differs between the different computing systems and thread configurations. The number and duration of thread migrations are different on the computing system. On KNL there are more thread migrations which on average take longer, compared to executions on Broadwell and Cascadelake. The thread configurations do not impact the number of cache misses. But on some systems, it leads to fewer context switches and thread migrations. The thread level scheduling techniques perform similarly to each other. But some scheduling techniques show on certain systems or with memory-bound applications worse performance than other scheduling techniques. Thread migrations and context switches take longer and are more numerous with applications that are parallelized with OpenMP than with MPI.

All aspects that we investigated are important. We can not point out a single factor that is in every case the most important one. Every aspect can influence the performance of the application. But for many observations, there is an example that shows the contrary.

6.1 Future Work

With perf, we recorded the overhead different events. But perf has no API to measure the important part of an application like it is possible with PAPI or the Likwid marker API. A tool like the perf-API [18] could deliver interesting measurements about the scheduling events during the execution of certain kernels.

We leave it to future work to investigate the memory and cache performance for the different parallelization techniques. We investigated the influence of the OS scheduler on applications that are parallelized with MPI, OpenMP, and hybrid MPI and OpenMP. For this kind of measurement, the tool *likwid-mpirun* should be suitable.

Why perf reports zero values for some thread configurations and not for others, needs

to be explored in future work.

Bibliography

[1] 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.

[2] 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.

[3] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018. Chapter, Paging: Faster Translations (TLBs).

[4] Emiliano Betti, Marco Cesati, Roberto Gioiosa, and Francesco Piermaria. A global operating system for hpc clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[5] MOpenMP Architecture Review Board. Openmp. https://www.openmp.org/, February 2022.

[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] Tommaso Cucinotta, Giuseppe Lipari, and Lutz Schubert. Operating system and scheduling for future multicore and many-core platforms. *Programming Multicore and Many-core Computing Systems*, 86, 2017.

[8] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. doi: 10.1109/99.660313.

[9] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. Using papi for hardware performance monitoring on linux systems. In *Conference on Linux Clusters: The HPC Revolution*, volume 5. Citeseer, 2001.

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

[11] Urs Fässler and Andrzej Nowak. perf file format. Technical report, Technical report, CERN Openlab, 2011.

[12] MPI Forum. Mpi forum. https://www.mpi-forum.org, February 2022.

[13] Balazs Gerofi, Yutaka Ishikawa, Rolf Riesen, and Robert W Wisniewski. *Operating Systems for Supercomputers and High Performance Computing*, volume 1. Springer, 2019.

[14] Roberto Gioiosa, Fabrizio Petrini, Kei Davis, and Fabien Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *Proceedings of the Fourth IEEE International Symposium on Signal Processing and Information Technology, 2004.*, pages 387–390. IEEE, 2004.

[15] 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.

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

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

[18] Alex Gustafsson. perf api. https://github.com/AlexGustafsson/perf, February 2022.

[19] M Tim Jones. Inside the linux scheduler. *IBM Developer Works*, 2006.

[20] Terry Jones. Linux kernel co-scheduling for bulk synchronous parallel applications. In *Proceedings of the 1st international workshop on runtime and operating systems for supercomputers*, pages 57–64, 2011.

[21] 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.

[22] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Lb4omp: A dynamic load balancing library for multithreaded applications, 2021.

[23] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Auto4omp. https://github.com/unibas-dmi-hpc/LB4OMP/tree/dev, February 2022.

[24] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. Lb4omp. https://github.com/unibas-dmi-hpc/LB4OMP, February 2022.

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

[26] Tang Peiyi and Yew Pen-Chung. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In *Proceedings of the International Conference on Parallel Processing*, pages 528–535, August 1986.

[27] 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.

[28] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987. ISSN 0018-9340.

[29] Sameer Shende. Profiling and tracing in linux. In *Proceedings of the Extreme Linux Workshop*, volume 2. Citeseer, 1999.

[30] Aman Singh, Anup Buchke, and Yann-Hang Lee. A study of performance monitoring unit, perf and perf_events subsystem, 2012.

[31] Top 500 Supercomputer sites. Top500. https://www.top500.org, February 2022.

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

[33] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[34] CORPORATE The MPI Forum. Mpi: A message passing interface. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 878–883, 1993.

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

[36] 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.

[37] Dan Tsafrir, Yoav Etsion, Dror G Feitelson, and Scott Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, 2005.

[38] Inovative Computing Laboratory University of Tennessee. Publications. https://www.icl.utk.edu/view/biblio/project/papi?items_per_page=All, February 2022.

[39] Unknown. likwid. https://github.com/RRZE-HPC/likwid, February 2022.

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

[41] 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

A.1 Measurements on MiniHPC

PAPI is the tool that made the least problems. Just load it with *ml PAPI* and follow the tutorials. It is the tool that requires the most effort for different measurements but it worked fine.

Perf requires sudo rights for most of the measurements. We did this with *alias sudo='sudo LD_LIBRARY_PATH="$LD_LIBRARY_PATH"'* and then we could execute the perf command with *sudo -E*. For example *sudo -E /usr/bin/perf sched record...* The best tutorial for perf We found is written by Brendan Gregg [17], there nearly everything about perf is explained with examples how to use it.

We performed the experiments on miniHPC and also analyzed the perf.data files on miniHPC. The reason for this is that these files can easily grow over 100MB. We did not need all of this data. Because creating the plots on miniHPC was cumbersome, we transferred the necessary and compressed data to our local machine with git. The only constraint is that the git repository can only reach 4GB. And deleting it from the history is not easy.

Likwid is easy to install on any Linux machine to test it out. On miniHPC we had some problems. *Likwid-mpirun* did not work at all. For the other measurements on miniHPC one problem was, that if fewer were threads pinned with Likwid than are available for OpenMP, The measurement with Likwid does not work correctly. For this we used *OMP_NUM_THREADS* to limit the number of threads for OpenMP.

A.2 SPH_EXA Kernel Analysis

With PAPI and Likwid marker API it is possible to analyze the individual kernels of an application. We hope, that this will someday work with perf too. During our work, We made some measurements of the SPH_EXA kernels. On the x-axis, there are the SPH_EXA kernels. On the y-axis is the measurement. So that the plots are visible, there are some cuts in the y-axis. All measurements were repeated 10 times and performed on Broadwell.

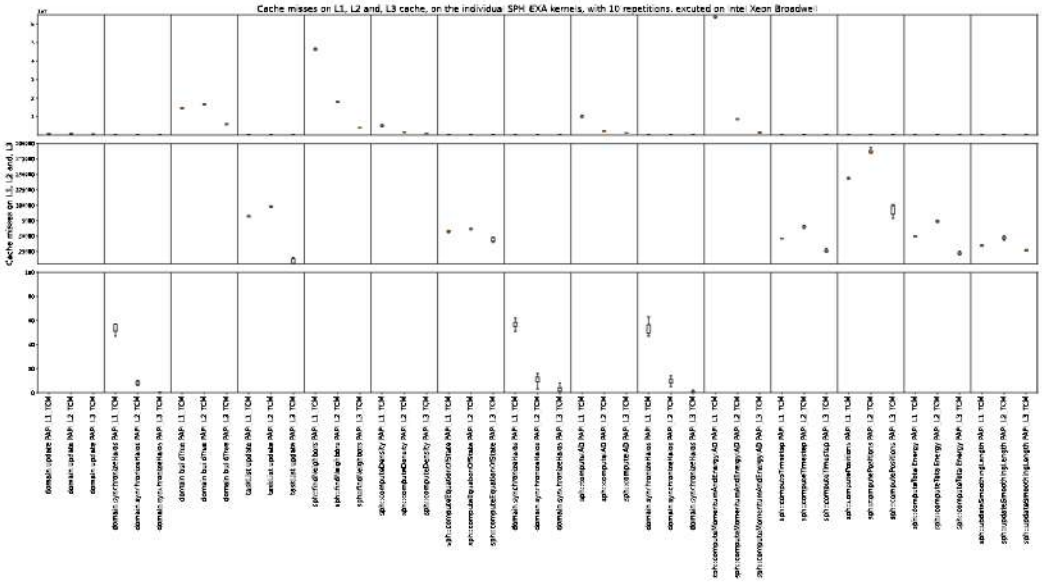


Figure A.1: Comparison of the cache misses of the individual SPH_EXA kernels.

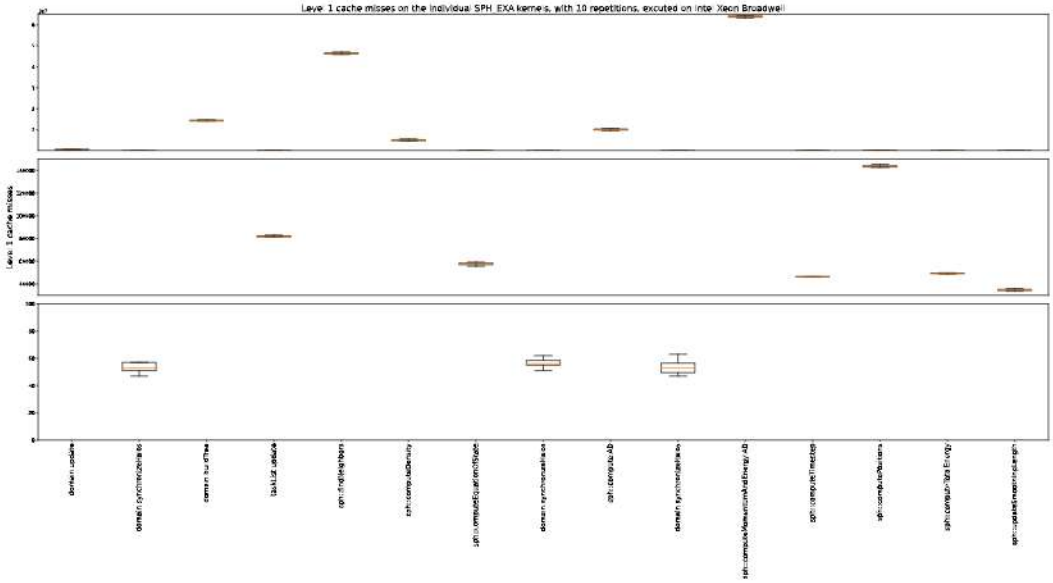


Figure A.2: L1 cache misses of the individual SPH_EXA kernels.

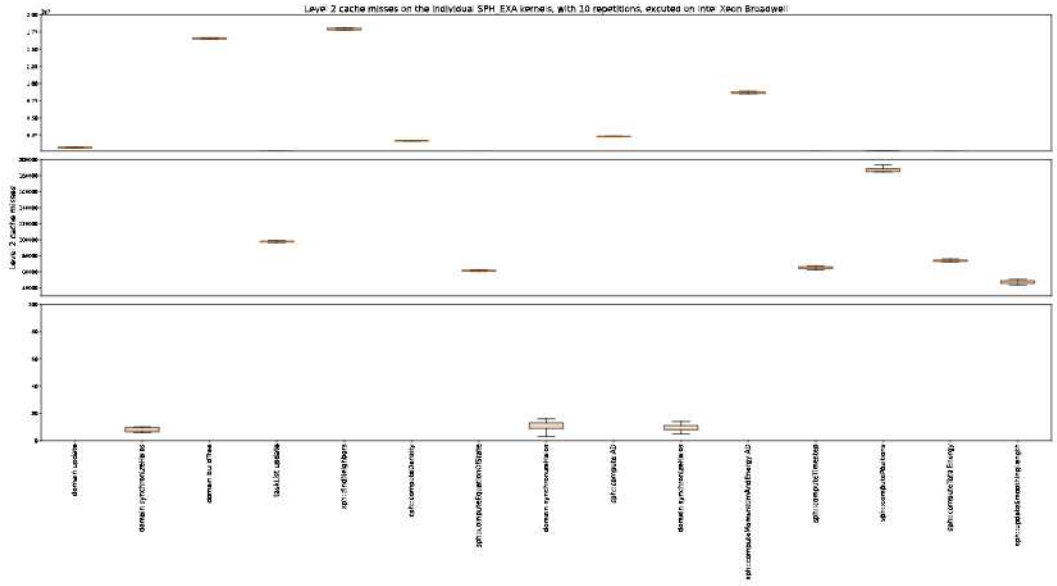


Figure A.3: L2 cache misses of the individual SPH_EXA kernels.

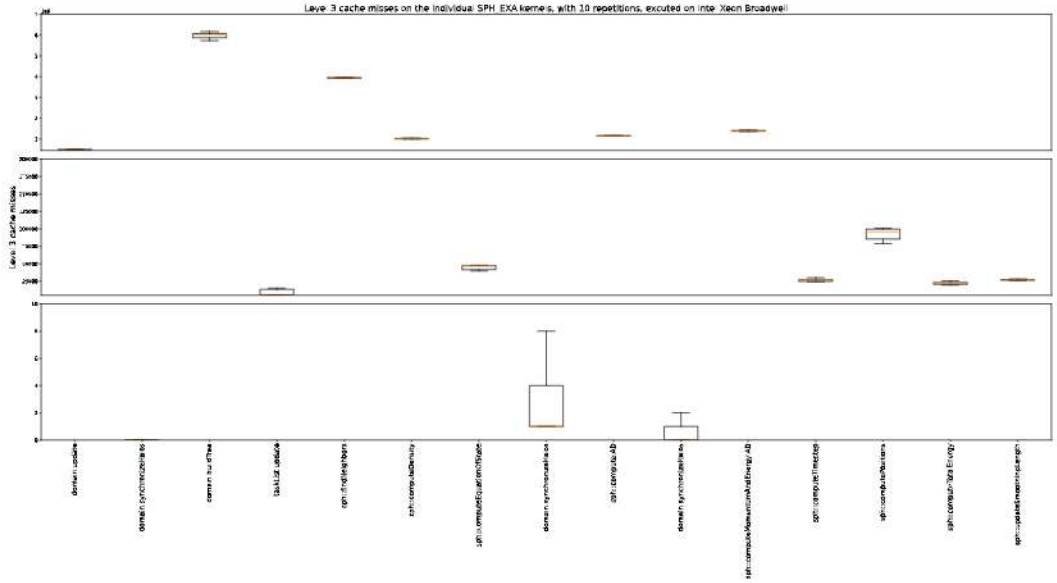


Figure A.4: L3 cache misses of the individual SPH_EXA kernels.

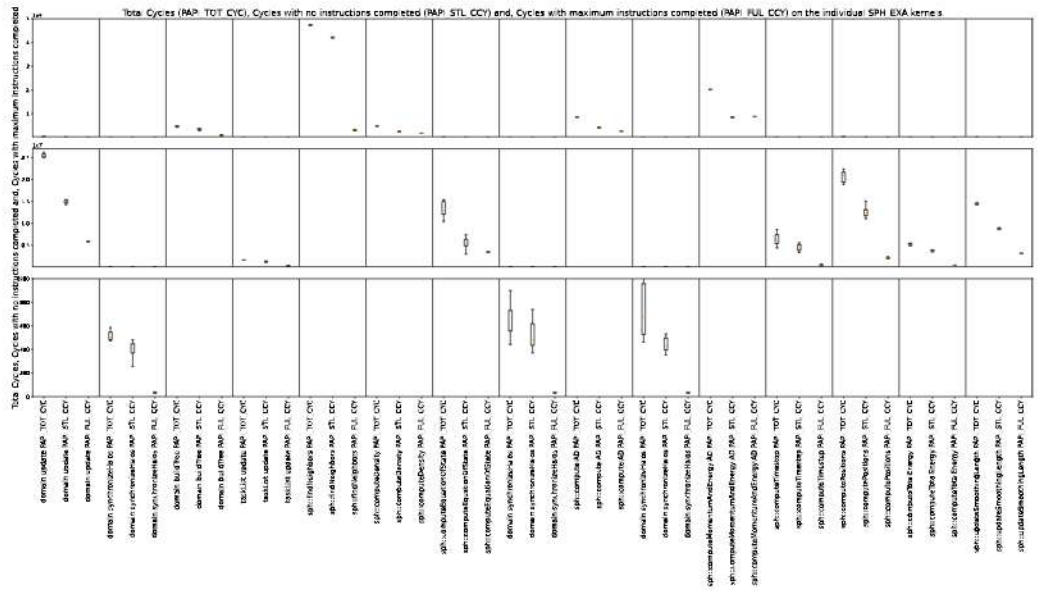


Figure A.5: Comparison of the CPU cycles of the individual SPH_EXA kernels.

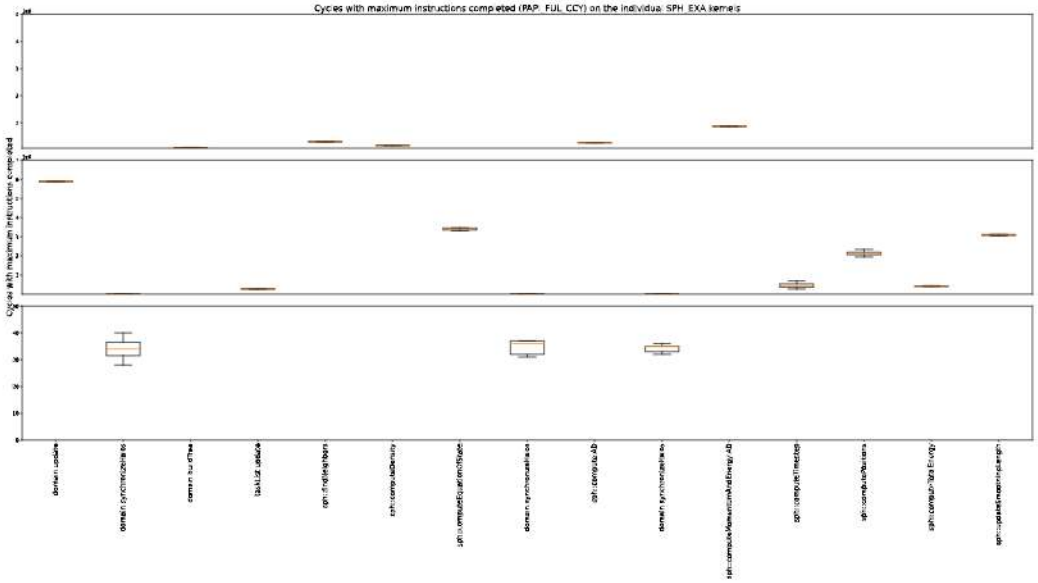


Figure A.6: Comparison of the CPU cycles with maximum instructions completed of the individual SPH_EXA kernels.

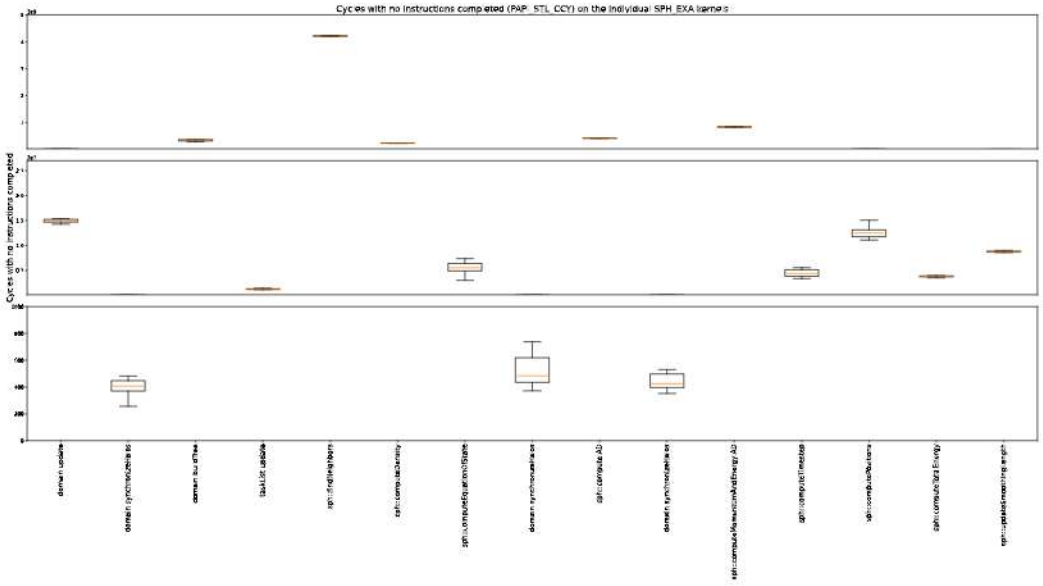


Figure A.7: Comparison of the CPU cycles with no instructions completed of the individual SPH-EXA kernels.

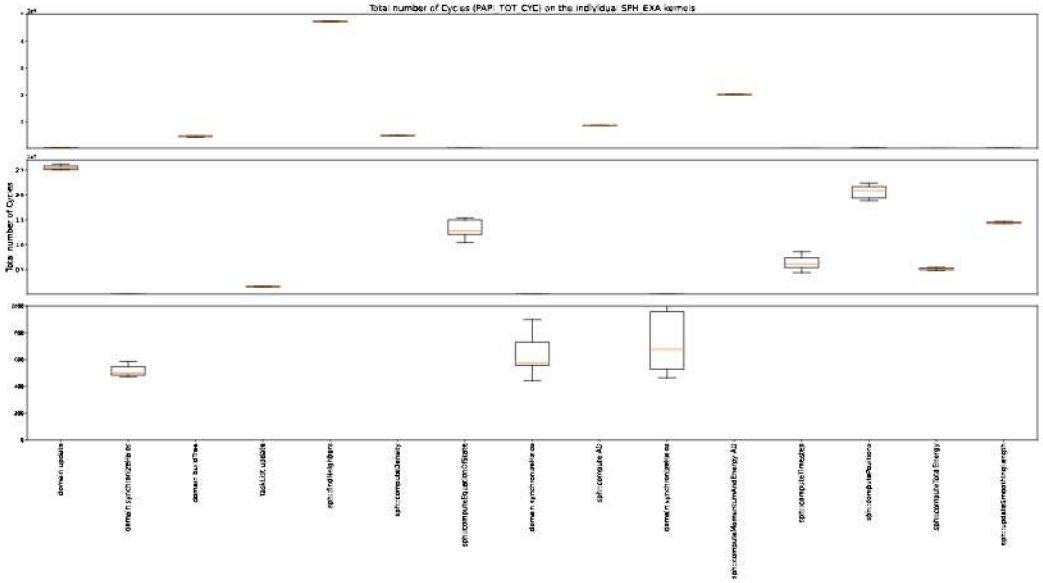


Figure A.8: Comparison of the total amount of CPU cycles of the individual SPH-EXA kernels.