CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Implementing online performance modelling in OpenMP

Adding an energy aware scheduler for increased energy efficiency

Master's thesis in Computer science and engineering

Thomas Alexandersson
Olle Nilsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Implementing online performance modelling in OpenMP

Adding an energy aware scheduler for increased energy efficiency

Thomas Alexandersson Olle Nilsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Implementing online performance modelling in OpenMP
Adding an energy aware scheduler for increased energy efficiency

THOMAS ALEXANDERSSON
OLLE NILSSON

Supervisor: Miquel Pericàs, Department of Computer Science and Engineering
Examiner: Risat Pathan, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Implementing online performance modelling in OpenMP
Adding an energy aware scheduler for increased energy efficiency
THOMAS ALEXANDERSSON
OLLE NILSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

OpenMP is the staple and foundation of many HPC applications. With the recent increase in heterogeneous computing platforms, it lacks the ability to utilise the architecture to its fullest potential. This new hardware allows for more advanced techniques than dynamic voltage and frequency scaling (DVFS) when minimising energy consumption. In this project, we have implemented ERASE (EneRgy Aware SchedulEr) into the LLVM's OpenMP run-time. Mainly intended for heterogeneous compute platforms, it schedules the *task* and *taskloop* pragmas, with the goal of their execution being to use the least amount of energy. This is achieved by a mix of online and offline classification of tasks. By storing previous execution times, it is adaptable to an ever-changing computing environment. In addition, we also evaluated different interpolation models to predict missing performance data and untested system configurations. Compared to the default OpenMP, we show that ERASE can reduce energy consumption by up to 17% in realistic scenarios using a heterogeneous compute platform. Adding interpolation can further decrease the energy consumption by 25% depending on task granularity and compute platform.

Keywords: energy efficient computing, online performance modeling, OpenMP, heterogeneous architectures.

Acknowledgements

This thesis would not have been possible without our excellent supervisor Miquel Pericàs which guided us through this journey. From the early days of writing the proposal to the final product. He took us under his wing and and provided support whenever needed.

We would also like to thank his research team, including but not limited to Jing Chen, which previous work this thesis rest upon. Without their foundational work this thesis would have been impossible. Their knowledge and assistance have been invaluable throughout the whole project.

Thomas Alexandersson, Gothenburg, June 2022
Olle Nilsson, Gothenburg, June 2022

Contents

1
Introduction

Modern parallel applications that utilise parallel run-time systems split their workload into smaller pieces called tasks. These tasks may vary in granularity, characteristics, and time complexity. Having a good understanding of these variables is the key to success when trying to model their performance. An accurate performance model is necessary to be able to schedule tasks in terms of both high-performance and energy efficiency.

Heterogeneous shared memory environments are systems that contain more than one type of CPU architecture using the same or different instruction set architecture (ISA). Cores of the same architecture are usually grouped in the cache hierarchy and referred to as a cluster. Performance on these systems may vary wildly depending on which cluster tasks are executed on, both in terms of absolute performance and the amount of energy consumed. Executing less demanding tasks on low power clusters allows more computationally demanding tasks to execute faster where the power is needed given that the power of the system is limited. This in turn increases the energy efficiency of the entire system. Incorporating these ideas into a well-used run-time system like OpenMP would allow for more efficient computation of a wide selection of applications without any changes to existing code bases, allowing them to make use of modern hardware advancements.

OpenMP is an API specification for parallel programming. It is meant to be used for easy parallelisation of existing code with minimal work by the programmer. The programmer specifies how a section shall be parallelised and then the API carries it out in the background using the available resources of the system.

Several approaches exist in trying to perform accurate performance modelling. Some, like PMNF (performance model normal form), focus on offline modelling where functions are derived to estimate the performance[1]. This can provide accurate predictions as long as the computing environment stays the same. However, it may have problems adapting to unforeseen events during execution. Other approaches focus on reactive online modelling with no assumed baseline or interpolating in between missing data points[2, 3]. This requires the task to first be executed to get an estimate, and thus, these solutions have difficulties predicting the performance of a previously unseen task or configuration. Other projects exist to allow programmers to specify the complexity of an algorithm to the run-time environment[4]. This will require more time spent developing programs and might also need existing programs to be rewritten to reap the benefit.

The problem arises with the dimensions of the configuration space available to the performance model. Modern computing platforms are often heterogeneous with multiple cores of different architectures. Each architecture might be better suited

for a different type of task, where a task may also be split to be computed by a subset of the available cores. The scheduler might even dynamically adjust the voltage and frequency of different clusters of cores or individual cores to achieve maximum efficiency[5]. The frequency of cores or clusters can also be influenced by external factors which are uncontrollable and unknown to the run-time.
Tasks themselves have different characteristics that complicate the performance modelling further. They may not scale linearly with the input. Tasks may be memory or compute-bound which implies they would be better suited to be executed on a particular core. To enhance the difficulty, a task may transfer from being compute-bound to memory-bound or vice versa during execution. Optimally the tasks would thus be migrated by their new characteristics.
This multi-dimensional problem makes learning approaches time-consuming and inefficient. Instead, quick accurate performance estimates are needed. These need to dynamically adapt to the hardware and current state of the system to simultaneously achieve high-performance and energy efficiency. Quickness is a key component since overhead must be kept low, especially regarding fine-grained tasks. Otherwise, the overhead of the run-time would increase the impact of the execution time negatively.
ERASE, short for EneRgy Aware SchedulEr, is an energy-efficient task scheduler[2]. It combines performance modelling, power profiling, and a scheduler to decide the optimal type and number of cores to run each specific task on. It consists of a couple of components, such as the performance model module, which tries to predict the execution time of a given task based on the history of previous executions. The power profiler module tries to estimate the energy consumed during the execution of a task based on system, and task, parameters. It also uses a core activity module to track which cores are active and which that are sleeping. The scheduler uses the information gathered from these modules to schedule tasks in the most energy-efficient execution places.
The tasks are classified offline based on their arithmetic intensity into one of three categories that are suitable for different configurations. The result of subsequent runs is stored and then used in performance modelling for future runs. Utilising this, ERASE outperforms many state-of-the-art solutions in terms of energy efficiency. However, it is not without limitations. ERASE works best on fine-grained heavily repeated tasks. And due to its reactive nature, it must test every configuration of cores and available architecture once to get a baseline in an exhaustive search before finding the optimal configuration. The performance model thus lacks any ability to predict untested configurations. It is also currently based on an experimental run-time system.

1.1 Goal

The primary goal of this thesis will be to implement ERASE into an OpenMP run-time framework to increase its energy efficiency. Furthermore, this thesis will explore the possibility of improving the current online performance model by answering a few sub-questions. As previously mentioned, the current performance model used in ERASE does not make predictions on system configurations it has not seen before. Depending on the system, there may be several different computing architectures,

each with its number of cores and core clusters. Because of this, there may be a large number of possible configurations for a given task to run on. Running an exhaustive search to figure out how the property of each cluster may be unfeasible. Especially given that each configuration will run differently depending on task properties.
Consider that we have a few sample configurations with some measured execution time out of each architecture. With this in mind, we would like to answer the following questions.

1. *Is it possible to implement ERASE and feasible to use ERASE as a scheduler in OpenMP?*
2. *Is it possible to effectively predict the performance of untested system configurations with a given task?*
3. *What accuracy is required to make a prediction model effective?*
4. *How many sample configurations are necessary to deem the model to be effective?*

1.2 Scope

The problem can be defined as there currently does not exist a quick efficient online performance modelling that is also capable of accurately predicting a previously untested configuration or unseen task. With that in mind, the scope of the project will be limited by the following statements.

1. *Only two pragmas in OpenMP will be considered, namely the task and taskloop pragmas*
2. *The scheduler will not actively change the frequency and voltage of cores or clusters for optimization but will react to external changes affecting them.*
3. *We limit testing and implementation to a single heterogeneous computing platform.*

The reason for these limitations is simply due to time. Given the time of the project, it is unfeasible to include all pragmas in OpenMP. All pragmas may also not be suitable to schedule using ERASE.

2
Background

This chapter will describe relevant concepts and previous work beginning with an explanation of performance modelling. Afterwards, a brief overview of heterogeneous computing platforms is presented along with some examples. Then an explanation of arithmetic intensity is presented, followed by an overview of ERASE with a breakdown of its modules. And finally explaining what OpenMP is, and how it works, accompanied by its implementation in the LLVM project.

2.1 Performance modeling

Performance modelling is the method of using various data and tools to try and estimate the performance of a program. This usually refers to estimating the execution time, but other metrics such as latency or energy may also be of interest. It is a complex subject due to the many parameters of the problem. Such as programs may have different complexity based on their input and perform differently on different hardware. Furthermore, a great deal of information that impact the performance might not be known without running the program. Such as cache misses for various cache levels, instruction-level parallelism etc. Several methods exist to tackle these issues, each with its drawbacks and advantages, which will be covered in section 2.1.1.
One important distinction between online and offline performance modelling is that one is where information is gathered during the execution of the program, where future executions utilise the new information. The other is done offline, by determining the behaviour beforehand to use during run-time. A performance model typically utilises offline information in combination with information gathered online. In this thesis, both of these techniques will be looked at.

2.1.1 Different approaches

Several different approaches exist which either uses online or offline performance modelling. These can generally be classified into different techniques.
A history-based approach is when the run-time remembers the previous execution time of a program. This time is saved and used to infer the next execution time. This is the approach ERASE utilises. To increase efficiency, programs with similar characteristics can be grouped and used to predict each other's execution times. This requires the programs to be classified into different groups, either offline or online. One advantage of this approach is its simplicity. It usually has low overhead

and does not require much computation. There is also a drawback. With programs of different time complexity and varying input, there is no guarantee that the next invocation of a program behaves similarly enough that the performance modelling is accurate enough.
Another approach is what can be called user-defined modelling. This would involve the programmer specifying the complexity of a program to the run-time, which would allow the run-time to make a well-informed guess of the execution time. The obvious drawback is that the programmer might not know the complexity of the program or estimate it incorrectly. Another drawback is that this will require refactoring of large existing codebases and would not be a "drop-in solution".
Other offline solutions exist, where one method introduces PMNF[1]. It was originally designed for larger MPI (Message Passing Interface) systems but applies to more fine-grained run-time systems as well. It utilizes the fact that function classes at the base of most programs usually are small enough to be iterated through. Using a mathematical model and various optimizations produces the best fit in terms of the scalability of each function call. The formula used to describe this behaviour can be seen in equation 2.1.

$$f(p) = \sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot \log_2^{j_k}(p) \tag{2.1}$$

In this context c, i and j are constants of which there are n numbers each. From this, it can extrapolate the execution time for the call and in turn the application. Like any offline model, it will be unable to adapt to changing circumstances during execution. One such event is strong cache scaling where threads benefit because the data needed is already in some level of the cache.
Other methods involve machine learning models that take application statistics, loops, branches, model performance, etc[6]. The inference overhead of these solutions is usually way too high to be used during online performance modelling and should thus be considered an offline model.

2.2 Heterogeneous compute platforms

Heterogeneous compute platforms refer to systems which are comprised of more than one type of processor unit or core. The most well known heterogeneous platform is a standard desktop computer which typically has both a CPU and a GPU of which the latter is either integrated into the hardware of the CPU or located on a separate die. Another heterogeneous system setup is where the CPU has a set of different types of cores. Such systems are typically comprised of high-performance cores together with in comparison, slower, but more energy-efficient cores. This allows applications with strict deadlines or heavy workloads to execute on high-performance cores while less demanding applications can be executed on low-power cores. A problem with homogeneous CPU designs, which is when all the cores are of the same type, is that they are not highly adaptable. For example, if they are high-performing cores, they suffer from high power usage which directly pushes the heat produced by the chip. This in turn hinders the number of cores you can have on a single die, because of

high temperatures. Instead, a heterogeneous CPU design allows the system to be high performing when needed with more flexible power consumption.
The number of systems utilising heterogeneous CPU designs has grown over the last decade. ARM's big.LITTLE architecture is the most known example which has had great success in the last decade[7]. For example, the top five most sold phones in 2021 have chiplets that are heterogeneous[8]. Another is a benchmark suite which ranks the performance of phone chiplets, showing that the 10 top as of April 21, are all based on heterogenous designs[9]. Where they either contain two or three different cores architectures.
Primarily implemented on smartphones and tablets where the power is limited but they are also present on embedded systems with low power consumption. Recently Apple released their M1 system on chip, with a CPU design that is based on ARM's big.LITTLE architecture. This chip, however, is mainly designed for desktop computers and is showing an impressive performance by pairing four high-performance cores with four energy-efficient ones[10]. Another is Intel with their latest generation of processors called Alder Lake, have leapt over to using a heterogeneous design. Pairing up to 8 performance cores with up to 8 energy-efficient ones on a single die [11]. The big difference from ARM, which uses its own ISA, is that Intel bases their processors on the x86 ISA. With both Apple's M1 and Intel's Alder Lake currently being among the top CPU designs in terms of performance, the future of heterogeneous designs is looking good.

2.3 Arithmetic intensity and roofline model

Arithmetic intensity (AI), sometimes referred to as computational density, is a metric used to determine the characteristics of a given program on a specific system. This can be used to determine what type of optimization of the program might be required [12, 13]. AI consists of two parts. First, the work, W is a measure of the computation performed by the kernel. The actual measure varies depending on the kernel but can be integer operations, array point modified etc. But by far the most common metric used is the amount of floating-point operations (FLOPS). Secondly, the memory traffic Q is the amount of memory traffic induced by the kernel, this may be bytes written to main memory, cache accesses from the last level cache or a combination of the two. The AI, I is then computed as seen in equation 2.2.

$$I = \frac{W}{Q} \tag{2.2}$$

Thus, I denotes the ratio between the work and memory traffic. Noteworthy is that W is highly dependent on the program in question. Whereas Q is much more platform bound and varies depending on the memory subsystem and the cache hierarchy.
The roofline model is a simple model for analyzing bottlenecks and bounds for a kernel on a specific system [14]. It is represented by a 2D plot, where the Y-axis is performance and the X-axis is AI. Performance is usually measured in gigaFLOPS (GFLOPS). The roofline model plot is created by the function in equation 2.3.

$$P = min \begin{cases} \pi \\ AI * \beta \end{cases} \tag{2.3}$$

P is the maximum attainable performance, π is the peak performance of the system and β is the system's maximum DRAM bandwidth. This will create two straight lines with an intersection point known as the ridge point. This point is valuable for determining the current bottleneck of the kernel on the system. An example roofline plot can be seen in figure 2.1.

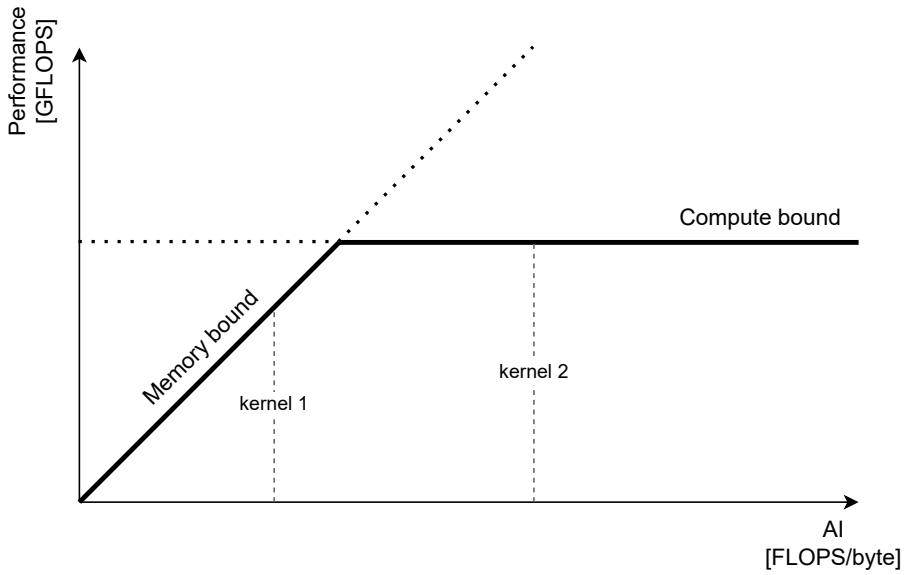

Figure 2.1: A roofline model with two hypothetical kernels where kernel 1 is memory bound and kernel 2 is compute-bound.

The AI of a specific kernel helps determine its bottleneck. If the $AI < \pi/\beta$, then the kernel is said to be memory-bound and is limited by the memory transferring capabilities of the system. Likewise, if $AI > \pi/\beta$, then the kernel is said to be compute-bound and is limited by the system's overall computing performance.

2.4 ERASE

As mentioned in the Introduction, ERASE is an online scheduler that schedules for maximum energy efficiency. It is an online work-stealing scheduler designed to be used on heterogeneous computing platforms. It leverages the fact that the different architectures are most energy-efficient at different types of kernels. This allows ERASE to perform the same computations using both less energy, and with improved performance compared to other solutions. The different tasks of the kernel are executed offline on the platform to measure their AI. They are then categorized into 3 different types of tasks depending on their AI. This is done using equation 2.4

$$AI = \frac{number\ of\ cycles * FLOPs/cycle}{number\ of\ cache\ misses * 64\ bytes} \tag{2.4}$$

which takes both memory access and work performed into consideration. Generally, the cache which to count misses from can vary depending on the needs. In the context of this project, however, the cache misses will refer to the last level cache misses. The three types are memory-bound, cache intensive and compute-bound. The scheduler also uses what is called task moldability. This means that a task, if possible, can be split for execution on different numbers of cores if this increases the energy efficiency of the application. In practice, this means when a task is scheduled it can be scheduled on 1, 2, 4,..., n cores by the scheduler's decision. This is called the resource width. Deciding this during run-time enables dynamic behaviour and the ability to adapt to a changing computing environment. The last key to energy efficiency is exponential back-off sleep. This states that when a thread unsuccessfully has tried to steal a task from all other threads it will go to sleep for a period of time before trying to steal again. After waking up, it tries to steal again. If a steal is successful the sleep time resets, otherwise it doubles for the next sleep duration.

ERASE consists of four main modules: *core activity tracer*, *power profiler*, *online performance model* and *task mapping algorithm*, seen in figure 2.2. Out of the four main components, the power profiler is the smallest and most simple. It is essentially a data structure that stores the idle power consumption for each core cluster, and the estimated power consumption when executing the different task categories. It takes as parameters, the task type, core type, frequency and resource width. From this, it returns the estimated power. The values used are gathered during a micro-benchmark performed during boot or install time. The values are thus very platform dependent and can not be used on another platform.

The core activity tracer is also simple but useful. It is implemented as an integer array in which the cores track their respective status and thus, track the current utilization of the system. This is vital for calculating and differentiating between the idle and execution power of the different clusters.

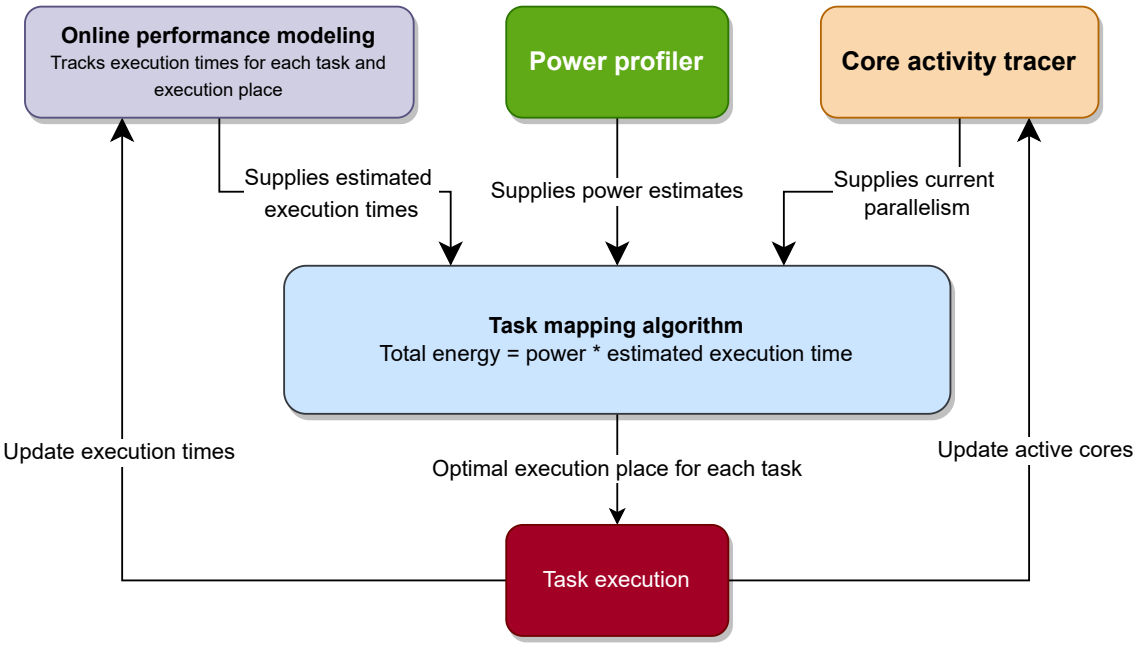

Figure 2.2: A brief overview of the ERASE scheduler, all of its core components, and how they interact.

The performance modelling is done by the module with the same name. It is a history-based model, so it draws its conclusions based on previous events. It takes core type, resource width and task type as inputs and returns the estimated execution time. On startup, all the values of the table are initialized to zero. When a task is completed, the lead core of that specific task updates the model for the correct core type, task, and resource width. The updates are done using a weighted sum which is balanced to give more influence towards older execution times over newer ones. ERASE makes no assumptions about unseen configurations and must thus try every configuration at least once to determine its energy efficiency. The model tracks the number of cycles and execution time of a task and from this, it interpolates the frequency of the core that executed the task. The frequency is then compared to previous values, if it diverges too much from previous events the model is reset and starts from zero again. This allows it to handle and adapt to a changed dynamic voltage and frequency scaling (DVFS) configuration during run-time and predict accurately under the new computing environment.

The final key component in the scheduler is the task mapping algorithm. It can be seen as the brain of the scheduler and determines the most energy-efficient execution place for a task. It iterates over all possible combinations of execution place and resource width. For each combination, it consults the other components to obtain the estimated execution time and power. From this the approximated energy consumption for a task can be calculated by $Energy = (Idle\ power + run - time\ power) * Execution\ time$. The combination with the lowest energy value is the most energy-efficient and is where the task will be scheduled. The task mapping can be performed on many cores concurrently allowing each core to schedule any task they release or encounter.

2.5 OpenMP

OpenMP is a specification and API meant to ease the development of parallel applications. It consists of various compiler directives, libraries and environment variables[15]. Furthermore, it can be used to add parallelism to already existing applications without much code refactoring. In addition, it does not require a deep knowledge of parallel programming. The most simple and well-known directive is the parallel for, which splits the iterations of a for loop between the running threads and executes them in parallel. Most well known OpenMP implementations such as the ones in GCC and CLANG, use a pool of threads that it assigns computations to instead of creating new ones for each separate event. The API is accessed through compiler directives through pragmas. Several different pragmas exist for different needs, but as previously mentioned in section 1.2, only two pragmas will be relevant for this project. In addition to the clauses there also exist multiple clauses to control scheduling, shared values and dependencies between different pragmas. Since OpenMP is a specification and not an implementation, the exact behaviour of the run-time differs between different implementations even if the API is the same from a programmer's view.
The *task* pragma takes a block of code or scope and creates a new and separate task which executes the code in the block. When a thread encounters a task pragma during execution, a task is created and put in a queue to be executed by some thread. How the allocation and scheduling work varies depending on implementation. A simple example can be seen in listing 1. Here the two functions *foo()* and *bar()* that will be executed as individual and independent tasks. When a thread encounters the *task* pragma, the task is created. This task can then be picked up and executed by another thread.

```
void main(){
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            foo();

            #pragma omp task
            bar();
        }
    }
}
```

Listing 1: Example of two independent task created using the task pragma

The taskloop pragma is similar to the task pragma in that it creates tasks. It is applied to a for loop and creates a task for each iteration of the loop. It can almost be viewed as "syntactic sugar" for creating tasks in loops with less boilerplate. The un-

derlying mechanics of the run-time during execution and creation may vary between the two, they are mostly equivalent from a programmer's perspective. Listing 2 shows a example where N calls to the function *foo()* will be created and executed independently in parallel.

```
void main(){
    #pragma omp taskloop
    for (int i=0;i<N;++i){
        foo();
    }
}
```

Listing 2: Example of a series of independent function calls using the taskloop pragma.

2.5.1 Implementation of OpenMP in LLVM

LLVM is an open-source compiler toolchain, more on this in section 3.1, with its own OpenMP implementation. In this specific implementation, OpenMP runs as a work-stealing scheduler for both the *task* and *taskloop* pragmas, where each thread has its own queue for tasks. This stands in contrast to other implementations like the GCC implementation, which uses a centralised queue for all threads[16].

When the run-time encounters a pragma *task* block, a new task construct is created for all tasks and the memory address of the entry point of the task block, the so-called routine, is stored onto the task construct. Regardless of how many tasks are created at the same *task* block, they all share the same routine address between the tasks. This address can then be used as a unique identifier to distinguish between different *task* blocks. Each task is then added to the thread's local queue before it continues executing the code after the task pragma. This goes on until the thread reaches a barrier, either an implicit at the end of a parallel section or an explicit barrier set by the programmer. Then the thread starts taking tasks from its own queue and executing them one by one until the queue is empty. Meanwhile, all other idle threads are trying to steal tasks from threads that have any available. This is done by randomly selecting a thread to see if they have any tasks available for execution. If so, the task is stolen and executed by the stealing thread.
When the program is recursive to some degree and the stolen task generates more tasks they will be executed before more tasks are stolen. The stealing works by constant busy polling, i.e. they are constantly checking for possible victims without any delay. This behaviour often leads to the main thread creating all the tasks and the other threads executing them, given that no recursion exists in the code. Besides this, there are various checks to see if a specific task is allowed to be stolen/executed by a given thread due to dependencies between tasks and other internal mechanics of the run-time. The execution logic of the task construct can be seen in figure 2.3.

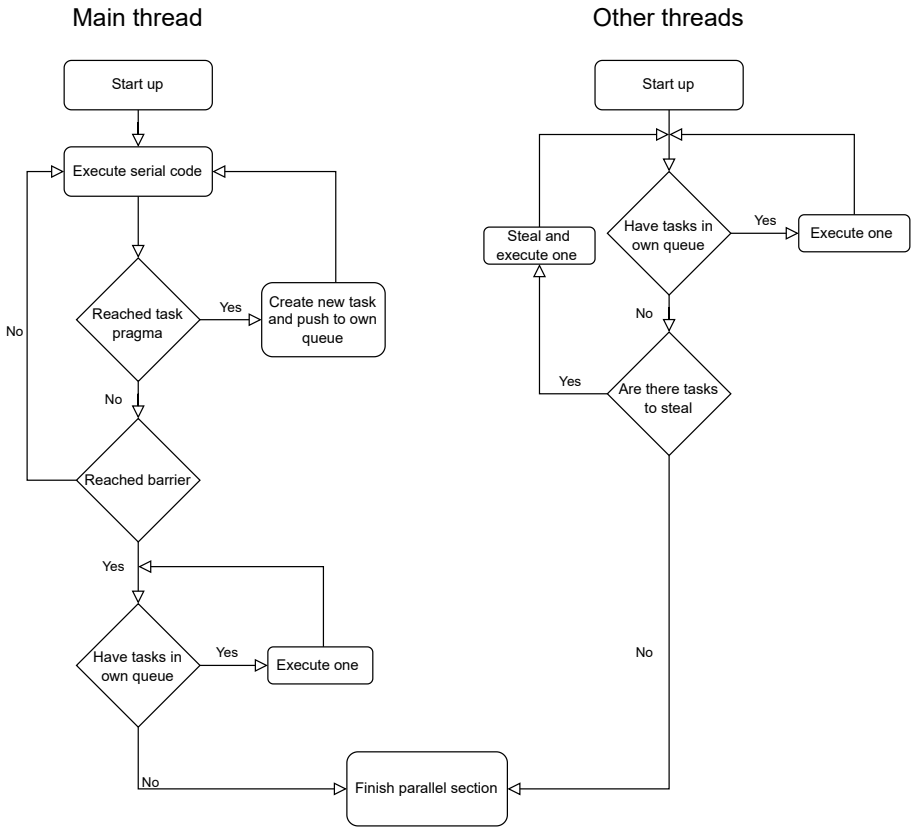

Figure 2.3: A flowchart depicting the execution pattern for the task pragma.

The *taskloop* pragma works a bit differently inside the run-time compared to the regular *task* pragma. It operates on a recursive divide-and-conquer strategy. The thread that encounters the pragma, calculates based on different parameters if the task should be handled recursive or linear. The linear behaviour is very simple, it iterates through all the loop iterations, creating a new task for each iteration and pushing it to its queue. In the recursive behaviour, there is a limit to the number of pattern tasks, which are tasks that only serves the purpose of creating new tasks. This is calculated either with information from the programmer or internally from the run-time and determines the level of nested recursion that occurs. If the current number of pattern tasks left to create is greater than a set threshold, the loop is split into two pattern tasks. The upper half will be created as a new task that is pushed to the queue of the thread and can be stolen while the current thread calls the recursive behaviour on the lower part of the loop. This continues until the number of tasks left to create is zero, then the pattern tasks switch to linear behaviour for their part of the loop and create the tasks that make up the actual loop-body. These are pushed to their respective queue and eventually executed until all iterations of the loop are done.

3

Implementing ERASE into OpenMP

This chapter will first present and describe the sets of tools used for this project. Which will be followed up with an explanation of the workflow used during the project. The end of the section contains a general description of the necessary modifications to the default run-time. The modified source code used during the project is publicly available[1].

3.1 Tools

Several different tools were used during various stages of the project. Some were used for information gathering while others are incorporated into the source code. What follows is a quick summary of their general usage and what role they played in the project.

LLVM

LLVM is a set of open-source modular compiler and toolchain technologies. It is capable of compiling from many programming languages to several different Instruction set architectures, or in short, ISAs[17, 18]. This is accomplished by having multiple front-ends that convert the source code to LLVM intermediate representation (LLMV IR)and hand it over to the middle-end. In this phase, multiple optimizations are performed on the LLVM IR, since all front-ends convert to the same middle-end it can be reused multiple times. The LLVM IR is eventually passed over to a back-end, which in turn converts it to machine code for the intended ISA. A visualisation of the components in the toolchain can be seen in figure 3.1

[1] `https://github.com/Grillan/llvm-project`

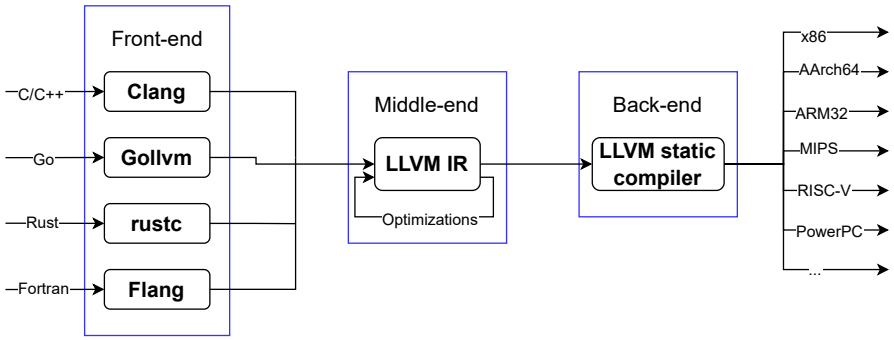

Figure 3.1: A summary of the key components contained in the LLVM compiler toolchain.

perf

perf is a performance analyzing tool available in the Linux kernel since version 2.6.31 released in 2009 and is used to measure and analyse system performance[19]. One of its functionalities is that it uses a set of hardware and software performance counters to track events such as the number of instructions, cache accesses and branch misses among many others. perf offers a generalised interface regardless of hardware which makes it easy to use and makes code migration to other hardware easier. However, some of the supported events of perf can be hardware-specific. It works by making a system call to the Linux kernel with a description of the event to track, event configurations along with a file descriptor and a shared memory region. perf then reads the memory region to track the counter.

Portable Hardware Locality

Portable Hardware Locality (hwloc) is an open-source software package for determining the hardware topology of modern architectures[20]. One of its main points is, as the name suggests, portability. This is accomplished by using information directly from the operating system instead of reading the topology itself. The information it provides the user ranges from cache size, shared caches between cores, available threads, locality of IO devices and much more. It supplies the user with this information either through the terminal or via a detailed API. An example of the graphical output can be seen in figure 3.2. It can also be used to set the core affinity of threads and processes.

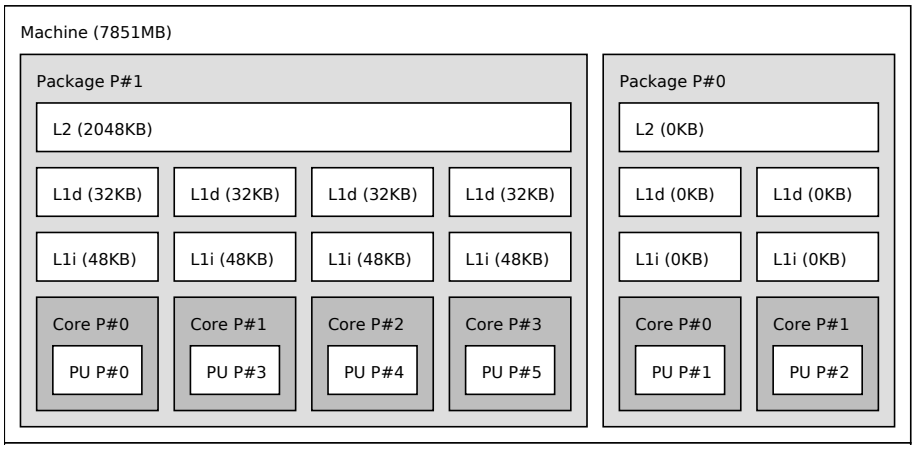

Figure 3.2: The graphical output from hwloc on the TX2 platform.

ALGLIB

ALGLIB is a multi-language numerical analysis and data processing library[21]. Its main selling points are its portability, easy usage and high performance. It is free of use for personal and academic purposes. In the project, it is used for interpolation purposes.

3.2 General workflow

The first step in the process of developing on the LLVM implementation OpenMP library is to set up a project containing LLVM, Clang and OpenMP. Clang links the OpenMP library with a target program which utilises OpenMP pragmas in its code. The Clang compiler then breaks down the program code into LLVM IR which is then turned into machine code after optimizations by LLVM.
The LLVM-OpenMP library is a big and complex library with many of its files containing several thousands of lines of code. A considerable amount of time when implementing a task scheduler into its run-time is therefore spent on analyzing and understanding the source code and thus the inner workings of the run-time. The first step of this process is to understand how the library works on a high level. This knowledge is vital in the second step, which is to determine where to inject the desired tools and functionality into the source code.
Due to the majority of the project being about modifying a run-time system, this comes with its own unique set of challenges. Since the aim is to have a functioning OpenMP implementation with ERASE as a task scheduler it is of utmost importance that any changes still adhere to the OpenMP specification. Failing to do so would mean that many OpenMP applications might face compatibility issues. Another challenge to consider is the fact that the run-time cannot run on its own but needs another program to function. This leads to an iterate approach were a feature of ERASE would be implemented and then verified for its correctness. For this, a short test program designed to test the intended feature was developed. When the intended behaviour of the run-time for the new feature was verified the next feature

could start being implemented.
The OpenMP task clause does not have a way to define task type, which means that the run-time does not distinguish between different tasks. This means in effect, the run-time treats all tasks the same. This is a problem since for ERASE, each task is required to be defined into a specific type to effectively schedule the task onto an optimal execution place given its properties. Instead with this implementation, tasks will have to be defined online during run-time. As previously mentioned, ERASE uses the arithmetic intensity of tasks to define task types, as seen in equation 2.4. Out of the four factors, two of them, the number of FLOPs per cycle, and the length of a cache row are platform dependent. Which means they are static during run-time. While the other two, the number of cache misses and clock cycles are unique for each task. Here is where perf comes in. During execution, perf event counters are used to track the execution of the individual tasks which makes it possible to get the arithmetic intensity of each task. Afterwards, the task can be classified into one of the three different task types.

3.3 First prototypes

For the first version of the ERASE implementation, the focus was put on supporting the *task* pragma which means that task moldability was not a priority. Thus, the basic *task* pragma does not support splitting a task to be executed by multiple threads. It can only be partway executed by a thread and then completed by another thread but that is not the sought after functionality. Something similar to moldability can be done using the *taskloop* pragma. What follows is an explanation of the first implementation of ERASE into OpenMP beginning with a description of how tasks are scheduled which can also be seen in figure 3.4.

3.3.1 Initial implementation

As previously mentioned, when tasks are created during execution they were previously attempted to be pushed to the current thread's queue. Instead, the task mapping algorithm of ERASE is called to determine an optimal cluster of execution, this process can be seen in figure 3.3.
Whenever the scheduler encounters new undefined tasks for the first time, the tasks are scheduled on the fastest cluster. When the undefined task is set to begin its execution, perf event counters detailing the number of clock cycles and last level cache misses are opened. Keeping perf counters constantly active results in performance degradation, thus they are only active during the classification of undefined tasks. Eventually, when the execution of the task is completed, the perf counters are read and closed again. The arithmetic intensity is calculated and the task is categorised into the correct type. What follows is that the task routine address is stored in a shared thread-safe hash table with the corresponding task type.
When new tasks are up for scheduling their routine address is checked to see if they are located in the hash table before going on to being scheduled. If they are, then the task mapper iterates over each CPU cluster, calling the power module to retrieve the idle power value which in turn calls upon the core activity module to verify active

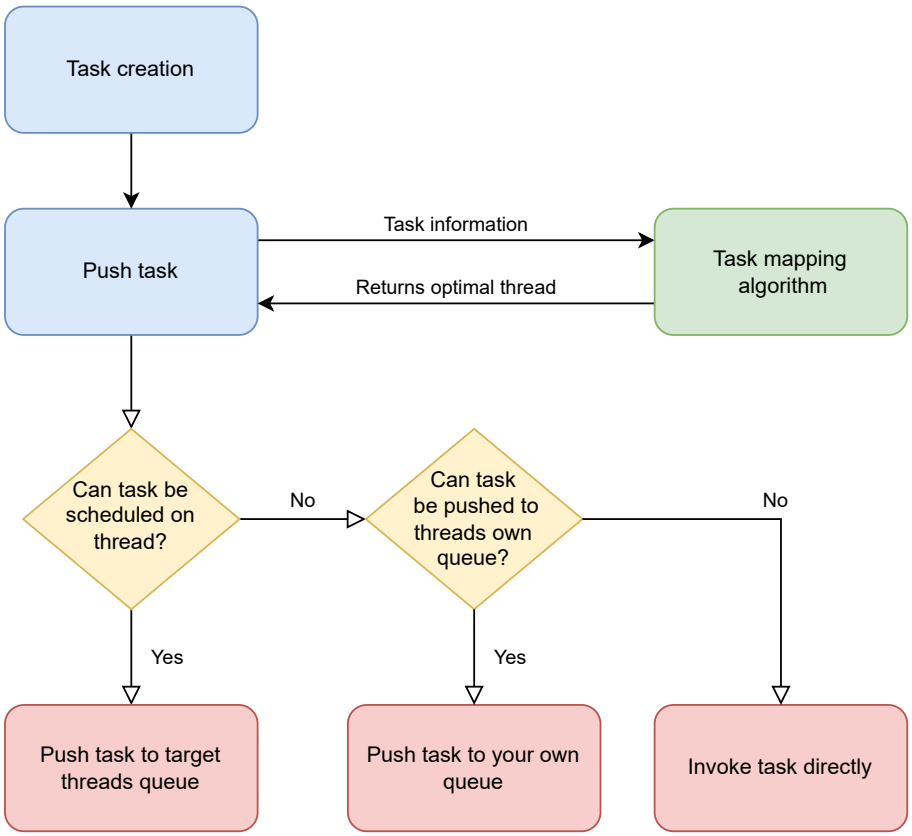

Figure 3.3: Displays how the first prototype of ERASE-OpenMP schedules tasks on threads.

threads. Where for example, if the task mapping module is currently determining the energy consumption of cluster A, and all of cluster B's threads are sleeping, then cluster B's idle power consumption is combined with cluster A's idle power. Then in the power module, the task types run-time power consumption is fetched and combined into total power consumption. Subsequently, the online performance module is called to retrieve an estimated execution time. The estimated energy consumption is calculated by multiplying total power by execution time. The cluster with the lowest energy consumption is then set as the optimal cluster. Since the execution time is initialised to zero, any task type that has no history of execution on a specific cluster will have a zero estimated energy consumption. Thus, that cluster will be targeted for execution. Finally, the core activity tracer module is called again to schedule the task on a non-active thread in that cluster, if none can be found, the task is scheduled on the current thread if it is located in the optimal cluster. If the thread is not in the optimal cluster it picks a thread in the optimal cluster and schedules the task there. In the case that the scheduled thread is sleeping, it will be woken up to minimize unnecessary sleep time.

When ERASE have found a thread to schedule the task on. Depending on if there are scheduling constraints like if the task is stuck at a barrier awaiting another task, or it may not be able to be transferred to the designated thread because of some execution condition. Then the task is pushed onto the current thread's queue if

possible, if not, the task is invoked for execution directly.

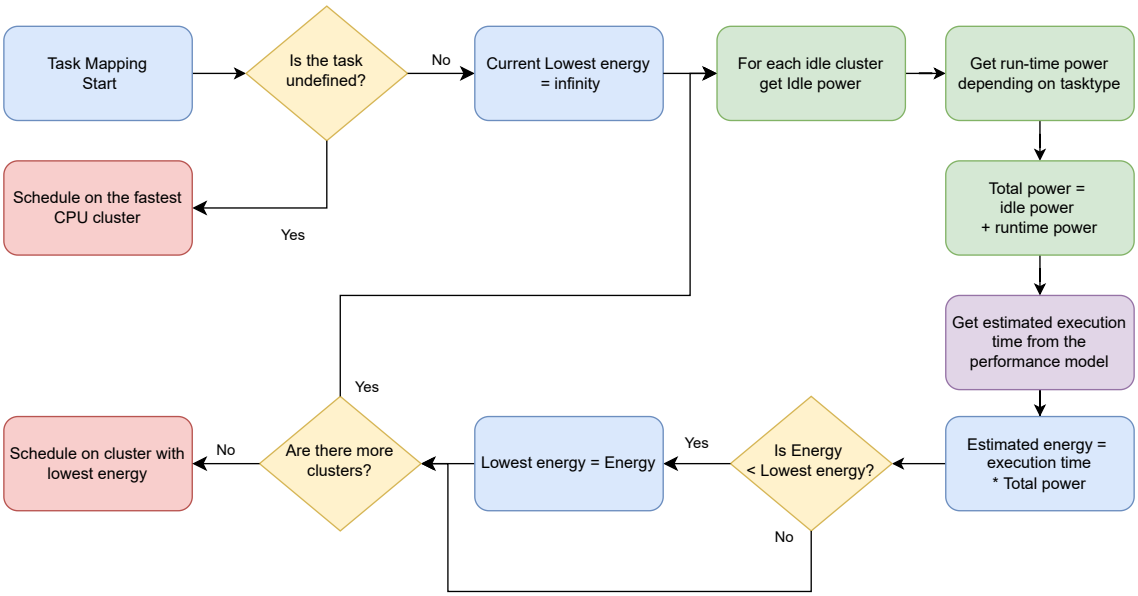

Figure 3.4: A summary of how the task mapping algorithm in ERASE-OpenMP is implemented in the first prototype.

When tasks are set to start executing, the current system time is sampled and if the task type is undefined, perf counters are also started before execution starts. Later when the task has finished executing, the system time is read again to get the task's execution time. Which is then added into the performance model using the weighted sum of the newly acquired time with the previous entry. Otherwise, if the entry was empty it just adds the current execution time. Because the execution time is stored using a weighted sum, frequency switches in a thread during run-time will affect the online performance model. Thus if DVFS changes or other external events, it can cause the most energy-efficient execution place to switch, then the model will detect this and adapt accordingly.

Another thing in need of modification is the initialisation of the thread's queues. In the default implementation, the threads initialise their own queues when they need them, usually when they push a task to their queue. This is not suitable for ERASE since it needs to be able to schedule tasks on any thread immediately from the start. The initialisation of the different queues thus had to be moved to a more suitable centralised location in the code.

Since OpenMP in LLVM is a work-stealing run-time other threads will try to steal tasks from each other if their own queue is empty. To preserve energy, a thread will check for possible victims to steal from up to a total of five times. Unless no task was found, the thread will put itself to sleep exponentially longer each time it wakes up and continues to not find any task to steal. When a thread successfully steals a task or is scheduled one the time is reset.

3.3.2 Taskloop support implementation

A strong functionality of ERASE is the ability to dynamically split tasks into smaller parts depending on the load of the system. A functionality the *taskloop* pragma is capable of. Thus, the main focus of the next implementation was to get the *taskloop* pragma to utilise ERASE. What follows is an explanation of how this was implemented.
When a code section containing the *taskloop* pragma is encountered, the number of loop iterations is derived from the loop's lower and upper bound. If the number of tasks or the grain-size clause is set, the loop is split into individual tasks based on the input given. These are then treated as tasks created by the *task* pragma. Such as they are scheduled individually by the task mapping module which is explained in section 3.3.1. Instead, if there is no clause set, the taskloop mapping module is called to determine if the *taskloop* will be split, and if so, the number of new tasks to create from it. The taskloop mapping works similar to the task mapping module, although with an additional layer on top. Now iterating over each core cluster and each possible core width based on the number of cores located on the cluster, to determine the optimal execution place in terms of lowest energy consumption. If the task has no history of previous execution when entering the taskloop mapping module, the cluster with the best performance regarding execution time is chosen combined with the highest possible width to determine the task type.
Afterwards, the taskloop's structure is marked with the number of tasks it will be split into and the optimal cluster. It is then split into new tasks by copying the taskloop's structure and adjusting the lower and upper loop bound of each task depending on how many tasks are to be created. When this process is finished, it is then treated as an ordinary task. Thus, it goes through the same path as ordinary tasks explained in section 3.3.1. But given that the task has already been scheduled, instead of calling the task mapping module, the core activity module is called to find available cores on the designated cluster to push the task onto. If no available core is found the same policy is used as with the *task* pragma, which can be seen in figure 3.3.

3.4 Improving performance modelling

The performance modelling in the original ERASE scheduler is pure history-based and makes no assumption regarding untested configurations. This is an opportunity for improvement when it comes to estimating the execution time of split or moldable tasks, implemented as taskloops.
Two main ideas with different strengths or weaknesses exist. Either an interpolation model is created online using the data available to the run-time at that moment. This has the capability of being very accurate but can also have a great cost in terms of the computations needed to create the model. The other option is to create a more complex model offline using gathered data on how tasks behave when split using different resource widths.
The general idea for both methods is to interpolate missing values using known values to receive a rough estimate. The previously mentioned PMNF can be used to

estimate the scaling behaviour of different kernels given enough data. This function is however too costly to be calculated online and in order to interpolate a non-linear function at least 3 samples are required, which is ill-suited for the TX2 test platformed later described in section 4.3 since its largest cluster only has 4 cores. Thus 75% of all available widths on that cluster would already be tested severely limiting the impact of interpolation. On a platform with more cores in a cluster, a smaller percentage of the available width would require testing. The high cost of computing the functions remains. It is however suited as a model to be created offline.
To create an offline model data is needed. Thus various kernels are classified and run multiple times with different resource widths to gather data. Then the data is normalised to make the model independent of the task's granularity. When sufficient data is gathered and normalised mathematical function is fitted onto the data. This function may either be in the form of PMNF, a polynomial or something else that models the data with high accuracy. The average value of the normalised data is exported along with the mathematical function to the run-time.
To interpolate unknown data during execution two steps are required. First, the function is evaluated at the resource width to be interpolated. Since the data from the function is normalised it needs to be scaled using the available data points. This is done by dividing every data point that is available to the run-time by its equivalent normalised value and then the average value of them is calculated to obtain the average scaling factor. The evaluated value from the function is then multiplied by the average scaling factor to obtain the final interpolated value. A detailed descriptive algorithm can be seen in algorithm 1.
If instead, the model is to be created online, there are other requirements. Both the model creation and interpolation have to be quick to keep the overhead at a minimum. Fitting a polynomial to data can be costly to compute and suffers from the way ERASE tests configurations. Given a hypothetical cluster with 16 cores. Without any interpolation, ERASE will test the resource widths in ascending order. The interpolation model could have values for the widths one to five. Fitting a polynomial to that data and evaluating it at 16 would likely lead to a value smaller than zero or much greater than the value at width one. Both of these values carry little interpolation value. Instead of fitting the data with a polynomial, a linear spline or piece-wise linear function is a better candidate[22]. It creates linear functions between known data points and when interpolating it simply evaluates the function at the desired width. It is fairly easy to compute and performs better when predicting data far away from previous values.
In total three models will be tested. Two models were created offline, one with a polynomial function and the other with one of the PMNF forms. One online function will also be tried, namely, interpolations using a linear spline.

3.5 Final product

The final working version of ERASE in OpenMP essentially behaves the same as the model it is based on, apart from some key differences. First of all, since it is implemented into one of recent LLVM builds (as of the publication of this thesis),

Algorithm 1 Algorithm for creation and interpolation using an offline model

Phase 1 - Data gathering

Run multiple kernels to gather data
Save performance table vectors

Phase 2 - Offline model creation

for each task type **do**
  $sum\_vec \leftarrow \{\}$
  for each saved vector **do**
    Normalise vector
    $sum\_vec \leftarrow sum\_vec + vector$
  end for
  $sum\_vec \leftarrow avg(sum\_vec)$
  Fit function $f(x)$ to sum_vec
  Export $f(x)$ and sum_vec to run-time
end for

Phase 3 - Interpolate value at X

$y \leftarrow f(X)$
$scale \leftarrow 0$
for index i in performance table[cluster][type] **do**
  if performance table[cluster][type][i] = 0 **then**
    Continue
  else
    $scale \leftarrow scale+$ performance table[cluster][type][i]$/sum\_vec[i]$
  end if
end for
$scale \leftarrow avg(scale)$
return $scale * y$

it requires at least Clang 14.0 to work properly.

Moving the classification online also has some unforeseen consequences. It struggles with recursive algorithms similar to that in listing 3. A task is only classified when it has been computed to completion at least once. A recursive algorithm will often create a tree structure where each node will spawn more tasks and await their results. A task will only be fully completed once a leaf is reached and by then most if not all of the tasks have already been created and scheduled. They will thus be computed correctly, but will not utilise the potential of ERASE since they will be scheduled as undefined tasks.

Another consequence of moving the classification online is that many tasks may be created before it has been classified. When a code as seen in listing 4 is executed, one thread will continue to create all tasks before invoking them. This can lead to many undefined tasks being created and not scheduled optimally. Since task creation usually is much quicker than execution, it is not guaranteed that any task is completed before all tasks are created. However, as soon as one iteration is completed by any

```
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
    {
      #pragma omp task shared(i) firstprivate(n)
      i=fib(n-1);

      #pragma omp task shared(j) firstprivate(n)
      j=fib(n-2);

      #pragma omp taskwait
      return i+j;
    }
}

int main()
{
  int n = 10;
  #pragma omp parallel shared(n)
  {
    #pragma omp single
    fib(n)
  }
}
```

Listing 3: Example of a recursive algorithm using the task pragma

thread the problem disappears since the task has been classified. Fortunately, this can be solved by restraining the creation of tasks. If a thread encounters an undefined task and the number of tasks in its queue is above a threshold the task will be invoked directly instead of pushed to the queue. This threshold is set to twice the amount of cores in the system. This provides other threads with enough tasks to steal while task creation is paused.
The memory usage of the scheduler and the run-time can be seen as a merge between OpenMP and ERASE. It uses the same performance tables as ERASE and almost the same thread and task structure as OpenMP. The added data fields to the task and thread structures are small compared to the original sizes and should have a negligible impact on performance. The one thing added that does not scale like any previous structure is the routine hash map. While other data structures scale either with the number of threads or cores in the system the routine hash map scales with the number of unique tasks the program executes. The performance consequences of this are hard to analyse from a theoretical standpoint since it will be very dependent on the application.

```
void main(){
    #pragma omp parallel
    {
        #pragma omp single
        for (int i=0;i<N;++i){
            #pragma omp task
            foo(N);
        }
    }

}
```

Listing 4: Example of a for loop creating many tasks using the task pragma

Finally, OpenMP does not acknowledge the existence of one of the clusters on the TX2 and is thus unable to schedule threads there. This problem persisted even when OpenMP uses *hwloc* to detect the topology of the system even though *hwloc* by itself could correctly determine it outside of OpenMP. As a solution, each thread had its process affinity manually set in the source code.

4

Experimental Methodology

This chapter describes the experimental methodology regarding what performance metrics were considered in the evaluation of the ERASE implementation. How the implementation of different sections of ERASE affects the performance, such as the implementation of perf. What testing platforms were used to perform these experiments on. Followed by a look at the benchmark suites which were used to evaluate the performance. Finally a description of the various interpolation methods and how they are going to be used.

4.1 Performance metrics

With the primary goal of this thesis being to improve energy efficiency, energy consumption is the metric where the most emphasis will be at. However, execution time is important to evaluate the performance of applications in terms of speed. Another metric is the energy-delay-product or in short, EDP. It is the product of the energy consumed by the program and execution time. By itself, it is hard to draw any conclusion from the EDP. For example, if two applications have the same EDP of $n * m$, where one may have an execution time of m and energy consumption of n while it is reversed for the other application. These are two very different behaviours but with the same EDP. However, when EDP is presented together with consumed energy and execution time it can serve as a summary for the reader. This merits its inclusion in the results.

4.2 ERASE parameters

To evaluate the implementation of ERASE, the major sections from it consist of perf, the exponential back-off sleep, and the task mapping module. Each of these was isolated and tested to determine how much they affected the total performance. The complete range of parameters tested can be seen in table 4.1.
The behaviour of perf was evaluated using two different settings. Dynamic, meaning that the counters were only active when needed, i.e when classifying tasks. Off, meaning that the counters were turned off and not read from at all. The benchmarks that were tested with the last configuration only produce memory-bound tasks and subsequently all tasks were classified as such with perf turned off. This is not a realistic scenario and should only be used to see the performance impact of reading the counters and classifying tasks.

Sleep indicates the base time if the configurations sleep. The maximum number of shifts for sleep is the same, so shorter base sleep will lead to shorter max sleep. The number of failed steal attempts before sleeping was also the same for different parameters.
The work-stealing policy of ERASE was also tested, to determine how much the performance was affected by allowing or disallowing work-stealing between different clusters. Stealing between cores within the same cluster is always allowed for all configurations.
ERASE-OpenMP was tested alongside the original OpenMP implementation to compare the different versions. The original version with an implementation of the same back-off sleep functionality as ERASE-OpenMP was also tested to compare how much it affects performance.

ERASE parameters:	Settings:			
Base sleep duration	Off	10us	100us	1ms
perf configuration	Off	Dynamic		
Allow work stealing between clusters	Off	On		

Table 4.1: Different ERASE configurations that were tested.

4.3 Testing platform

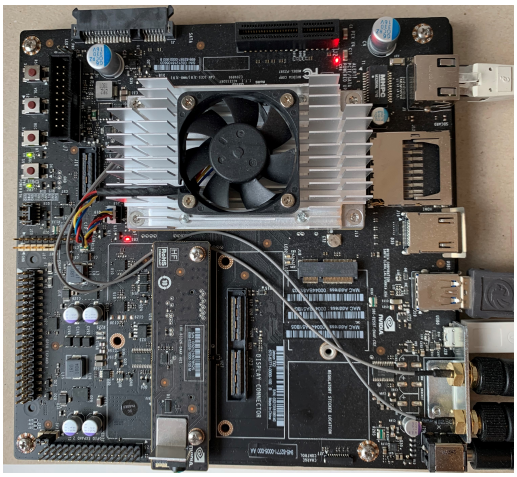

Figure 4.1: The Jetson TX2 module that was used to evaluate ERASE-OpenMP.

Almost all benchmarks were conducted on the Jetson TX2, which is a low power embedded computing device primarily made for artificial intelligence applications produced by NVIDIA[23]. It uses a Tegra X2 system on chip which contains a processor with a set of two different core designs, each based on the ARMv8 architecture split into two clusters. The system has a combined number of six cores, of which two of them are NVIDIA Denver 2 cores, and the other four are ARM Cortex-A57 cores. The board has 8GB LPDDR4 memory for its disposal with a memory bandwidth of up to 57.7GB/s and a 32GB eMMC storage capacity. It also contains an NVIDIA pascal type GPU architecture with 256 CUDA cores which will not be used. The machine operates with the Ubuntu 18.04.6 LTS operating system with the Linux 4.9.253-tegra kernel. Given the heterogeneous CPU architecture, the module will be used as a platform for testing and conducting benchmarks for this project's ERASE-OpenMP implementation. The TX2 platform also features INA3221 sensors

for measuring voltage, current and power.

4.4 Benchmarks

Here the different benchmark applications which were used to evaluate the performance of ERASE-OpenMP are described. Unless stated otherwise all benchmarks are compiled with the -03 optimisation flag.

Parsec

The Princeton Application Repository for Shared-Memory Computers, also called PARSEC [24]. Is a benchmark suite which is comprised of 12 multi-threaded programs with different properties, created to evaluate shared-memory systems. However, the latest PARSEC 3.0 release support OpenMP 2.0, which does not include the *task* pragma. There are several extensions of the original PARSEC which add support for other run-times that support multi-threading. One is PARSECSs which was created to evaluate the task parallelism in the PARSEC benchmark suite [25]. It officially supports 8 out of the 12 original parsec benchmarks using *task* pragma implemented in OpenMP version 4.0. However, It officially only supports the GCC compiler OpenMP implementation. With this project utilising the CLANG OpenMP implementation, each benchmark had to be tested and verified for compatibility issues. Due to compilation issues the benchmarks of *dedup* and *facesim* were not usuable. *canneal*, *freqmine* and *ferret* compiled but neither produced any output due to run-time errors, which was tested using the OpenMP version without the ERASE implementation. This leads to three benchmarks left which are as followed: *blackscholes*, which uses partial differential equations to calculate prices on finance options. *fluidanimate* that simulates fluids interaction animations. And lastly, *swaptions* which is a financial analyse simulation to price portfolios. Each benchmark has four different input sizes, which will all be used, ranging from small to what they call native, where execution times range from a few milliseconds to several minutes.

BOTS

The Barcelona OpenMP Tasks Suite or in short, BOTS. Is a collection of eight benchmarks created to evaluate the task parallelism in OpenMP on multicore architectures. [26]. Each benchmark has several different task implementations. For example, it supports the creation of tasks from multiple threads with the use of #pragma omp for directive or letting a single thread create all tasks. It also supports tied and untied types of tasks. Tied tasks mean that when a thread has started executing a specific task, only the very same thread is allowed to continue working on the task if it for example has been interrupted because of a context switch. Untied tasks do not have this limitation. Most of these benchmarks are heavily recursive by design. As previously mentioned in 3.5 ERASE will not be able to classify the task until it finishes executing. Thus, kernels that are recursive in nature will be excluded. The kernels from BOTS which ERASE-OpenMP will be evaluated on are *alignment* and *sparselu*.

Taskloop

With the *taskloop* pragma being introduced a few years ago, there was no prominent benchmark suite which utilises it. As mentioned earlier, a big strength of ERASE is the possibility of splitting up tasks when deemed necessary. Thus, testing the performance of ERASE with programs utilising the *taskloop* was necessary to more accurately evaluate this thesis implementation. Because of this, a minor benchmark was created to test how ERASE perform with different types of tasks each created with *taskloop*.

```
void main(N){
    double A[N][N], B[N][N], C[N][N] D[N][N], E[N][N];
    A = randint();
    B = randint();

    for (int i=0;i<100;++i){
    {
        //Memory copy
        #pragma omp taskloop shared(C)
        C = A;

        //Matrix multiplication
        #pragma omp taskloop shared(D)
        D = A X B;

        //2D Jacobi stencil iterations
        #pragma omp taskloop shared(E)
        E <- A

        #pragma omp taskwait
    }

}
```

Listing 5: Pseudocode of benchmark which utilises the taskloop pragma.

The kernel itself is quite simple, it allocates space for five, two-dimensional arrays. It continues to then fill two of the arrays with pseudo randomised numbers. Inside a for loop there are three different taskloop clauses. The first one is a straight memory copy kernel, copying the content from one matrix to another which represents the memory-intensive task. The second one is matrix multiplication, originally naive but optimised for better memory accesses, representing the cache intensive task. The third and last one is a 2D stencil Jacobi kernel representing the compute-intensive task. The pseudo-code for this benchmark can be seen in listing 5. The full code can be viewed in the appendix, listing A.1. There were two main purposes for the design of this benchmark. The first reason was to see how ERASE coordinates several different types of tasks arriving concurrently. The second reason was to have

dependencies between tasks, currently, the taskloop pragma does not support explicit task dependencies like those of the task pragma with the *depend* clause. Thus, by declaring a *taskwait* between the iterations it acts as a simulated dependency.

4.5 Interpolation

To evaluate the different interpolation methods, each interpolation model was tested using the taskloop benchmark seen in figure 5. They were then compared to the default ERASE performance model. The configuration used for all benchmarks was 10us, dynamic perf and no stealing between clusters. In addition to the normal metrics, each model was tested again and exported the predicted and actual time for the task. This data was then used to determine the root-mean-square error (RMSE) on each task type and size for each model. Since the number of taskloop pragmas is the same regardless of matrix size, the different sizes will also test how granularity affects the interpolation.
The interpolation space is limited on the TX2 platform, i.e the online model requires at least two samples making it unable to interpolate on the Denver cluster. Because of this, the interpolations were also tested on a platform with a higher core count. The platform for this is the Tetralith. It is part of the Tetralith of the National Supercomputer Centre at Linköping University[27]. It consists of two 16 core Intel Xeon Gold 6130 CPUs on separate sockets for a total core count of 32 and is running Linux version 3.10.0. Since ERASE will test all different widths it will spend roughly one-third of the iterations filling the performance table. This should give a good indication if interpolation helps reduce the amount spent filling the performance table and if it decreases energy consumption. To measure the energy consumption Running Average Power Limit (RAPL) was used[28, 29]. It is included in Intel CPUs since the Sandy Bridge generation and provides a running accumulative energy counter. The testing methodology will be the same as for the TX2 platform.

5
Results

This chapter presents the result of the different benchmark suites and some findings from the results. For each benchmark, the energy consumption, execution time and EDP will be compared with the emphasis placed on energy consumption. Unless stated otherwise all benchmarks are sorted in ascending order for energy consumption.
To limit the space of legends in figures and to increase the readability of the text the configurations have been abbreviated, They follow the pattern:

<version>_<sleep behavior>_<perf behavior>_<stealing constraints>.

The details of the different parameters are described in section 4.2. The full set of abbreviations can be seen in table 5.1. In the original version, the perf and stealing parameters are always pO and aS respectively since that functionality is not used. These are subsequently not displayed to minimize information overload. Furthermore, some results are omitted due to the sheer number of them. The missing figures can be seen in the appendix.

Parameter	Abbreviations			
Version	E: ERASE	O: original		
Sleep	nS: no sleep	10us: 10 us sleep	100us: 100 us sleep	1ms: 1 ms sleep
perf	pO: perf off	pD: perf dynamic		
Stealing	nS: no steal	aS: all steal		

Table 5.1: Table of the abbreviations used throughout the results chapter.

5.1 Parsec

To limit the sheer amount of data visualised, at first, only benchmarks with different ERASE parameters will be presented. The most interesting in terms of performance will then be compared to the different versions of the original OpenMP implementation.
Starting with *blackscholes* seen in figures 5.1 and 5.2. It produces only tasks classified as memory bound. With input sizes native and large a few trends emerge. Removing exponential back-off sleeping, which makes threads busy-wait instead, generally increases the energy consumption for both input sizes. Disabling perf leads to minor or no improvements indicating that the classification of tasks has a very small overhead. Disabling inter-cluster stealing increases execution time but not necessarily energy consumption. Likewise, increasing sleep time also increases execution time but is not guaranteed to increase energy consumption.

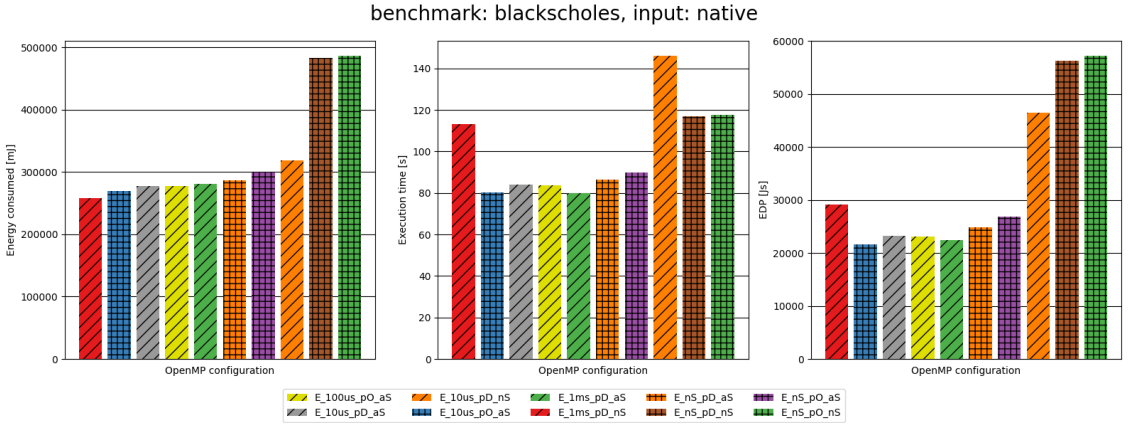

Figure 5.1: Energy consumption, execution time and EDP of the *blackscholes* benchmark from the PARSEC benchmark suite using native input size. Results are from different ERASE configurations.

With *native* as input size, allowing stealing between different clusters produces a lower energy number than not allowing it. The opposite is true for size large where not allowing stealing consumes less energy. The highest execution time for both inputs is where stealing is not allowed, it is not with the same configuration though. Not counting the configurations where perf is turned off, the one with the lowest energy consumption is with 1 ms sleep, dynamic perf and no inter-cluster stealing, E_1ms_pD_nS, for native, see figure 5.1. For large the most energy-efficient configuration is E_10us_pD_nS, i.e the same as for native but with a shorter base sleep time, figure 5.2. Turning off sleeping and at the same not allowing stealing yields the worst result in both cases. This should not come as a surprise since memory-bound tasks, which *blacksholes* contains a lot of, tend to be scheduled on the cortex cores. Meanwhile, the Denver cores are spinning and consuming unnecessary energy.

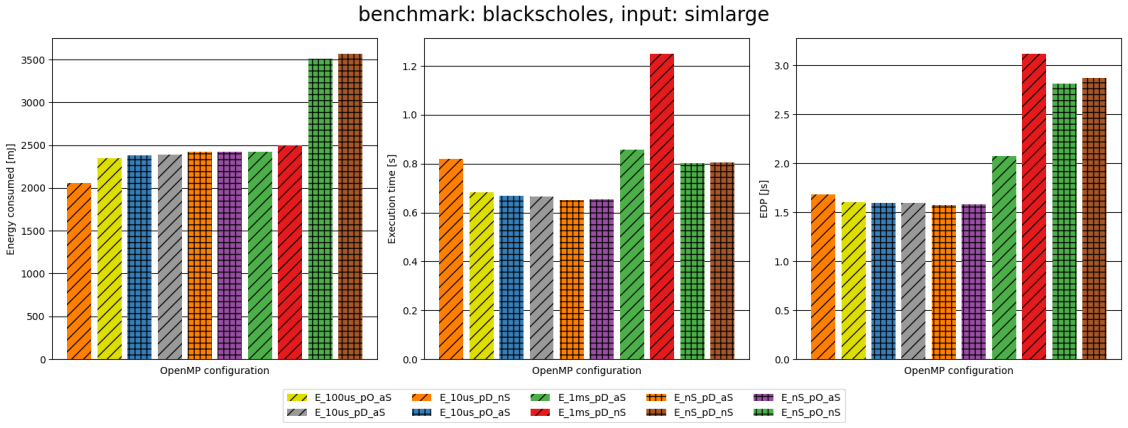

Figure 5.2: Energy consumption, execution time and EDP of the *blackscholes* benchmark from the PARSEC benchmark suite using a large input size. Results are from different ERASE configurations.

From these configurations three will compared against the original: E_10us_pD_nS, E_1ms_pD_aS and E_10us_pD_aS. Where the E_1ms_pD_nS configuration

which was the most energy-efficient for native size will be omitted due to its abysmal performance in other benchmarks. After comparing the chosen configurations against the original and its modified variants, more conclusions can be drawn. The result varies not only between benchmarks but also between different input sizes.

Continuing with *blacksholes* and *native* input size the three ERASE versions perform the worst and the best is the original version without any sleep, figure 5.3. It consumes 4% less energy than the best ERASE version, E_10us_pD_aS. Apart from E_1ms_pD_nS which is significantly worse than the others the differences between the rest are generally small. The difference between the best and second to worst is just 5% making it hard to draw conclusions. Execution times are also very similar except with E_1ms_pD_nS. This might be because all cores are equally busy executing similar tasks. As a result of the similar execution times, the EDP is also very similar for all configurations except one.

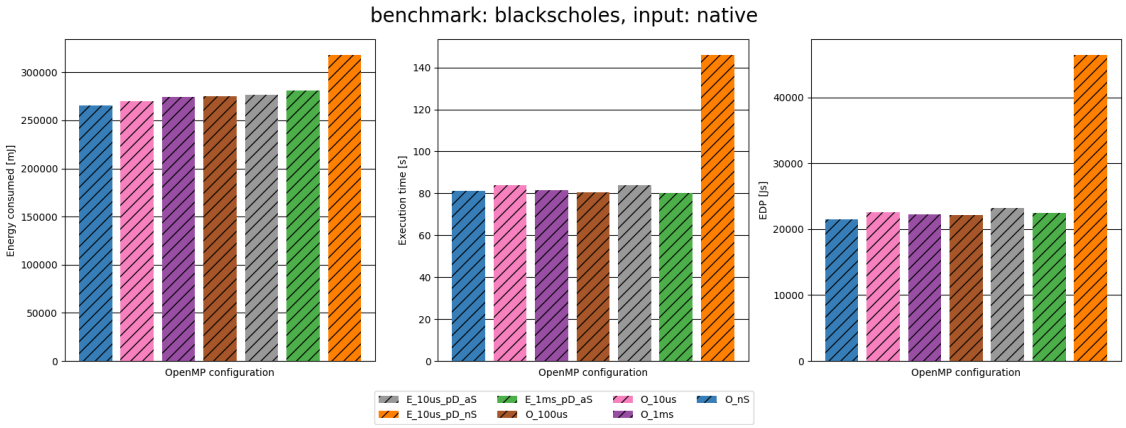

Figure 5.3: Energy consumption, execution time and EDP of the *blackscholes* benchmark from the PARSEC benchmark suite using native input size. Results are from different OpenMP and ERASE configurations.

When the input size is switched to *large* a different pattern emerges. The E_10us_pD_nS which was the worst previously is now the best and the O_1ms is the worst by far, figure 5.4. E_10us_pD_nS performs 14% better than the next best, E_10us_pD_aS, and 15% better than the original. In general all configurations except E_10us_pD_nS and O_1ms, once again perform very similar in terms of energy efficiency. In contrast to native, the execution times vary between different configurations and the original performs the best. Due to this the EDP also varies between them.

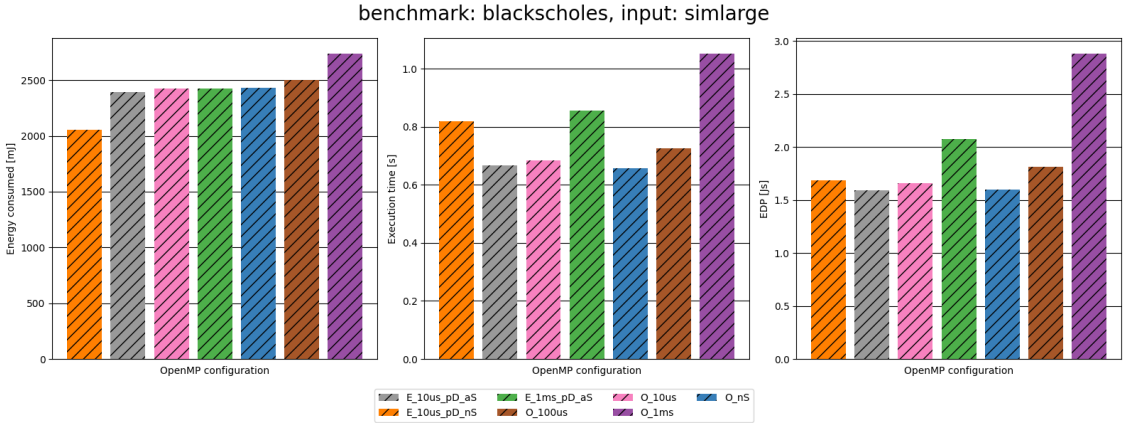

Figure 5.4: Energy consumption, execution time and EDP of the *blackscholes* benchmark from the PARSEC benchmark suite using a large input size. Results are from different OpenMP and ERASE configurations.

Switching benchmarks to *swaptions* using the *native* size, the best is once again E_10us_pD_nS, while the original is the worst. All are very similar in terms of energy efficiency except the best which is 15% better than the original and 11% better than the second-best. It is noteworthy that the best performing configuration does not allow stealing between clusters, allowing tasks to be scheduled in the most energy-efficient execution place and guaranteeing that they are executed there. It also has by far the longest execution time showing that no correlation can be drawn between execution time and energy consumption. The remaining configurations once again have a very similar execution time.

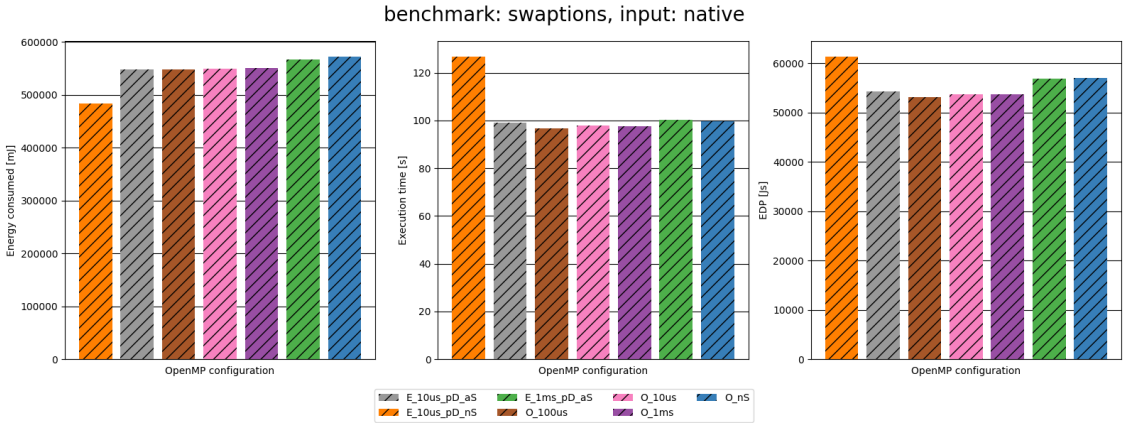

Figure 5.5: Energy consumption, execution time and EDP of the *swaptions* benchmark from the PARSEC benchmark suite using native input size. Results are from different OpenMP and ERASE configurations.

With the input size set to *large*, the tables are turned. E_10us_pD_nS now uses the most energy during its execution, figure 5.6. The best in terms of energy is now the O_10us. However, it is only 0.9% better than the best ERASE configuration, E_10us_pD_aS. This can be considered within the margin of error and it is thus hard to draw any substantial conclusions from these results. The best

version is just 8% better than the worst, further solidifying this fact. Execution times are also very similar for all the configurations except the one that does not allow inter-cluster stealing, E_10us_pD_nS. The likely reason behind the results is the fact that *swaptions* creates mostly memory bound tasks. When all tasks are fairly equivalent, it comes down to whether the non-optimal cluster is used or not. Whether that is better or not does vary depending on the input size, as seen in figures 5.5 and 5.6. The EDP is once again almost a replica of the execution times in their internal ordering and scale.

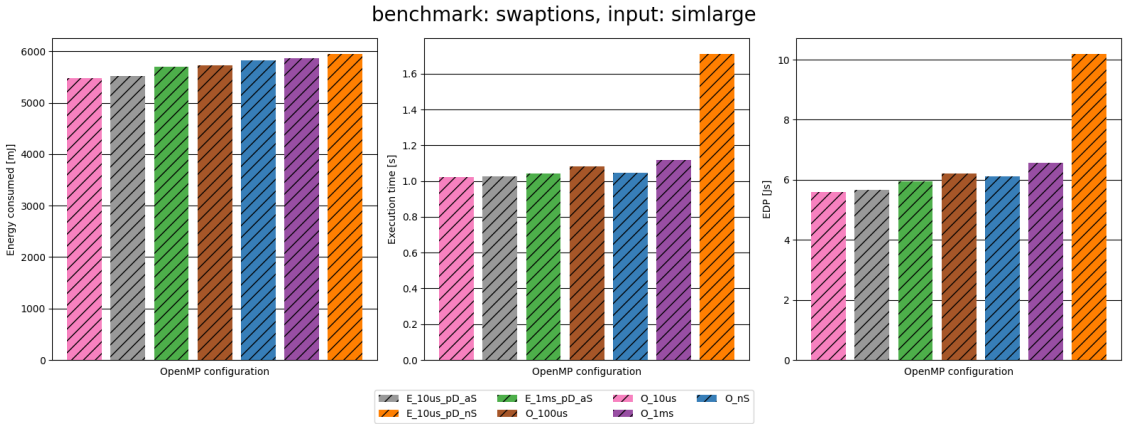

Figure 5.6: Energy consumption, execution time and EDP of the *swaptions* benchmark from the PARSEC benchmark suite using a large input size. Results are from different OpenMP and ERASE configurations.

The last available benchmark in the PARSEC suite is *fluidanimate*. With *native* as input size E_10us_pD_aS performed the best and beat O_10us by 0.5%, figure 5.7. This once again is too close to draw any meaningful conclusions. Though it is 17% more energy efficient than the original version. Furthermore, except for O_10us, the original versions performed worst in terms of energy. This is the first occurrence of this in this benchmark suite. Since both configurations with 10us sleep are so similar in energy it is hard to deduce if the lower energy is due to the added sleep or ERASE. However, the most probable is that the sleep duration is what is yielding the energy reduction since the results are very similar. Execution times are hard to draw any conclusions from, but in general, longer sleep tends to increase execution time as expected. Since the relative difference between the execution times is much greater than those of the energy consumption the EDP is almost identical in scale to the execution times.

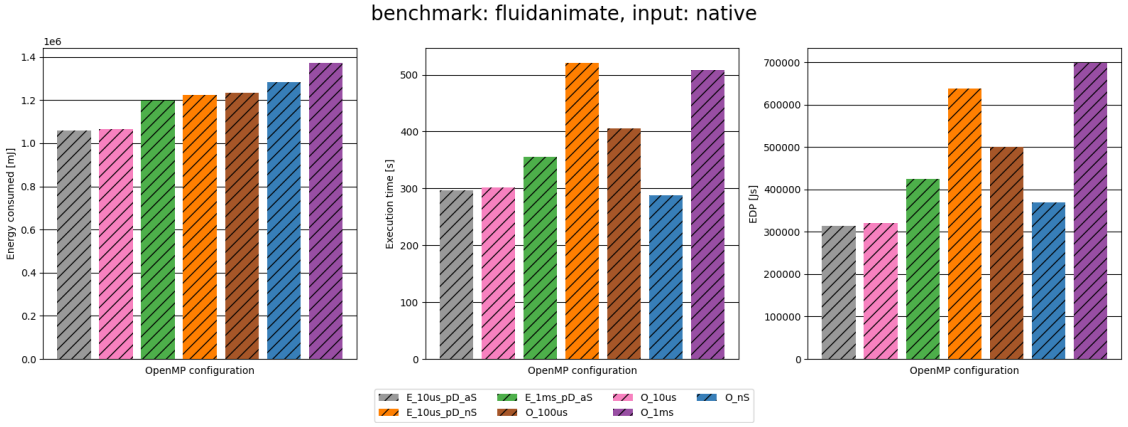

Figure 5.7: Energy consumption, execution time and EDP of the *fluidanimate* benchmark from the PARSEC benchmark suite using native input size. Results are from different OpenMP and ERASE configurations.

Decreasing the input size to *large* puts O_10us first 0.5% better than E_10us_pD_aS, figure 5.8. The best ERASE configuration is E_10us_pD_aS which uses 6.9% less energy than the original. As with the native input size allowing stealing between clusters both decreases energy consumption and execution time. E_10us_pD_aS outperforms E_10us_pD_nS by 17.9% with the only difference between them being the stealing policy. Execution time follows the usual pattern where shorter base sleep and allowing inter-cluster stealing lowers it.

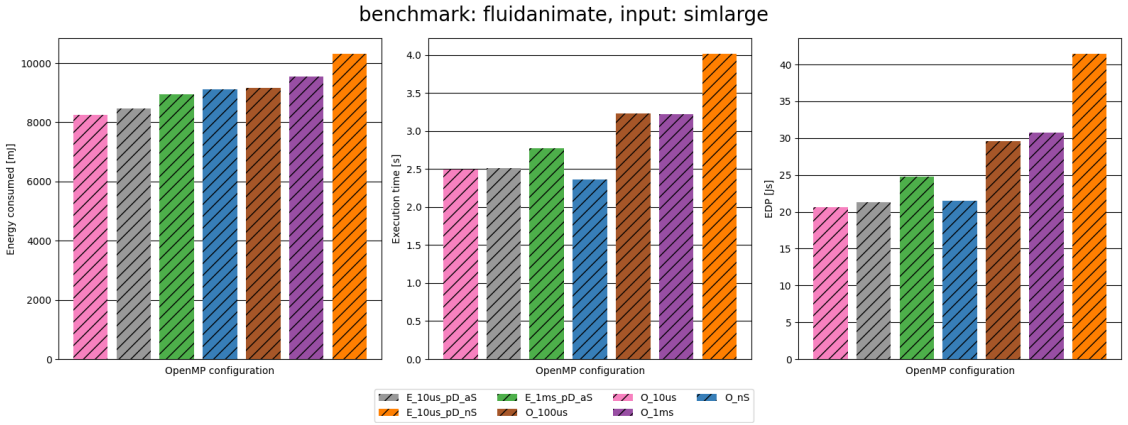

Figure 5.8: Energy consumption, execution time and EDP of the *fluidanimate* benchmark from the PARSEC benchmark suite using large input size. Results are from different OpenMP and ERASE configurations.

Trying to summarize an entire benchmark suite across all inputs is not easy and can be hard to draw decisive conclusions from. However, it can also be good at getting a quick overview of the result as long as you remember that it is an average of averages when conclusions are drawn.
With that in mind, the results were normalised for all the compared configurations. The average for every benchmark was calculated from all different input sizes and normalised values. A total average value from all the different benchmarks was

also calculated, the results can be seen in table 5.2. It becomes clear, not only from the summary but also from the individual benchmarks, that the easiest way to increase the energy efficiency of OpenMP tasks is to sleep threads instead of busy-waiting when they are unable to steal tasks. The most efficient ERASE configuration also clearly outperforms the original OpenMP version. Once again it is hard to determine how much of the improvement comes from the added sleep and how much from the other ERASE components. This is because of the similarity between the E_10us_pD_aS and O_10us configurations yet again. The ERASE version performs better on all metrics but by a very small margin. It might be the case that the benchmarks do not allow ERASE to make intelligent decisions. If the benchmarks feature different phases where all tasks created in a phase have the same characteristics, all ERASE can do is fill the available cores. Starting with the most energy-efficient instead of the default random ordering might be the reason the ERASE come out on top and perform slightly better.

Matrix size	Metric	E_10us_pD_aS	E_10us_pD_nS	E_1ms_pD_aS	O_100us	O_10us	O_1ms	O_nS
Blackscholes	Energy	**64.74**	65.33	73.93	75.65	68.06	96.53	69.07
	Time	47.54	67.48	68.23	58.18	51.06	88.92	**45.87**
	EDP	**34.05**	49.68	50.53	44.58	36.77	86.99	34.12
Swaptions	Energy	**88.28**	96.12	94.01	92.62	89.60	92.78	94.04
	Time	66.73	100	66.25	66.34	64.32	67.69	**63.71**
	EDP	62.29	100	66.08	64.90	**61.27**	66.43	64.04
Fluidanimate	Energy	82.38	96.09	87.40	89.80	**82.23**	97.58	88.52
	Time	61.06	100	69.33	77.38	62.54	84.96	**56.93**
	EDP	51.38	97.84	61.99	70.82	52.64	84.69	**51.34**
Total	Energy	**78.47**	85.85	85.11	86.02	79.96	95.63	83.87
	Time	58.44	89.16	67.94	67.30	59.31	80.52	**55.50**
	EDP	**49.24**	82.50	59.53	60.10	50.23	79.37	49.83

Table 5.2: Summarised normalised results from the parsec benchmark suite. The results for each benchmark show the average of all the available inputs. For all metrics, a lower value is better.

5.2 BOTS

The second benchmark suite is BOTS, and as with PARSEC at first different ERASE configurations will be compared and see what conclusions can be drawn.

Looking at the *alignment* benchmark with single task generation, figure 5.9, the results are very similar. This is unfortunately the case for much of the BOTS benchmark suite. What conclusions can be drawn is that disabling stealing seems to yield a small improvement. Otherwise, the results are too similar to make any assumption. The well-observed pattern is that the longer sleep time, in combination with not allowing stealing, increases the execution time.

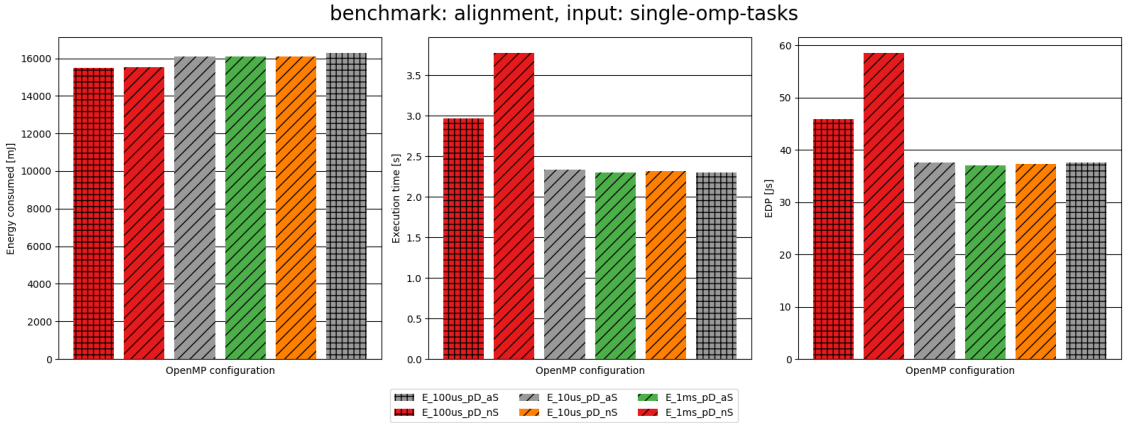

Figure 5.9: Energy consumption, execution time and EDP of the *alignment* benchmark from the BOTS benchmark suite using single task generation. Results are from different ERASE configurations.

Switching benchmark to *sparselu*, another pattern emerges. Lower energy consumption is achieved by allowing inter-cluster stealing, figure 5.13. All configurations that allow it, to perform fairly similar both in terms of energy and execution time. For the other three, it seems that increasing sleep time also increases energy consumption. Most likely this is due to the execution times which are two to three times longer. This behaviour is seen in some other benchmark suites as well when stealing is disabled.

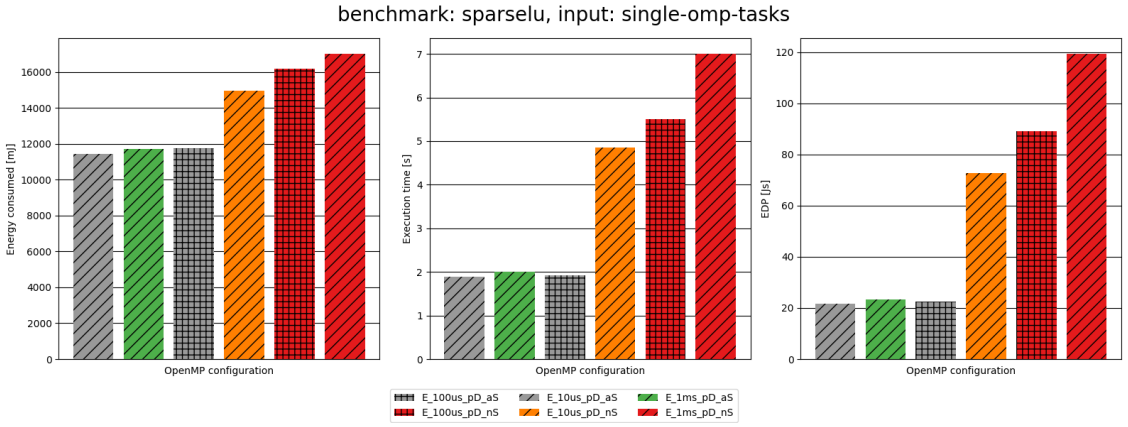

Figure 5.10: Energy consumption, execution time and EDP of the *sparselu* benchmark from the BOTS benchmark suite using single task generation. Results are from different ERASE configurations.

Comparing different ERASE configurations to OpenMP for *alignment* yields inconclusive results. Regardless of whether the task creation is from single tasks or single tied tasks the results are extremely similar, figure 5.11 and 5.12. For single task creation, the best configuration is 1.7% better than the worst, and for the single tied tasks, it is just 0.8%. The execution times for both versions are also very similar. This makes drawing any kind of conclusions from it hard. What can be said though

is that the layout of the benchmark does not hinder nor benefit from the increased overhead added on by ERASE.

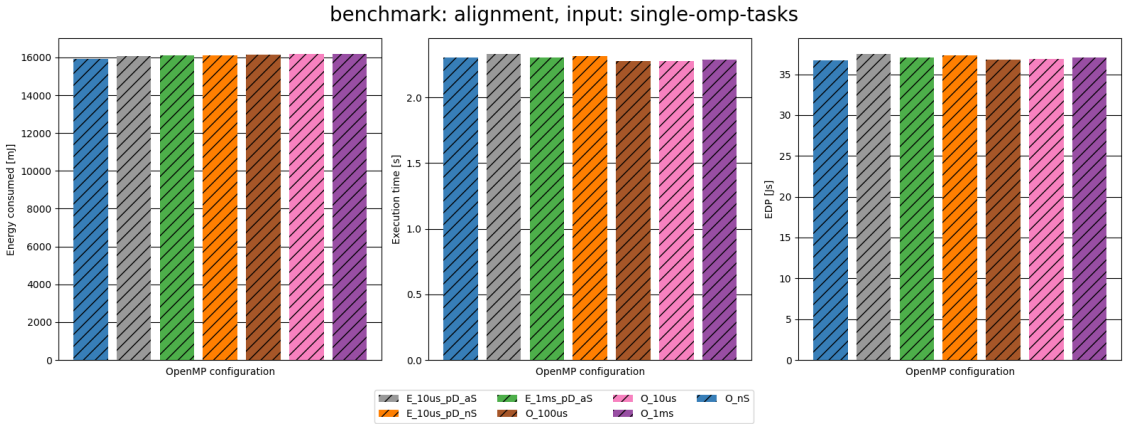

Figure 5.11: Energy consumption, execution time and EDP of the *alignment* benchmark from the BOTS benchmark suite using single task generation. Results are from different OpenMP and ERASE configurations.

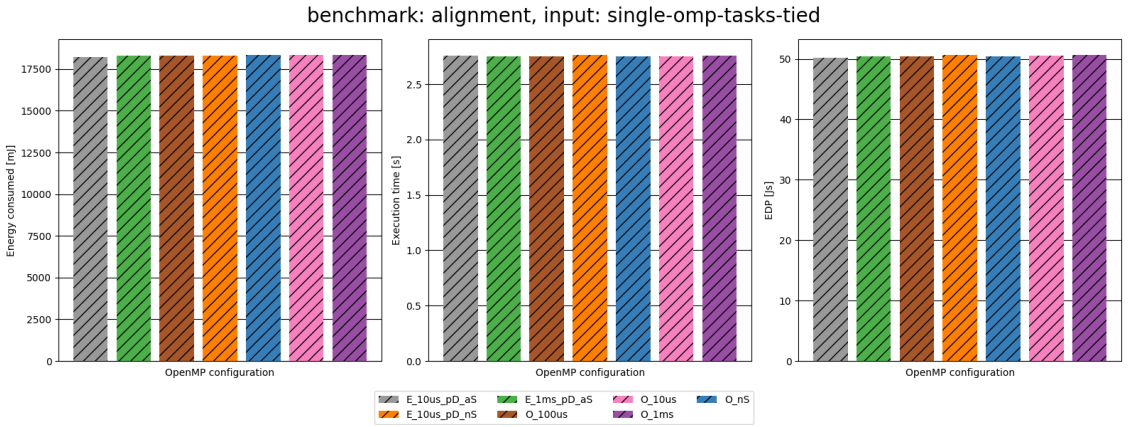

Figure 5.12: Energy consumption, execution time and EDP of the *alignment* benchmark from the BOTS benchmark suite using single tied task generation. Results are from different OpenMP and ERASE configurations.

The *sparselu* benchmark can like many others be evaluated in two tiers. Where the only difference between the tiers is the stealing behaviour. The better configurations all allow stealing and the worst-performing one does not, figure 5.13. When everything else is the same in a configuration allowing stealing results in 23% energy reduction. This is most likely once again due to the major increase in the execution time. Compared to the original OpenMP the best ERASE configuration shows a small but notable improvement of 3.5% while having a longer execution time. ERASE thus make some informed scheduling decisions which reduce energy. One possible scenario which has been observed previously is that the benchmarks create many tasks of the same type and the only decision ERASE can do is in which order to fill up the clusters.

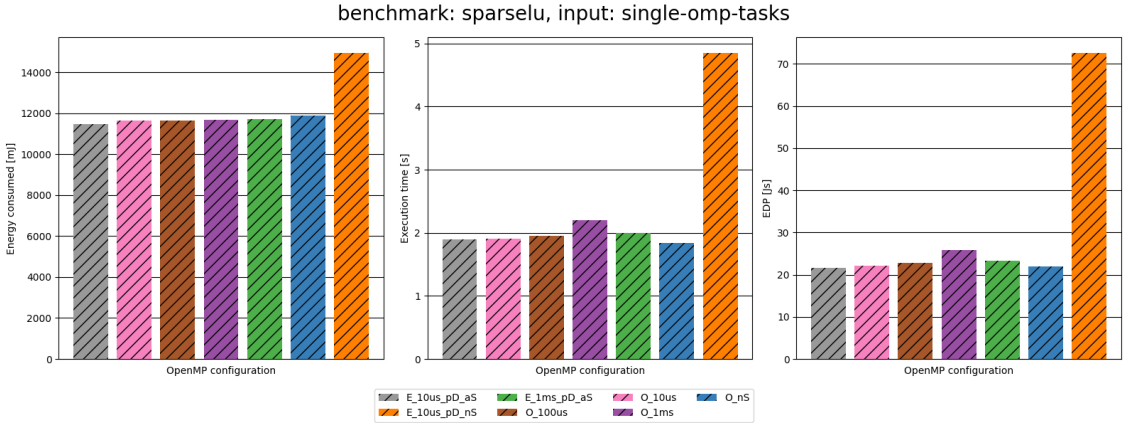

Figure 5.13: Energy consumption, execution time and EDP of the *sparselu* benchmark from the BOTS benchmark suite using single task generation. Results are from different OpenMP and ERASE configurations.

When the task creation is changed to a single tied task the roles are reversed. Now the configuration without stealing uses the least energy by far while all others are fairly equal,5.14. It also has a far longer execution time indicating that tasks are scheduled in their optimal place and are executed there.

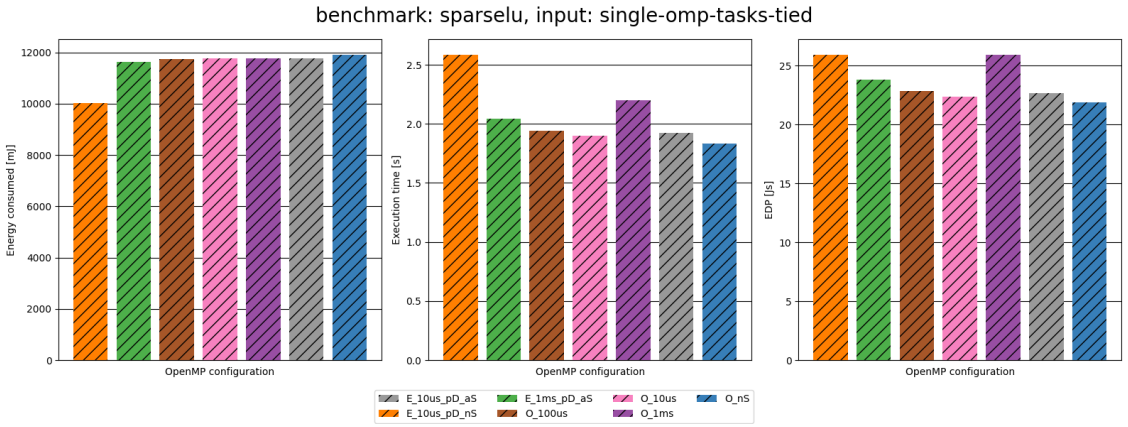

Figure 5.14: Energy consumption, execution time and EDP of the *sparselu* benchmark from the BOTS benchmark suite using single tied task generation. Results are from different OpenMP and ERASE configurations.

Summarising the BOTS benchmark suite is hard since almost all configurations, be that ERASE or the original, perform comparatively similar. Thus the whole benchmark suite is ill-suited for any lengthy discussions or conclusions.

5.3 Taskloop

The final benchmark is the custom taskloop benchmark. As previously different configurations of ERASE are compared before against each other before comparing to various OpenMP configurations.

Testing different ERASE configurations with a matrix size of 128 the top performers are the same from previous benchmarks,E_10us_pD_nS and E_10us_pD_aS, figure 5.15. They perform very similarly. However, when sleep is increased while allowing inter-cluster stealing it increases the energy consumption. The overall results also follow the pattern that increased sleep times also increase the consumed energy.

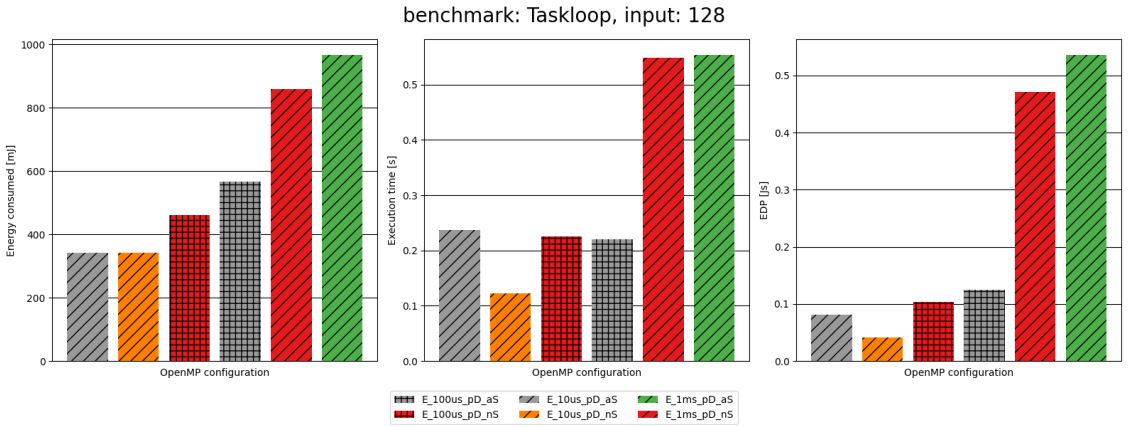

Figure 5.15: Energy consumption, execution time and EDP of the taskloop benchmark with 128x128 matrix size. Results are from different ERASE configurations.

Increasing the matrix size to 256 reveals another pattern. The energy results can be split into two tiers where the results within each tier are very similar, figure 5.16. The only difference between the tiers is sleep time. For the better ones, the times are either 10us or 100us and for the other it is 1ms. The likely reason for this is due to the increase in execution time that the longer sleep causes. Even if they are mostly sleeping during the extra time they are still using energy.
From these results, the configurations to be compared are the same as previously. The best performers are E_10us_pD_nS and E_10us_pD_aS to see how stealing changes the outcome. E_1ms_pD_aS is also included to see how the increased sleep behaves when compared.

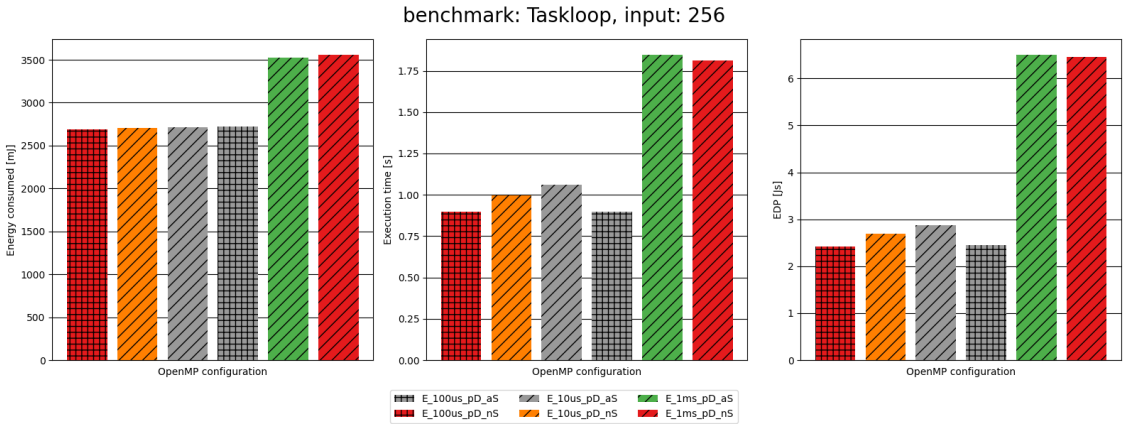

Figure 5.16: Energy consumption, execution time and EDP of the taskloop benchmark with 256x256 matrix size. Results are from different ERASE configurations.

Starting with the smallest matrix size it is clear that ERASE with 10us sleep is very energy efficient. The two variants with 10us sleep are the top performers with inter-cluster stealing enabled coming out on top, figure 5.17. It consumes just 0.2% less energy compared to when stealing is disabled. In regards to the original OpenMP, it is 26% more energy efficient. This once again shows that ERASE classifies the tasks and schedules them in an energy-efficient manner. One possible explanation as to why it does not matter whether stealing is enabled or not is the benchmark being run. The taskloop benchmark only releases a few tasks in each iteration. So when a core has completed its scheduled task there might simply not be anything to steal since the other cores already depleted their queues of tasks. Thus if stealing is enabled or not, does not seem to matter. E_1ms_pD_aS is the worst of all the configurations. Probably due to the long execution time as discussed earlier. Surprisingly the best OpenMP configuration is the default without any sleep compared to other benchmarks were adding a sleep generally improved energy efficiency.

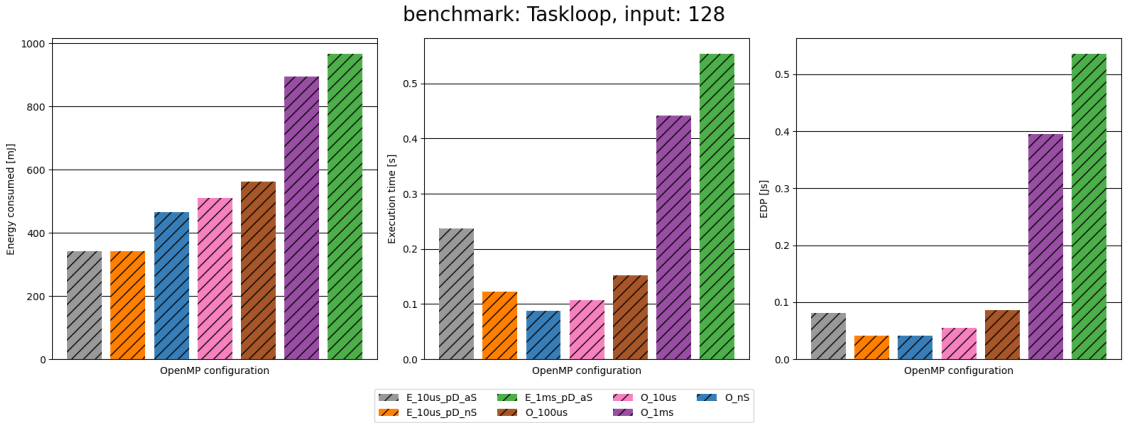

Figure 5.17: Energy consumption, execution time and EDP of the taskloop benchmark with 128x128 matrix size. Results are from different OpenMP and ERASE configurations.

When the matrix is increased to 256, results are once again broken down into tiers. Alas, this time it is three tiers instead of the previous two, figure 5.18. The two 10us sleep configurations consume the least energy and not allowing stealing is 0.2% more efficient. They are roughly 17% better compared to the original. In the middle tier are the original without sleep, 10us sleep and 100us sleep. The least energy-efficient are the configurations with 1ms sleep. Once again this shows that ERASE can classify tasks and schedule them in more optimal execution places. ERASE configurations are as usual slower than their original counterparts but for the most part more energy efficient.

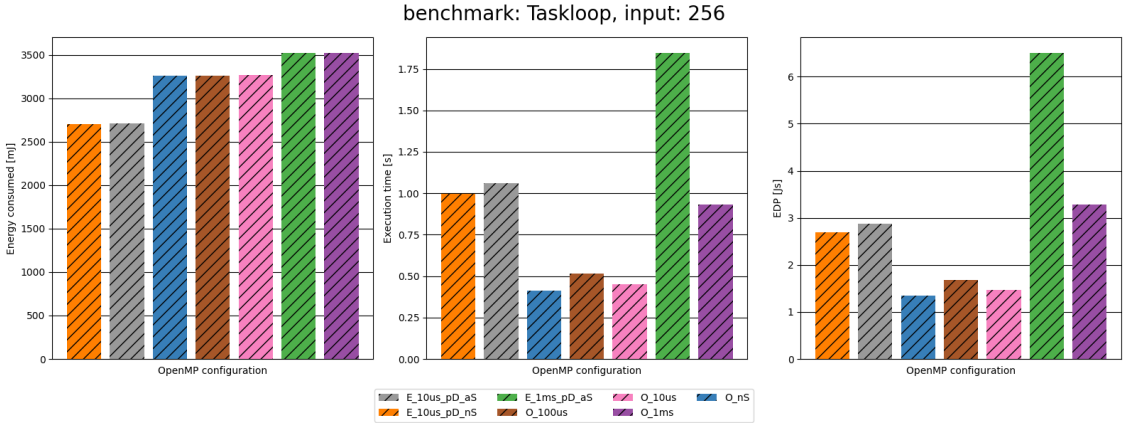

Figure 5.18: Energy consumption, execution time and EDP of the taskloop benchmark with 256x256 matrix size. Results are from different OpenMP and ERASE configurations.

Setting the matrix to 512 yields unexpected results. Almost all configurations use the same amount of energy while having drastically different execution times, figure 5.19. Except for the E_1ms_pD_aS which is the worst again, even though by just 7%, the difference between the best and second to worst is just 3%. It is unclear what causes this anomaly. The matrices do not fit into the cache, but as will be seen with a larger matrix, ERASE still performs better, so that should not be the reason. The E_10us_pD_aS is again the best although it is extremely close and thus hard to draw any conclusions from. In contrast to other sizes, the original without sleep is also the worst.

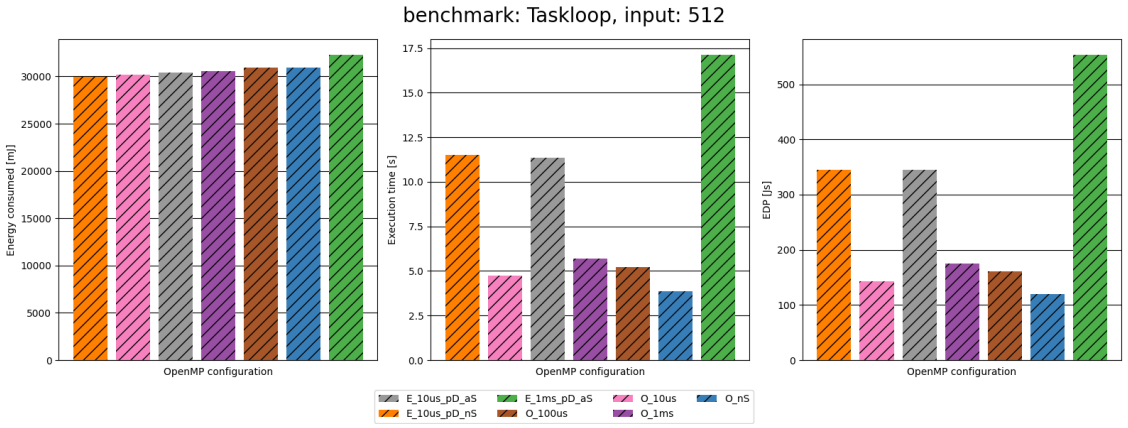

Figure 5.19: Energy consumption, execution time and EDP of the taskloop benchmark with 512x512 matrix size. Results are from different OpenMP and ERASE configurations.

The last matrix size is 1024. The results are more in line with what is expected. The two 10us ERASE configurations are again the most energy-efficient. Surprisingly the 1ms configuration outperforms all original configurations, figure 5.20. With all other matrix sizes, it is among the worst. The best ERASE implementation is 30% more energy efficient than the original. Adding sleep to the original also only increases

energy consumption. Why the 10us configurations perform well has been discussed previously and is only solidified by the new results. The sudden increase in energy efficiency for E_1ms_pD_aS might depend on several factors. The default may split the loops and the created tasks executed on different clusters. Since the clusters do not share any caches they might have to resort to synchronising via the main memory instead of only within the clusters cache. This should not be an issue for ERASE since it will not split a taskloop into more tasks than there are cores on the cluster. Also, when ERASE schedules a task on an idle core it is guaranteed to be executed on that cluster limiting the inter-cluster stealing. The larger matrix size might also diminish the relatively large base sleep time. Looking at the data increasing the matrix size decreases the impact of the sleep time.

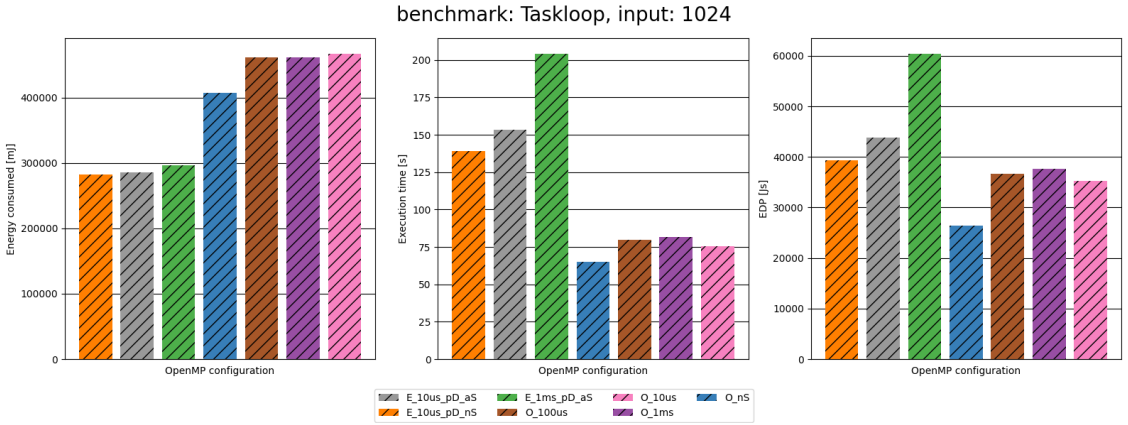

Figure 5.20: Energy consumption, execution time and EDP of the taskloop benchmark with 1024x1024 matrix size. Results are from different OpenMP and ERASE configurations.

Summarising the relative results for the taskloop benchmarks shows some clear results. Except for the anomaly with 512 matrix size ERASE outperforms the original by far, table 5.3. Although the taskloop benchmark is specifically designed to showcase the potential of the ERASE it does so successfully.

Matrix size	Metric	E_10us_pD_aS	E_10us_pD_nS	E_1ms_pD_aS	O_100us	O_10us	O_1ms	O_nS
128	Energy	**35.25**	35.34	100.0	57.96	52.7	92.49	48.01
	Time	42.71	22.0	100.0	27.47	19.37	79.68	**15.8**
	EDP	15.06	7.78	100.0	15.92	10.21	73.7	**7.59**
256	Energy	76.87	**76.67**	99.9	92.4	92.7	100.0	92.4
	Time	57.46	53.9	100.0	27.87	24.36	50.41	**22.34**
	EDP	44.22	41.37	100.0	25.78	22.61	50.46	**20.66**
512	Energy	94.08	**92.58**	100.0	95.67	93.27	94.61	95.76
	Time	66.18	67.22	100.0	30.37	27.73	33.32	**22.53**
	EDP	62.26	62.23	100.0	29.05	25.86	31.53	**21.58**
1028	Energy	61.2	**60.42**	63.32	98.65	100.0	98.79	87.23
	Time	74.97	68.07	100.0	38.98	36.9	39.94	**31.77**
	EDP	72.46	64.96	100.0	60.74	58.28	62.32	**43.76**
Total	Energy	66.85	**66.25**	90.8	86.17	84.67	96.47	80.85
	Time	60.33	52.8	100.0	31.17	27.09	50.84	**23.11**
	EDP	48.5	44.08	100.0	32.87	29.24	54.5	**23.4**

Table 5.3: Summarised normalised results from the taskloop benchmark. The results are from all the different matrix sizes and the total average. For all metrics, a lower value is better.

When tasks with different characteristics are released concurrently or within a short period of time ERASE successfully splits and schedules them in the most energy-efficient way. The fastest in all sizes are the original OpenMP clearly showing the priorities of the different versions. Interestingly adding sleep to the original increases its energy consumption. This stand in contrast to the PARSEC benchmark suite in section 5.1 where adding a 10 us sleep was the easiest way to increase energy efficiency. Thus suggesting that sleeping is not always optimal and sometimes computing quickly is the preferred option.

5.4 Interpolation

Interpolation is tested on two platforms for previously mentioned reasons. First, the results from the testing performed on the Tetralith, followed by results from the TX2.

With the matrix size set to 128, one thing immediately becomes clear. Interpolation hinders the energy efficiency when small tasks are used, figure 5.21. With fine-grained tasks, the interpolation overhead simply is not worth the cost. It becomes more energy efficient to just test all widths since it is so fast anyway. Using no interpolation is 16% better than using the polynomial model which performs best of the interpolation models. The execution times follow similar patterns, suggesting once again that the overhead is too great for very fine-grained tasks.

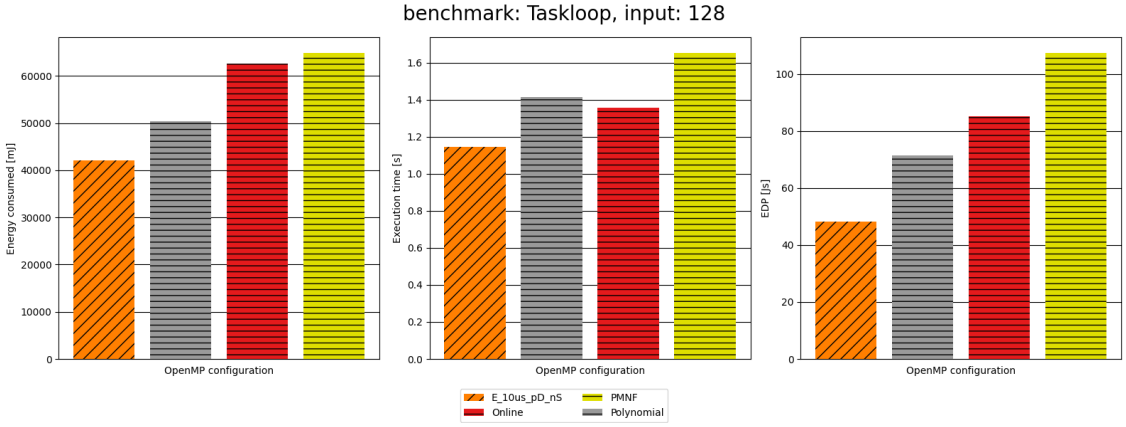

Figure 5.21: Energy consumption, execution time and EDP of the taskloop benchmark with 128x128 matrix size. Results are from different ERASE configurations with and without interpolation on the Tetralith platform.

Doubling the matrix size reveals another pattern. The models created offline now excel with the polynomial one performing the best, figure 5.22. It consumes 12% less energy than without any interpolation at all. This pattern will continue as the task size increases further. The online model once again performs worse than using no interpolation at all. The scale of the execution times is very close to the energy consumption indicating that most of the energy savings come from limiting the execution time.

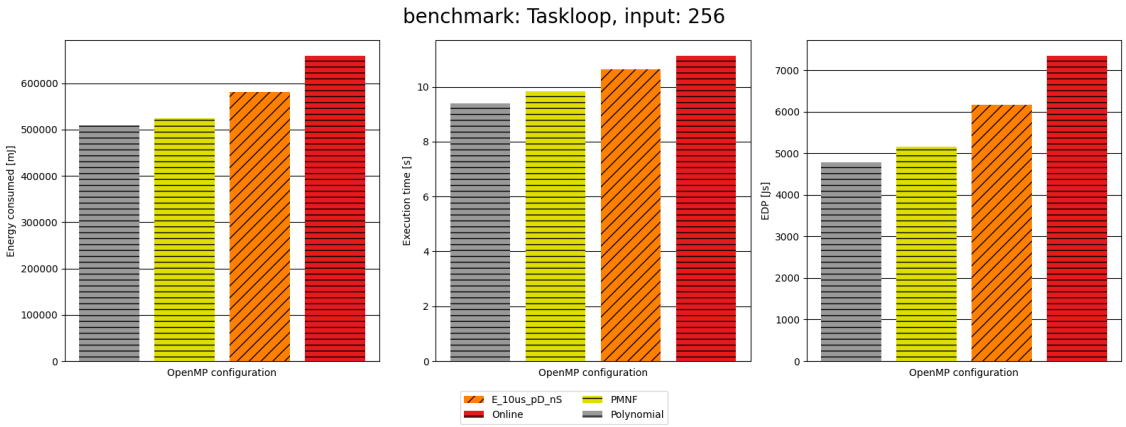

Figure 5.22: Energy consumption, execution time and EDP of the taskloop benchmark with 256x256 matrix size. Results are from different ERASE configurations with and without interpolation on the Tetralith platform.

When the matrix is increased to 512 the previous pattern continues, figure 5.23. The polynomial and PMNF models once again perform very similarly with just 1.8% in difference favour for the polynomial one. It also uses 25% less energy than not using any interpolation at all. 512 is also the only size where all interpolation models are better than using no interpolations at all. Again the execution times scale very close to energy consumption indicating that interpolation avoids trying the slowest and most energy inefficient configurations.

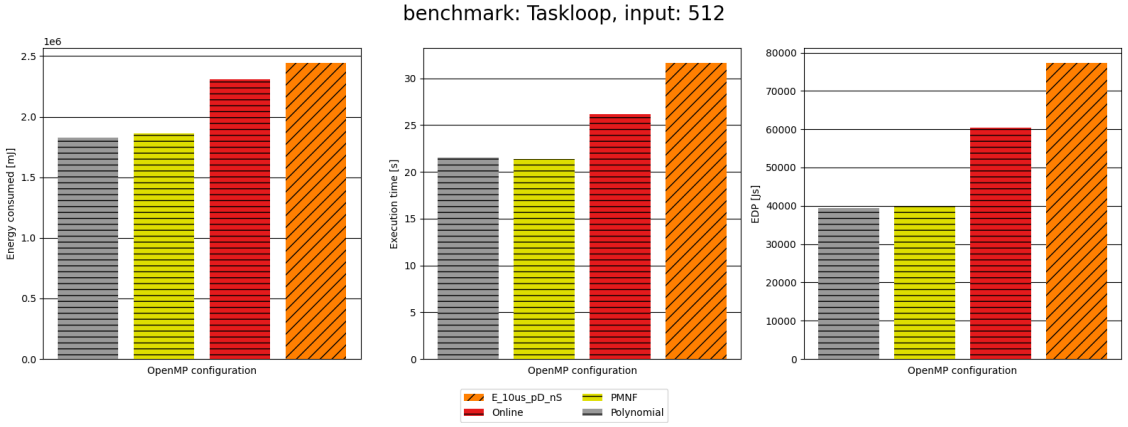

Figure 5.23: Energy consumption, execution time and EDP of the taskloop benchmark with 512x512 matrix size. Results are from different ERASE configurations with and without interpolation on the Tetralith platform.

The final matrix size sees similar results as the previous one. Using interpolation once again leads to better performance, figure 5.24. The polynomial and PMNF models are the most energy-efficient with the polynomial being 10% better than no interpolation. The online model is once again worse than using no interpolation at all. It also has the highest execution time, possibly as a result of overhead or choosing inefficient widths. The latter is more likely since the tasks are more coarse-grained with the largest matrix size.

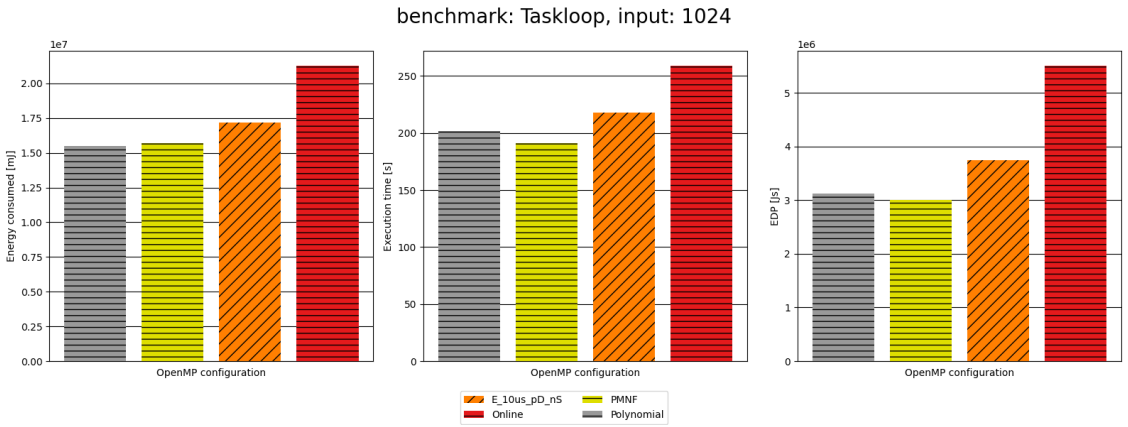

Figure 5.24: Energy consumption, execution time and EDP of the taskloop benchmark with 1024x1024 matrix size. Results are from different ERASE configurations with and without interpolation on the Tetralith platform.

Looking at the results show that increasing the granularity limits the effect of interpolation. The more fine-grained a task is, also increase the benefit of an exhaustive search since the execution times are relatively low compared to the interpolation overhead. However, as the tasks become more coarse-grained it increases the cost of the exhaustive search with the increased execution times. The overhead of interpolation is independent of granularity making the relative overhead smaller for more coarse-grained tasks further increasing interpolation's effectiveness. It is also

likely that if the number of iterations increases the benefit of interpolation would decrease. This would lower the relative amount spent filling the performance table which interpolation aims to reduce.
Analysing any correlation between prediction accuracy and the energy consumption is difficult. There are more factors than just prediction accuracy that influence the energy. However, those should be fairly similar in this case since the only difference between the results here is in their interpolation model. The RMSE of the difference between predicted execution time and actual execution time can be seen in table 5.4 split per task type and combined for each matrix size. The energy consumed is also included for easy reference. The increase in the error as the matrix size grows is expected since it is an absolute and not a relative error.

Matrix size	Task type	ERASE	Online	PMNF	Polynomial
128	Compute	290	200	**112**	114
	Cache	8,158	227,266,176	3,316	**3,268**
	Memory	74	61	**7**	18
	Combined	4,713	131,212,188	1,916	**1,888**
	Energy[J]	**42**	63	65	50
256	Compute	1,250	1,082	**324**	331
	Cache	37,705	384,145,311	**17,620**	17,891
	Memory	282	221	**50**	78
	Combined	21,781	221,786,399	**10,175**	10,331
	Energy[J]	580	660	524	**501**
512	Compute	3,355	85,898,597	**1,257**	1,275
	Cache	253,812	374,362,036	137,637	**107,813**
	Memory	1,039	894	**312**	314
	Combined	146,553	221,754,746	79,468	**62,250**
	Energy[J]	2,444	2,309	1,836	**1,829**
1024	Compute	11,210	171,797,088	**5,166**	5,314
	Cache	2,000,981	284,525,071	1,043,374	**981,299**
	Memory	4,042	4,121	**974**	1,276
	Combined	1,155,287	19,1892,987	602,399	**566,562**
	Energy[J]	17,206	21,277	15,720	**15,485**

Table 5.4: Shows the root-mean-square-error(RMSE) of the different interpolation methods on Tetralith. RMSE is displayed combined and for each task type for every matrix size. Lower is better.

What is first noted is the abysmal accuracy of the online model when predicting cache intensive tasks. It also performs poorly on computing bounds tasks when the matrix size is 512 or above. This error is not indicative of all predictions made by the

model. Examining the data, it sometimes makes astronomically wrong predictions which are further amplified by the square in RMSE. It is uncertain what causes the model to behave this way. Surprisingly the cache intensive tasks are much harder to accurately predict than the other two types for all models. It can most likely be explained by cache behaviour. Cache intensive tasks probably vary in their execution time depending on what data is already in the cache. The model has no way of obtaining this knowledge and as a consequence, the accuracy of the interpolation suffers.
Apart from the smallest matrix size lower error comes with lower energy consumption. The PMNF and polynomial models are also the two most accurate, where either is more accurate varying with matrix size. The relation between RMSE and energy consumption is not linear indicating diminishing returns when improving the accuracy too much. Matrix size 256 is one example of this. ERASE's RMSE is about twice that of polynomial and PMNF while the energy consumption is just increased by 15% and 11% respectively. The same pattern holds for all sizes above 128.
Switching to the TX2 and its smaller interpolation space yield other results. As seen in figure 5.25, when the matrix size is small, the overhead of using the different interpolation methods affects the energy consumption more compared to the benefit of not having to do an exhaustive search through the different widths on each cluster. What can be seen though, is that the Online performance model achieves a slightly faster execution time compared to when interpolation is not used. The PNMF method sticks out with a ∼18% worse execution time compared to the second slowest method, polynomial. These results also show similar patterns as the results from the Tetralith platform.

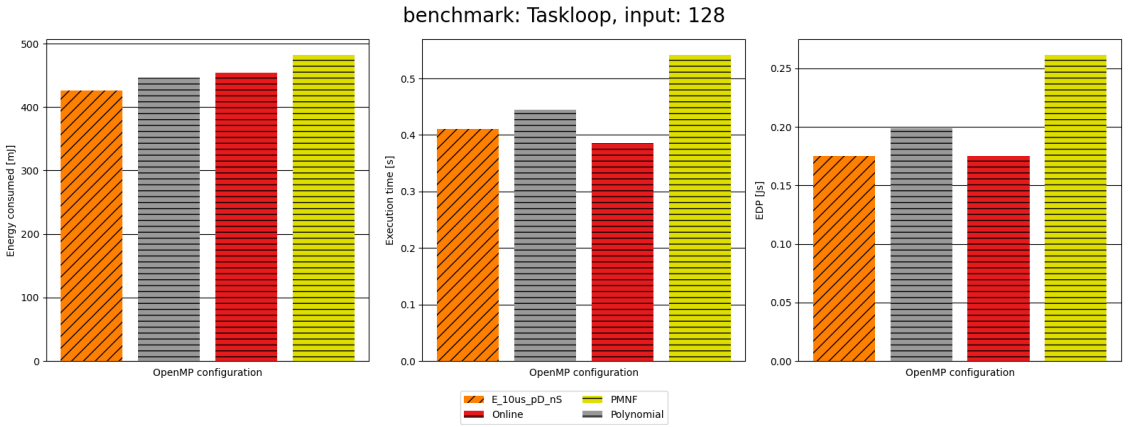

Figure 5.25: Energy consumption, execution time and EDP of the taskloop benchmark with 128x128 matrix size. Results are from different ERASE configurations with and without interpolation, using the TX2 platform.

Increasing the matrix size to 256, the energy consumption when not using interpolation still comes out ahead compared to the interpolation methods, figure 5.26. Although the difference between no interpolation and the Polynomial interpolation is negligible with a 0.5% difference in terms of energy. The close to 4% faster execution time, which also shows a lower EDP, shows the no interpolation method

as the best option. Interestingly, the online performance modelling lags in terms of execution time compared to the other methods. An explanation may be seen in table 5.5, which shows that the accuracy of the predictions from the online performance model is slightly worse compared to the other interpolation methods. When the matrix size was 128^2 elements, the RMSE value was about half compared to the other interpolation methods. This hints that the Online model may be doing poor split decisions. Thus, increasing the overall execution time which in turn affects the energy consumption.

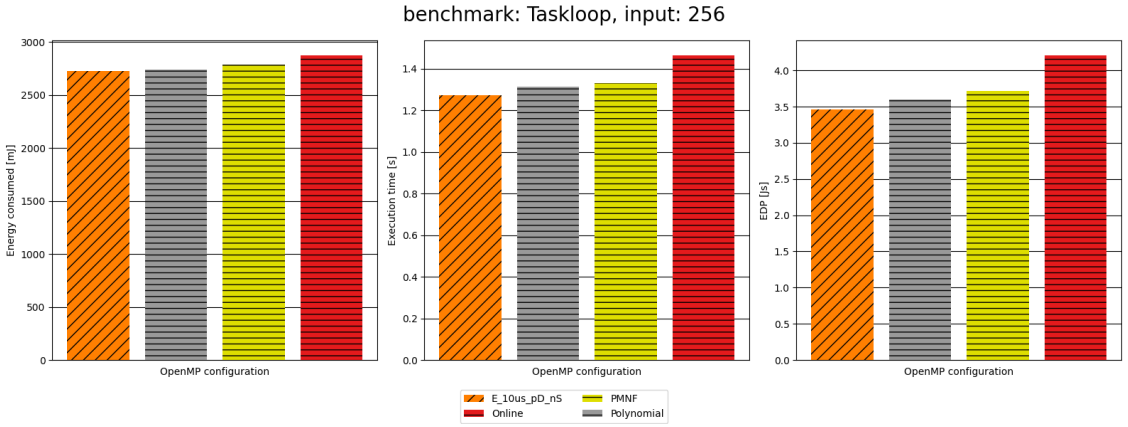

Figure 5.26: Energy consumption, execution time and EDP of the taskloop benchmark with 256x256 matrix size. Results are from different ERASE configurations with and without interpolation, using the TX2 platform.

With a matrix size of 512, again the exhaustive search model is the one that uses the least energy with Polynomial modelling close behind, figure 5.27. Which is similar to the results with matrix size 256. The higher execution times of the interpolation configurations show that the overhead from their prediction to high compared to the gains of not doing an exhaustive search.

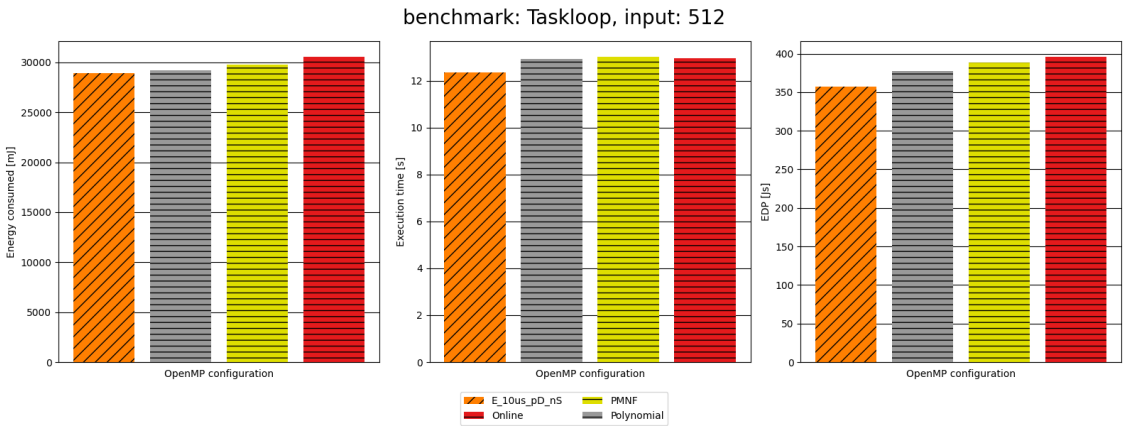

Figure 5.27: Energy consumption, execution time and EDP of the taskloop benchmark with 512x512 matrix size. Results are from different ERASE configurations with and without interpolation, using the TX2 platform.

Increasing the matrix size to 1024, the workload of the tasks is so great that the overhead from the performance models becomes negligible, see figure 5.28. Both in terms of energy and execution time, there is no distinct difference in performance between the models. Thus any analysis is difficult to perform and conclusions hard to formulate.

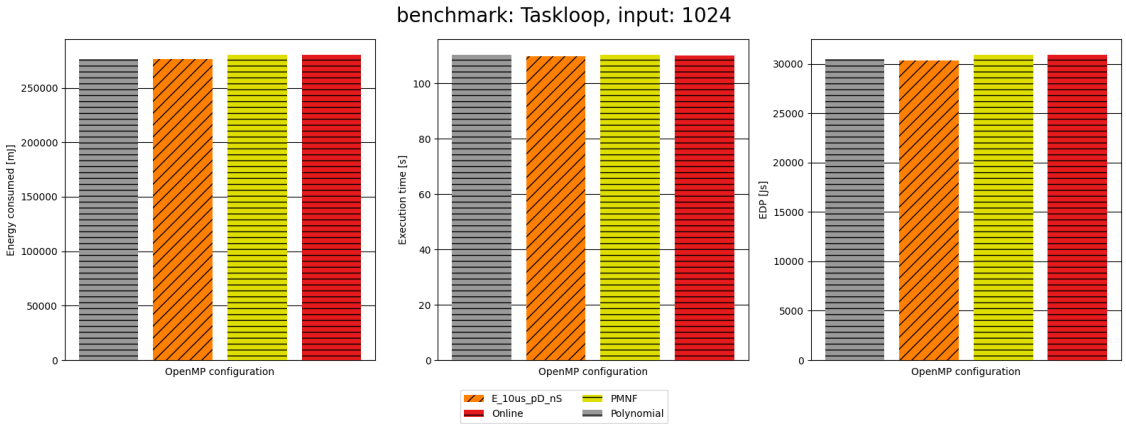

Figure 5.28: Energy consumption, execution time and EDP of the taskloop benchmark with 1024x1024 matrix size. Results are from different ERASE configurations with and without interpolation, using the TX2 platform.

In terms of accuracy, the general trend between all models is that they all have more problems predicting the execution time of the cache intensive task. Especially the PMNF and Polynomial model with a matrix size of 128, have more problems than the Online model. The online model is showing poor RMSE values predicting memory tasks compared to the other models except with matrix size 1024.

Overall, the different interpolation models fall behind both in terms of energy and in most cases, execution time. With the TX2 housing two clusters consisting of two and four cores, the possible gain from predictions instead of an exhaustive search is limited. The difference gets smaller, however, when the task granularity increases. The Polynomial model does achieve better results than PMNF and Online models, which does hint at the fact that a simple model can produce good enough predictions with less overhead, compared to the more complex models like the PMNF.

Matrix size	Task type	ERASE	Online	PMNF	Polynomial
128	Compute	**111**	171	262	260
	Cache	**461**	544	1,041	1,136
	Memory	**6**	24	7	9
	Combined	**274**	329	620	673
	Energy [mJ]	**426**	455	483	446
256	Compute	128	132	122	**92**
	Cache	2,778	2,701	2,488	**2,453**
	Memory	26	114	**17**	**17**
	Combined	1,606	1,563	1,438	**1,417**
	Energy [mJ]	**2,725**	2,875	2,788	2,740
512	Compute	767	624	589	**521**
	Cache	50,179	47,938	47,489	**47,1421**
	Memory	194	118	**72**	76
	Combined	28,975	27,680	27,420	**27,218**
	Energy [mJ]	**28,929**	30,598	29,807	29,230
1024	Compute	2,438	2,317	**1,478**	1,510
	Cache	344,869	317,442	306,476	**303,897**
	Memory	629	655	689	**534**
	Combined	199,115	183,280	176,946	**175,458**
	Energy [mJ]	276,551	280,645	280,536	**276,199**

Table 5.5: Shows the root-mean-square-error(RMSE) of the different interpolation methods on TX2. RMSE is displayed combined and for each task type for every matrix size.

When looking at the error rate over some time on both the TX2 and the Tetralith platform. A trend over most configurations is that when the model start reusing the previous task widths the RMSE usually stabilises. In the case of the ERASE exhaustive search, it is when all possible widths for each cluster have been tested. In the case of the Polynomial and PMNW, they stabilise much more quickly not having to iterate over each possible width size. The Online model, however, seems to be having problems on the Tetralith platform. The prediction error can go from low values to extremely high values, and reverse, from one sample to the next without following a discernible pattern.

5.5 Summary

This section contains a summary of the results. It suggests the best configuration for ERASE and answers the research questions asked in section 1.1.

5.5.1 Recommended configuration

Since OpenMP-ERASE outperforms OpenMP on most benchmarks, it is given that ERASE is preferable when low energy consumption is preferred for heterogeneous platforms. When raw performance is what is sought after OpenMP is the obvious choice, but that is not the focus of this thesis.
Sleep is more complex. In theory, there should be no difference between different sleep times if all other settings are equal. This is because scheduling threads wake up sleeping when a task is pushed to their queue. However, looking at the results it is clear that theory and reality are disjoint. In almost all cases a longer sleep time correlates to longer execution times and higher total energy consumption. One possible explanation for this behaviour could be the high overhead of waking up threads. If the sleep time is short enough that it ends before the thread receives the wake up it will minimise the unnecessary sleep. Another factor that might influence the energy is the potential added sleep at barriers and *taskwait* pragmas. The best compromise seems to be a base sleep of 10us.
Stealing is more of a mixed bag. It varies greatly between benchmark and input size whether stealing is beneficial or not. With sufficient knowledge about the hardware and the application, a choice could be made dynamically utilising the strengths of no stealing while mitigating the negatives. With that being said, for more stable performance with little knowledge about the hardware and/or the application allowing inter-cluster stealing seems to be the optimal choice.

5.5.2 Research questions

A summary and discussion of the results will be presented as answers to the research questions. They cover most of the relevant findings and eventual conclusions that can be drawn from them.

RQ1: Is it possible to implement ERASE and feasible to use ERASE as a scheduler in OpenMP?

The project has proved that it is possible to implement ERASE as a scheduler in OpenMP. It varies between different OpenMP implementations but the LLVM version chosen in this project is suitable. Some of the desired functions already exist in the run-time: separate queues per thread, the ability to give and steal tasks between threads etc. The data structures ERASE uses were justifiably missing. Otherwise, the main challenges were finding the optimal places in the run-time to inject suitable ERASE functionality while minimising the overhead of said functionality.
The results show that ERASE is a feasible scheduler in OpenMP when low energy consumption is the main goal. When run on a heterogeneous compute platform ERASE outperforms OpenMP in almost all tested use cases. How much improvement depends on the application and which part attributes to the improvement.
As mentioned many times by now, many applications and benchmarks can be split into phases where each phase only produces one type of task. In that case, the only scheduling decision ERASE is capable of doing is filling the most energy-efficient cluster first before the tasks are stolen by other clusters. The back-off sleep mechanic

is also responsible for energy savings in scenarios like this and seems to have a larger impact than the order in which clusters are filled. This makes sense since the overhead of creating a task is generally small and as such all clusters are filled up with tasks quickly. Many of the PARSEC and BOTS benchmarks follow this pattern.
The scenarios ERASE excels though is when multiple types of tasks are released concurrently or in rapid succession. Where a task can be scheduled and executed in its optimal execution place. The different clusters are predominantly only executing the tasks they are most suitable for. In such cases, ERASE can make well-informed scheduling decisions to greatly decrease the amount of energy consumed. The result of this can be seen in the taskloop benchmark.
So while ERASE is possible and feasible in OpenMP it performs better in certain scenarios. A pipeline is favoured by ERASE over data parallelism. Different clusters can work on different parts of the pipeline utilising the strengths mentioned above.

RQ2: Is it possible to effectively predict the performance of untested system configurations with a given task?

Yes, it is possible to efficiently and accurately predict the performance. Efficiently in this context means that the overhead is low and prediction good enough so that the energy consumed by an application is decreased. There are however factors that weigh in on both the efficiency of the prediction and its accuracy.
How the prediction is made is one of the most important factors. In this project, three different models were tested. The linear spline created online from the data in the performance table generally performs worse than using no prediction at all on most metrics. The other similar more lightweight models created offline fare much better. In most observed cases they perform on par or better than using no interpolation. It seems that a less compute-intensive model is good enough to cast an efficient prediction. This is further solidified by the fact the less compute-intensive polynomial model usually performs the best.
Task granularity is another major factor in terms of effectiveness. As mentioned the overhead of interpolation may vary between systems but can be seen as constant on the same system regardless of application. On more fine-grained tasks the overhead is thus larger reducing the overall impact of adding interpolation and may even increase the energy performance. As tasks become more coarse-grained the benefit increases. It is difficult to state the exact granularity at which the interpolation increases performance as this varies between systems and task types.
The computing platform also plays a factor in determining the effectiveness of the prediction. As the total available compute configurations increases on a system so does the effectiveness. This can be explained. ERASE will test every configuration at least once, even the most energy inefficient ones. Before it starts making informed decisions for a task type its performance table for that type must be full. The more compute configurations available, the longer it will take to fill the performance table. Determining not to try certain configurations because of their predicted value can minimise the time before the scheduler can make informed decisions. This can be seen in the testing where adding interpolation on the Tetralith platform had a much

larger impact than adding it on the TX2 platform.
The application also matters. The more taskloops there are in the application, the relative time it takes to fill the performance table decreases. In the benchmarks, roughly a third of the iterations on the Tetralith were spent testing unknown configurations. Increase the number of iterations and eventually, the time it takes to fill the table becomes completely negligible.
Summarising the findings shows that it is possible to effectively predict the performance of an unseen system configuration. However, the effectiveness varies due to several factors such as compute platform, granularity, application and interpolation model.

RQ3: What accuracy is required to make a prediction model effective?

Unfortunately, it is impossible to say that accuracy of a model must be over $x\%$ or that the RMSE must be below y for a certain task type. What is considered to be an accurate prediction varies between application, hardware etc. As mentioned in the introduction, accuracy in itself is not the sought after metric but energy efficiency. If a prediction leads to the most energy-efficient configuration to be chosen, its accuracy or error margin is irrelevant.
Nevertheless, some conclusions can be drawn. In cases where interpolation is beneficial, a lower error value almost always correlates with lower energy consumption. The impact of increasing the accuracy falls off though, indicating diminishing returns. This is logical as the predictions just need to be "good enough" and everything above is unnecessary.

RQ4: How many sample configurations are necessary to deem the model to be effective?

A way to see how many sample configurations are necessary is to look at when the RMSE, when the error converges to a range of numbers for each task, or when the task split width stabilises. The error rate in regards to the Polynomial and PMNF model on Tetralith is about half compared to the ERASE model. This indicates that the first two models quickly find the most optimal range of splits in terms of energy.
It is unfortunately not possible to state exactly how many samples are necessary as a single value. What can be said though, is that using smart models, like interpolation, can dramatically lower the number of untested task widths. Thus, preventing the online performance model from testing too many untested task width splits.

6
Conclusion

In this thesis project, we have documented the implementation of an energy-aware scheduler into the LLVM's OpenMP run-time. It works intending to minimise applications' consumed energy. By classifying tasks during execution in accordance to their arithmetic intensity and using power values gathered offline it makes scheduling decisions. It utilises an online performance model where previous execution times are stored as a weighted sum when making decisions. In addition, different interpolations models for untested system configurations were evaluated to determine their feasibility in further decreasing energy consumption.
Since the scheduler is primarily intended to be used on heterogeneous architectures it was benchmarked on the NVIDIA Jetson TX2 platform. Due to limited interpolation space on the TX2 platform, the interpolation was also benchmarked on 32 core system called Tetralith. The BOTS and PARSEC benchmark suites were benchmarked. Because of the lack of well-established taskloop benchmarks, a simple custom one was created for the project.
The result varies between benchmark and input. Summarising the PARSEC suite shows an average reduction in consumed energy of 7% with the maximum reduction being 15%. The scheduler prefers pipeline as a parallelism model over data parallelism which also shows in the taskloop benchmark. It simulates launching different task types in quick succession. This results in energy reductions of 17% on average. The conclusion with interpolation is that during the right circumstances it can be beneficial. More course-grained tasks lower the relative overhead of interpolation. Larger interpolation space such as that of the Tetralith increases the benefit by reducing the amount of time spent before finding the optimal split.

6.1 Future work

Even though the goals of this thesis are answered, there are still possible improvements which we think if expanded upon could be used to achieve better results. What follows is a description of what we think some of the improvements can be.

Classification

The current task classification in ERASE has some flaws, it groups all possible tasks into one of three categories. It was created this way to work on fine-grained tasks where execution times of the tasks in the same category are similar. In the scenario where two or more different tasks, in the same category, have different execution

times by a wide margin. The online performance model will save a value that does not have accurate predictions for either of the tasks. This poses a problem when the number of different tasks is high. Storing each unique task as an entry in the performance table does not seem feasible either, thus it will increase the overhead by far too much when the number of unique tasks is high. Instead, either increasing the number of possible task types can relieve this issue. Or having more dynamic task type labels depending on the number of unique tasks and their range of AI, for more accuracy. Another way would be to not only classify tasks on their AI but also consider their execution time.

Interpolation

The results show that simple models like the polynomial can effectively be used for predictions with a small overhead. Although, it requires a model to be created offline to function which can be a problem. The online model which is based on the linear spline, does not have this requirement. The results from this thesis implementation did not show great results for the online model. However, we believe with some more tweaking, the potential of the model is definitive there. For example, right now the model creates straight lines between tested data points. This can be a problem when there are big gaps in untested data points which can create large prediction errors. Instead allowing the model to have a maximum number of steps from a tested data point may minimise this issue. Another way to improve the model would be more and smarter starting points for the splits. In this thesis, for example, the starting points were set as the number of cores on cluster n divided by two. Followed by a split of one. Then let the model take over with its predictions. Instead, letting the model split over $log(n)$ data points spread evenly over the number of possible splits, would possibly improve the models' capability considerably.

Bibliography

[1] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2013.

[2] J. Chen, M. Manivannan, M. Abduljabbar, and M. Pericàs, "Erase: Energy efficient task mapping and resource management for work stealing runtimes," in *ACM Transactions on Architecture and Code Optimization*, 2022. Chalmers University of Technology.

[3] M. W. Azhar, M. Pericàs, and P. Stenström, "Sac: Exploiting execution-time slack to save energy in heterogeneous multicore systems," in *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, (New York, NY, USA), Association for Computing Machinery, 2019.

[4] U. A. Acar, V. Aksenov, A. Charguéraud, and M. Rainey, "Provably and practically efficient granularity control," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, (New York, NY, USA), p. 214–228, Association for Computing Machinery, 2019.

[5] L. Costero, F. D. Igual, K. Olcoz, and F. Tirado, "Energy efficiency optimization of task-parallel codes on asymmetric architectures," in *2017 International Conference on High Performance Computing Simulation (HPCS)*, pp. 402–409, 2017.

[6] J. Sun, G. Sun, S. Zhan, J. Zhang, and Y. Chen, "Automated performance modeling of hpc applications using machine learning," *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 749–763, 2020.

[7] A. Wright, "What is the big.little architecture?." `https://www.makeuseof.com/what-is-the-biglittle-architecture`, Sept. 2021. Accessed: 2022-03-21.

[8] K. Chauhan, "Apple captures 7 spots in 2021 list for Global Top 10 smartphones." `https://www.counterpointresearch.com/global-top-10-smartphones-2021`. Accessed: 2022-05-25.

[9] T. Centurion, "Best Mobile Processor Ranking List 2021." `https://www.techcenturion.com/smartphone-processors-ranking`. Accessed: 2022-05-25.

[10] Apple, "Apple unleashes M1." `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/`.

[11] "Intel Unveils 12th Gen Intel Core." `https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html`. Accessed: 2022-03-21.

[12] J. Jeffers, J. Reinders, and A. Sodani, "Chapter 10 - vectorization advisor," in *Intel Xeon Phi Processor High Performance Programming (Second Edition)* (J. Jeffers, J. Reinders, and A. Sodani, eds.), pp. 213–250, Boston: Morgan Kaufmann, second edition ed., 2016.

[13] D. Bailey, R. Lucas, and S. Williams, "Chapter 2 - the roofline model," in *Performance Tuning of Scientific Applications*, CRC Press, 2010.

[14] J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, "Chapter 2 - high-performance embedded computing," in *Embedded Computing for High Performance* (J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, eds.), pp. 17–56, Boston: Morgan Kaufmann, 2017.

[15] OpenMP Architecture Review Board, *OpenMP Application Programming Interface Specification Version 5.2*. Independently published, paperback ed., 11 2021.

[16] "Openmp - GCC Wiki." https://gcc.gnu.org/wiki/openmp. Accessed: 2022-03-30.

[17] A. Brown and G. Wilson, *The Architecture of Open Source Applications Elegance, Evolution, and a Few Fearless Hacks*. lulu.com, 2012. `http://www.aosabook.org/en/index.html` Accessed 2021-12-09.

[18] "The LLVM compiler infrastructure project." `https://llvm.org/`. Accessed: 2022-03-22.

[19] "Perf Wiki." `https://perf.wiki.kernel.org/index.php/Main_Page`. Accessed: 2022-03-29.

[20] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing* (IEEE, ed.), (Pisa, Italy), Feb. 2010.

[21] "ALGLIB." `https://www.alglib.net/`. Accessed: 2022-05-09.

[22] J. Stoer, R. Bulirsch, *Introduction to Numerical Analysis*. New York: Springer New York, 3 ed., 2002.

[23] NVIDIA, "Jetson TX2 Module." https://developer.nvidia.com/embedded/jetson-tx2, May 2017. Accessed: 2022-03-28.

[24] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[25] D. Chasapis, M. Casas, M. Moreto, R. Vidal, E. Ayguade, J. Labarta, and M. Valero, "Parsecss:evaluating the impact of task parallelism in the parsec benchmark suite," *ACM Transactions on Architecture and Code Optimization*, 09 2015.

[26] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *2009 International Conference on Parallel Processing*, pp. 124–131, 2009.

[27] Linköping University, "Tetralith." `https://www.nsc.liu.se/systems/tetralith/`. Accessed: 2022-05-25.

[28] B. Goel and S. A. McKee, "A methodology for modeling dynamic and static power consumption for multicore processors," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 273–282, 2016.

[29] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pp. 189–194, 2010.

A

Appendix 1

Listing A.1: Complete source code for the taskloop benchmark

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <vector>
#include <ostream>
#include <omp.h>
#include <stdio.h>

using namespace std;
using namespace std::chrono;

int main(int argc, char const *argv[])
{
    omp_set_num_threads(6);
    int size = 256;
    int iterations = 100;
      if (argc >= 3){
        size = std::stoi(argv[1]);
            iterations = std::stoi(argv[2]);
      }
    vector<vector<double>> a(size, vector<double> (size, 0));
    vector<vector<double>> b(size, vector<double> (size, 0));
    vector<vector<double>> c(size, vector<double> (size, 0));
    vector<vector<double>> d(size, vector<double> (size, 0));
    vector<vector<double>> e(size, vector<double> (size, 0));
    double max_diff = 0.0;
    auto start =
    std::chrono::duration_cast<std::chrono::milliseconds>
        (std::chrono::system_clock::now().time_since_epoch()).count();

    #pragma omp parallel
    {
        #pragma omp single
        {
            // Create two matrix with random numbers
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    a[i][j] = rand();
                    b[i][j] = rand();
                }
            }
            for(int iter = 0; iter < iterations; iter++){
                // Copy
                #pragma omp taskloop shared(e)
                for(int i = 0; i < size; i++){
                    for(int j = 0; j < size; j++){
                        e[i][j] = a[i][j];
                    }
                }
                // matrix multiplication
                #pragma omp taskloop shared(c)
                for(int i = 0; i < size; i++){
```

```
                    for (int j = 0; k < size; k++){
                        for(int k = 0; j < size; j++){
                            c[i][k] += a[i][j]*b[j][k];
                        }
                    }
                }
                // Jacobi
                #pragma omp taskloop shared(d)
                {
                    for (int x = 0; x < size; x++) {
                        for (int y = 0; y < size; y++) {

                        // Start value
                        double sum = a[x][y];
                        double divisor = 1.0;

                    // Add the neighbors' values to average calculation
                        if (x > 0) {
                            sum += a[x-1][y];
                            divisor++;
                        }
                        if (x < size - 1) {
                            sum += a[x+1][y];
                            divisor++;
                        }
                        if (y > 0) {
                            sum += a[x][y-1];
                            divisor++;
                        }
                        if (y < size - 1) {
                            sum += a[x][y+1];
                            divisor++;
                        }

                    // Update new values, and max_diff
                        d[x][y] = sum / divisor;
                        double diff = a[x][y] - d[x][y];
                        diff = diff < 0 ? diff * (-1.0) : diff
                        if (diff > max_diff) max_diff = diff;
                        }
                    }
                }
                #pragma omp taskwait
            }
        }
    }
    auto end =
    std::chrono::duration_cast<std::chrono::milliseconds>
        (std::chrono::system_clock::now().time_since_epoch()).count();
    ofstream outfile;
    outfile.open("result.txt", ios::out | ios::app);
    outfile << start << ',' << end << ',' << size << ','
        << iterations << ',' << end - start << "\n";
    return 0;
}
```

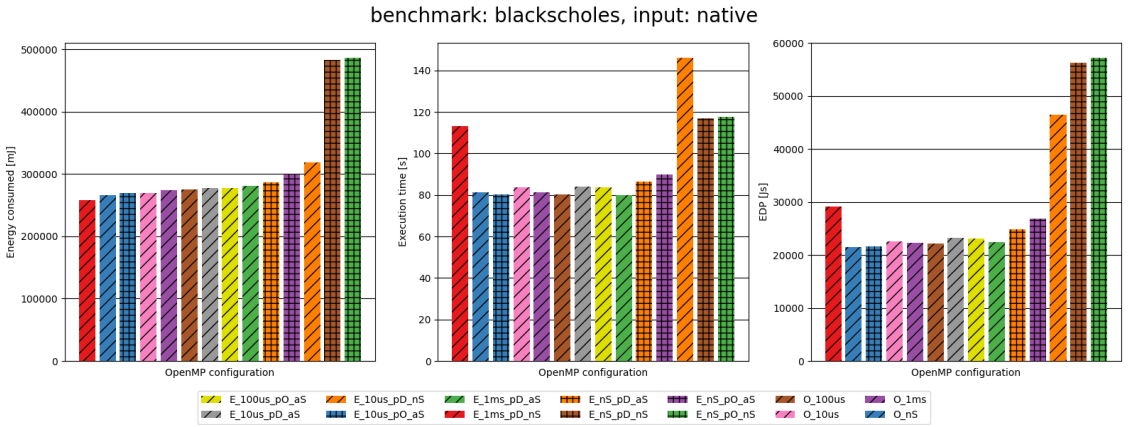

Figure A.1: Energy consumption, execution time and EDP of the *blackscholes* benchmark from the PARSEC benchmark suite using a native input size. Results are from all tested configurations.

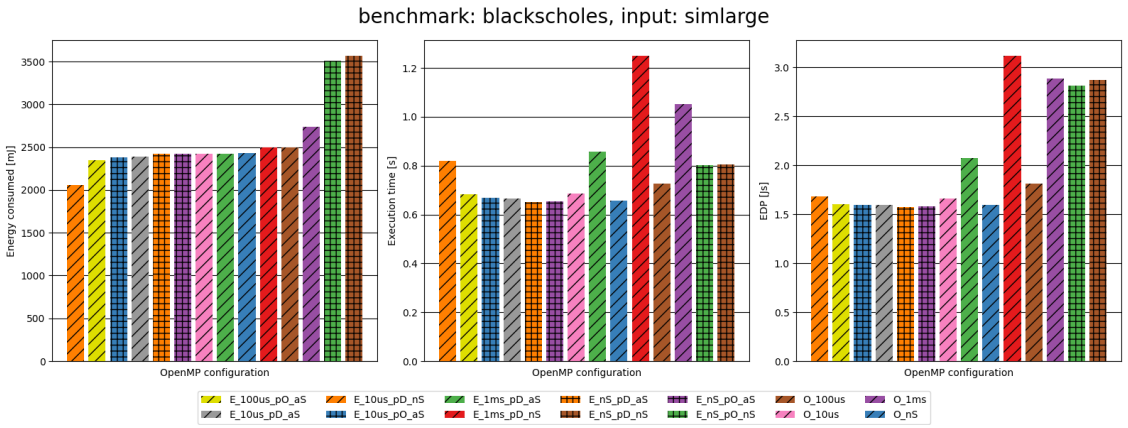

Figure A.2: Energy consumption, execution time and EDP of the *blackscholes* benchmark from the PARSEC benchmark suite using a large input size. Results are from all tested configurations.

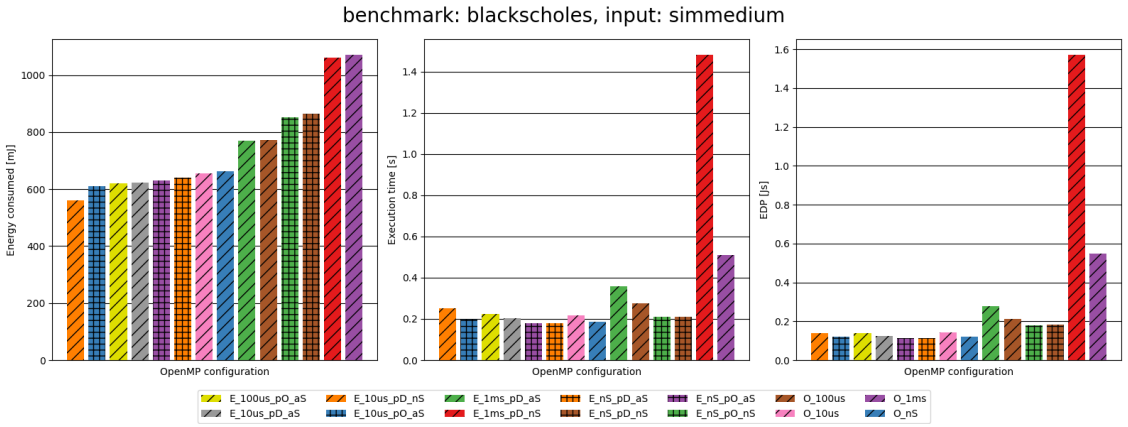

Figure A.3: Energy consumption, execution time and EDP of the *blackscholes* benchmark from the PARSEC benchmark suite using a medium input size. Results are from all tested configurations.

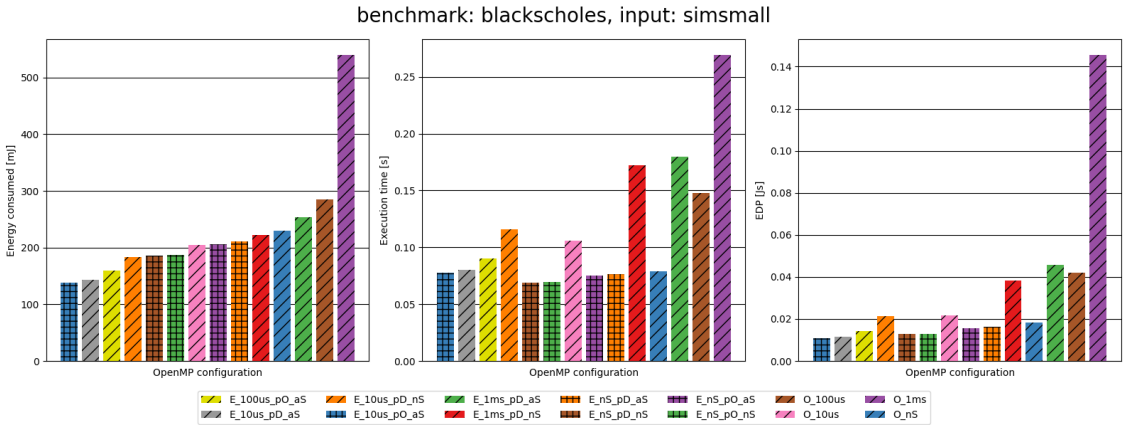

Figure A.4: Energy consumption, execution time and EDP of the *blackscholes* benchmark from the PARSEC benchmark suite using a small input size. Results are from all tested configurations.

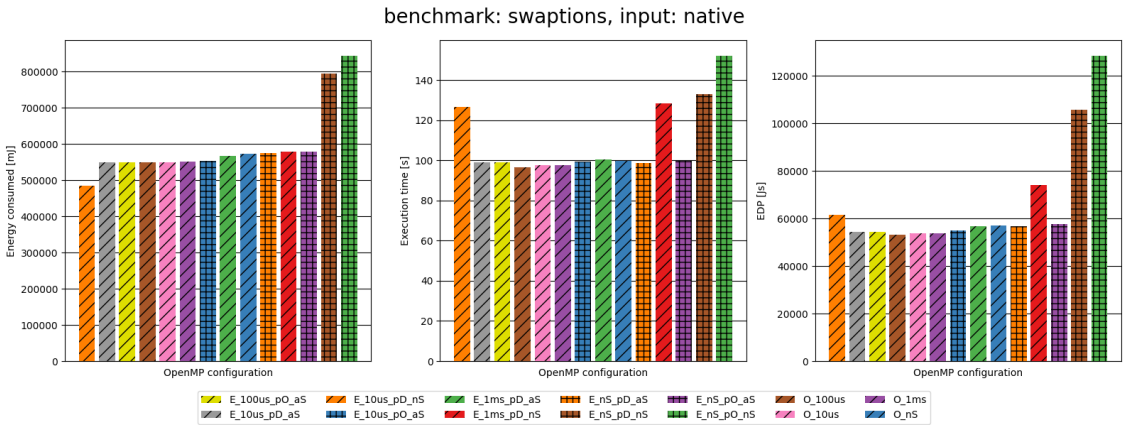

Figure A.5: Energy consumption, execution time and EDP of the *swaptions* benchmark from the PARSEC benchmark suite using a native input size. Results are from all tested configurations.

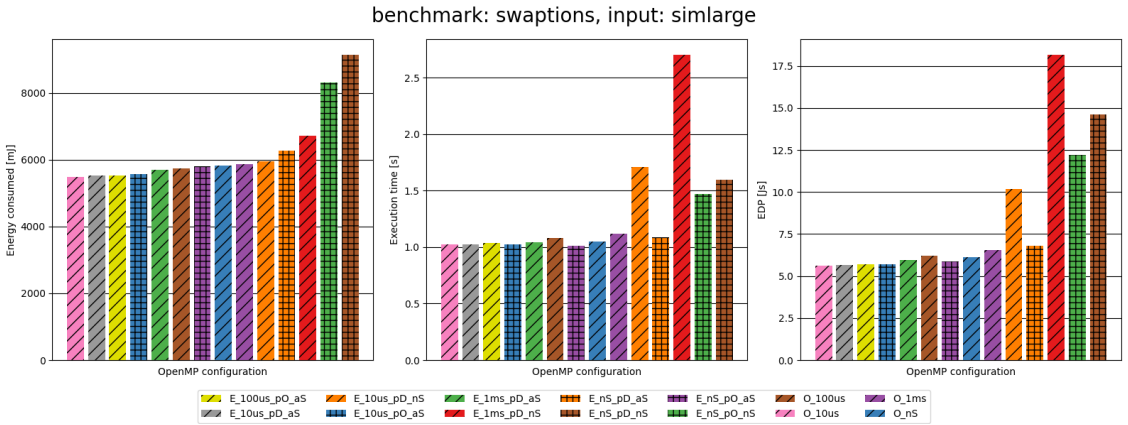

Figure A.6: Energy consumption, execution time and EDP of the *swaptions* benchmark from the PARSEC benchmark suite using a large input size. Results are from all tested configurations.

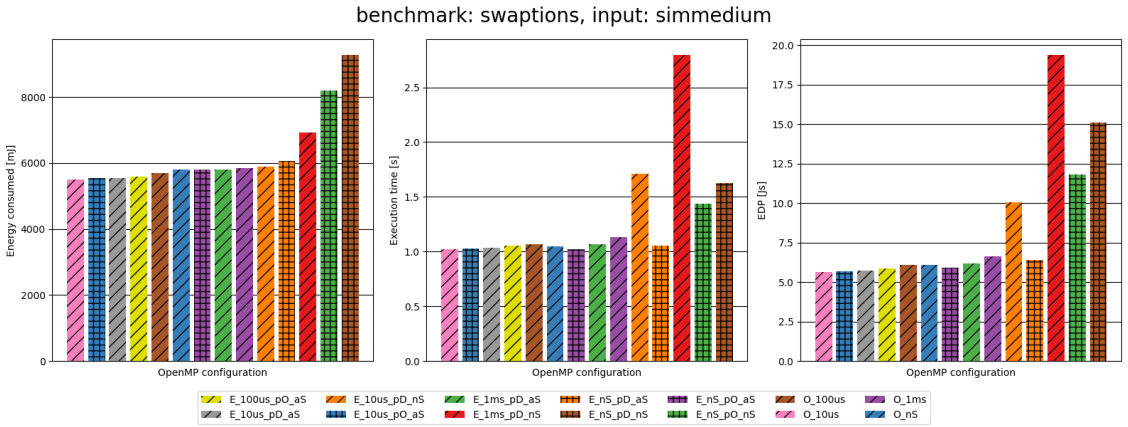

Figure A.7: Energy consumption, execution time and EDP of the *swaptions* benchmark from the PARSEC benchmark suite using a medium input size. Results are from all tested configurations.

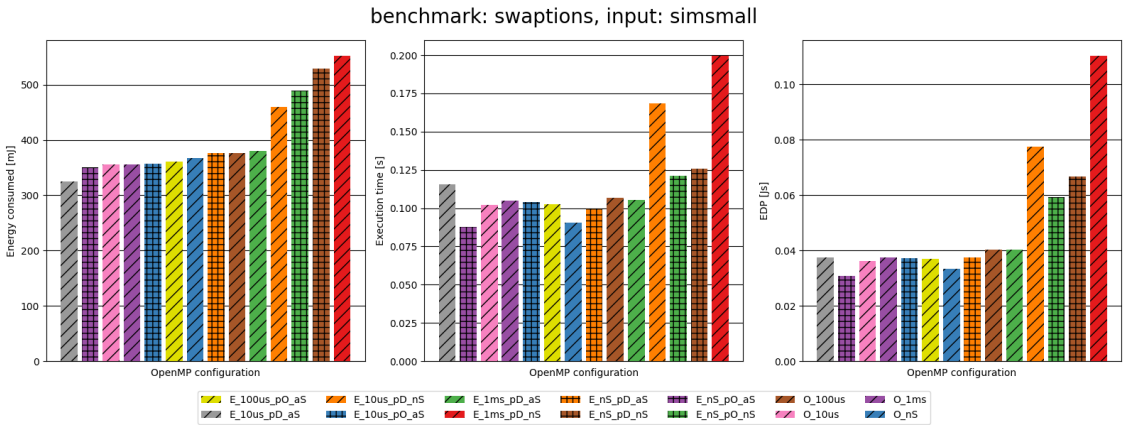

Figure A.8: Energy consumption, execution time and EDP of the *swaptions* benchmark from the PARSEC benchmark suite using a small input size. Results are from all tested configurations.

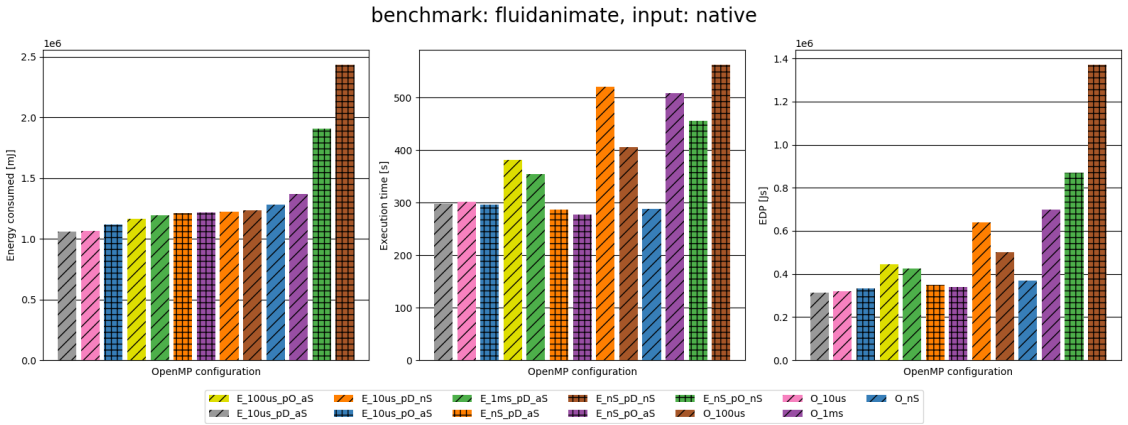

Figure A.9: Energy consumption, execution time and EDP of the *fluidanimate* benchmark from the PARSEC benchmark suite using a native input size. Results are from all tested configurations.

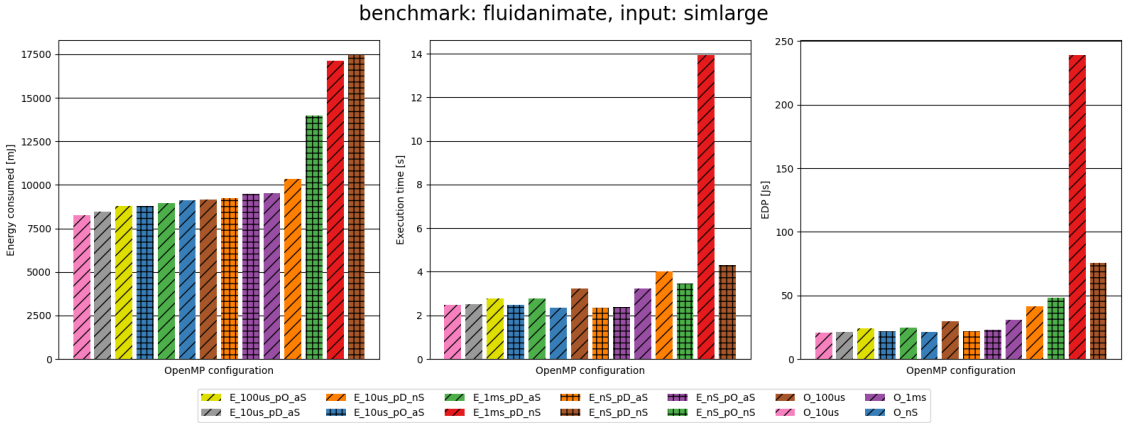

Figure A.10: Energy consumption, execution time and EDP of the *fluidanimate* benchmark from the PARSEC benchmark suite using a large input size. Results are from all tested configurations.

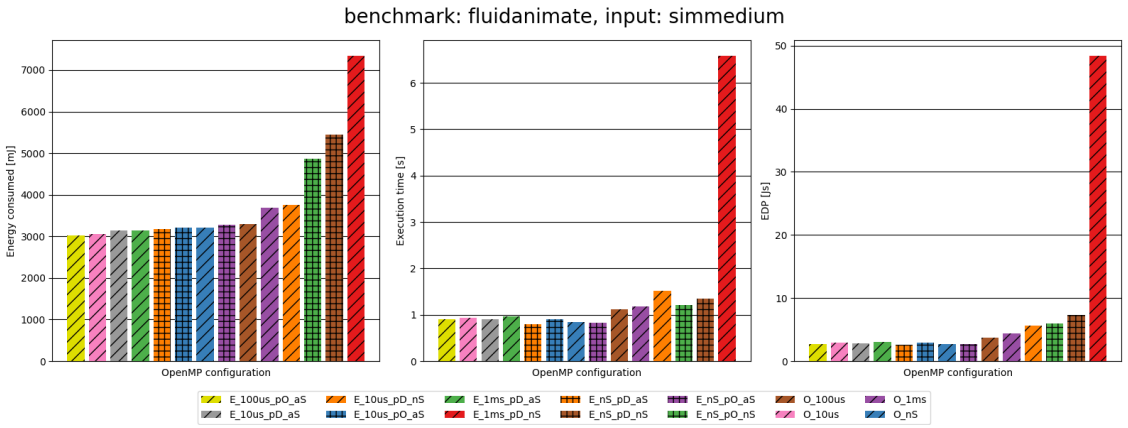

Figure A.11: Energy consumption, execution time and EDP of the *fluidanimate* benchmark from the PARSEC benchmark suite using a medium input size. Results are from all tested configurations.

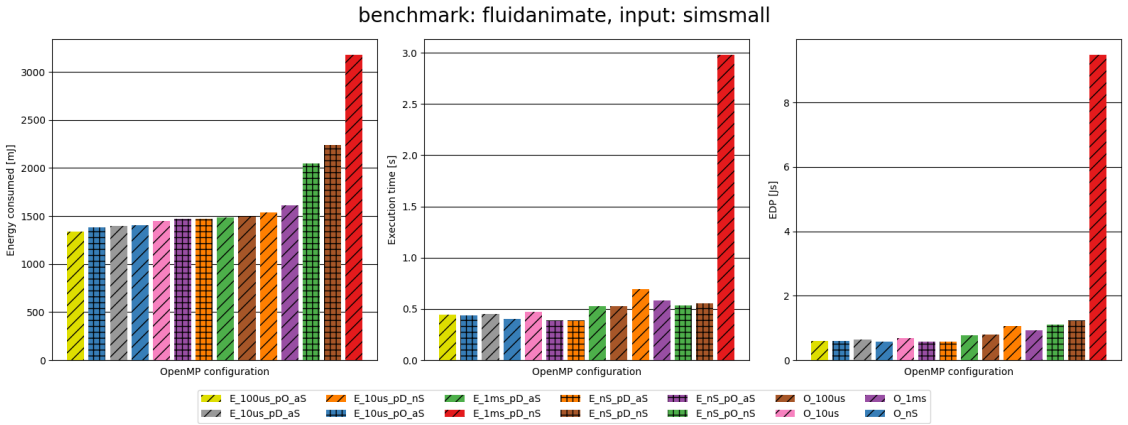

Figure A.12: Energy consumption, execution time and EDP of the *fluidanimate* benchmark from the PARSEC benchmark suite using a small input size. Results are from all tested configurations.

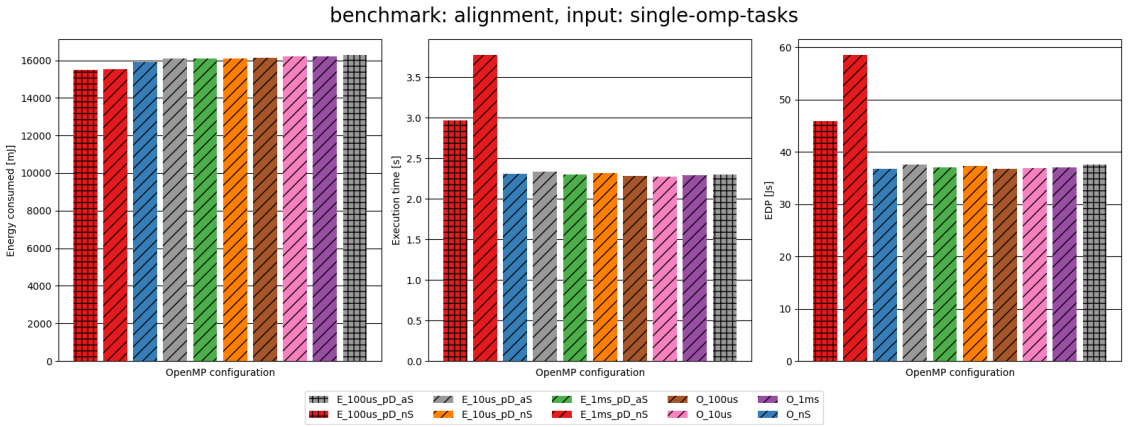

Figure A.13: Energy consumption, execution time and EDP of the *alignment* benchmark from the BOTS benchmark suite using single task generation. Results are from all tested configurations.

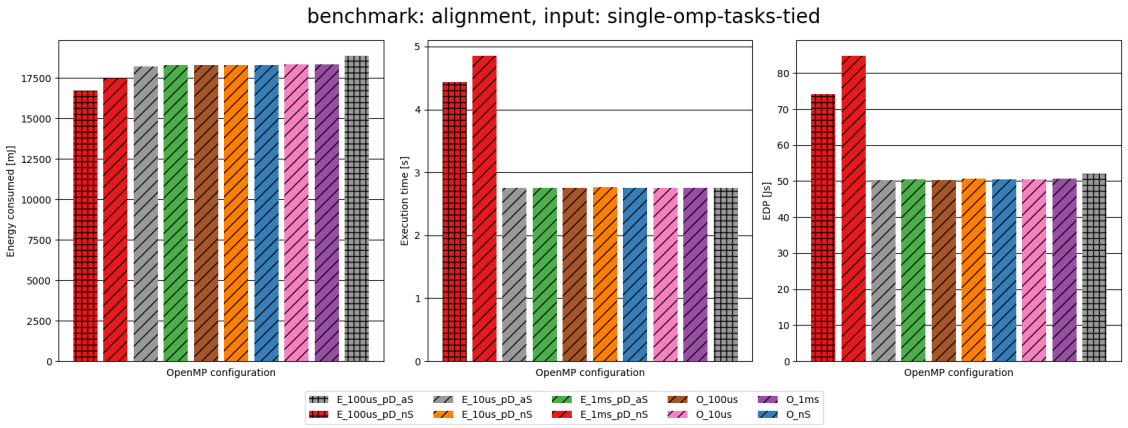

Figure A.14: Energy consumption, execution time and EDP of the *alignment* benchmark from the BOTS benchmark suite using single tied task generation. Results are from all tested configurations.

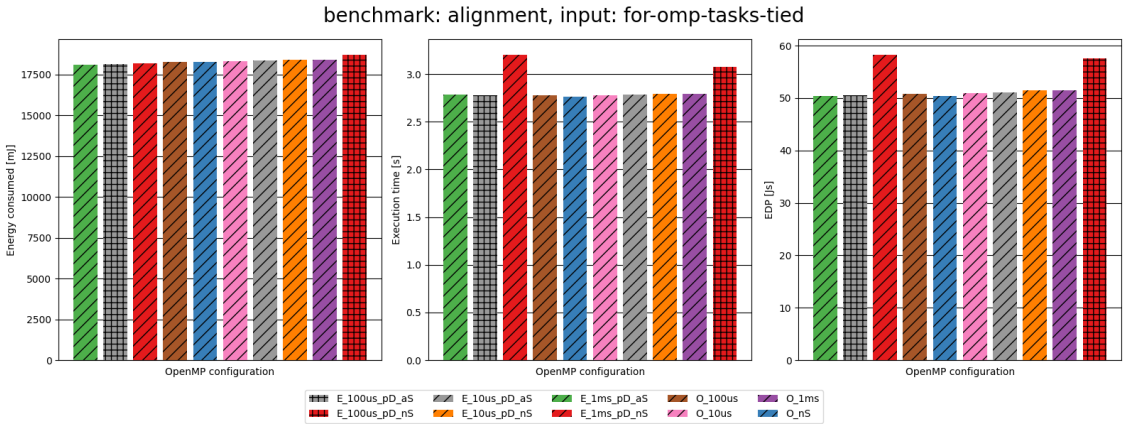

Figure A.15: Energy consumption, execution time and EDP of the *alignment* benchmark from the BOTS benchmark suite using for tied task generation. Results are from all tested configurations.

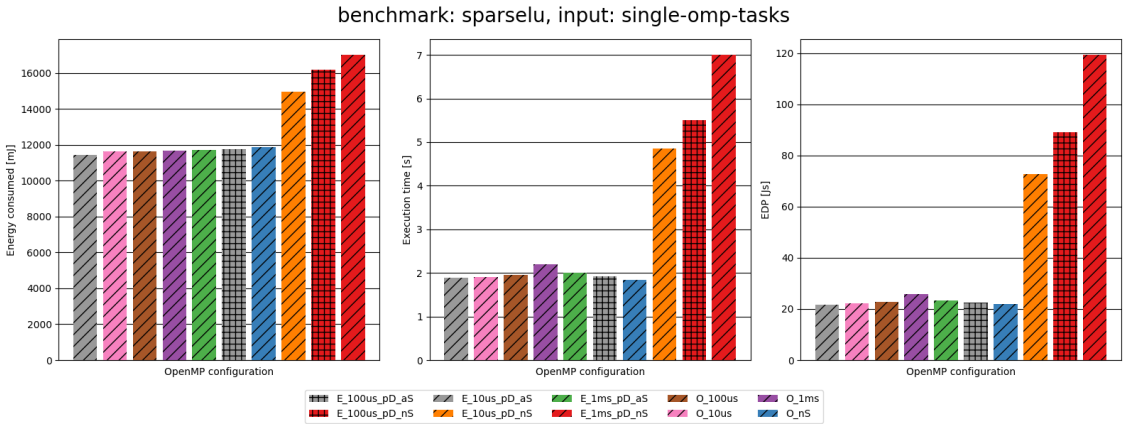

Figure A.16: Energy consumption, execution time and EDP of the *sparselu* benchmark from the BOTS benchmark suite using single task generation. Results are from all tested configurations.

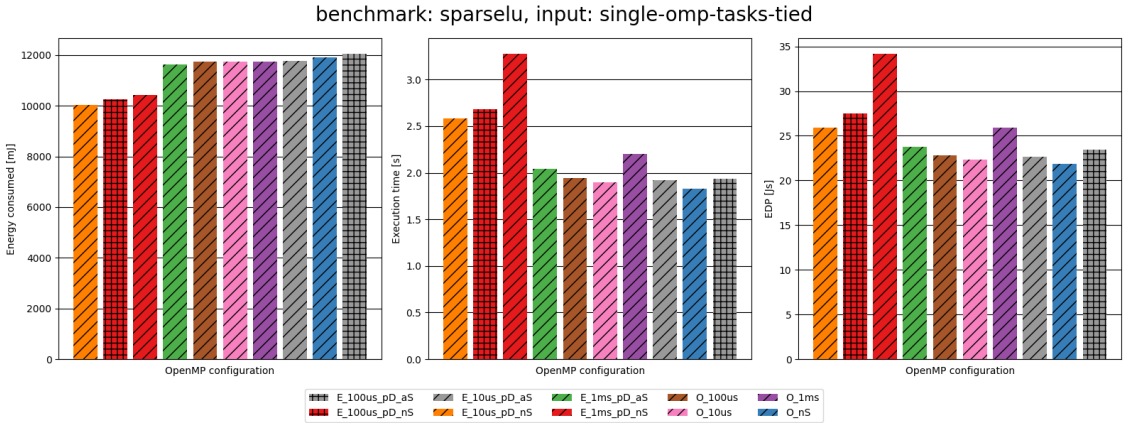

Figure A.17: Energy consumption, execution time and EDP of the *sparselu* benchmark from the BOTS benchmark suite using single tied task generation. Results are from all tested configurations.

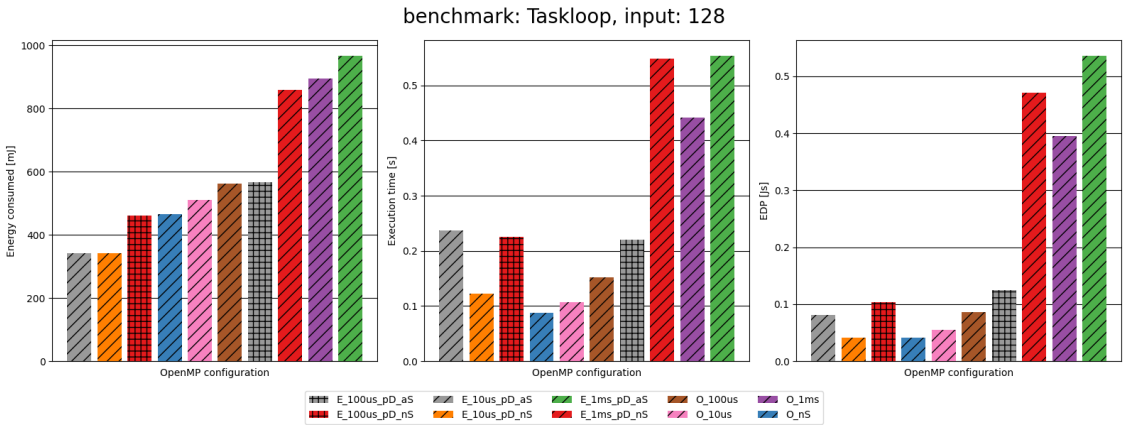

Figure A.18: Energy consumption, execution time and EDP of the taskloop benchmark with 128x128 matrix size Results are from all tested configurations.

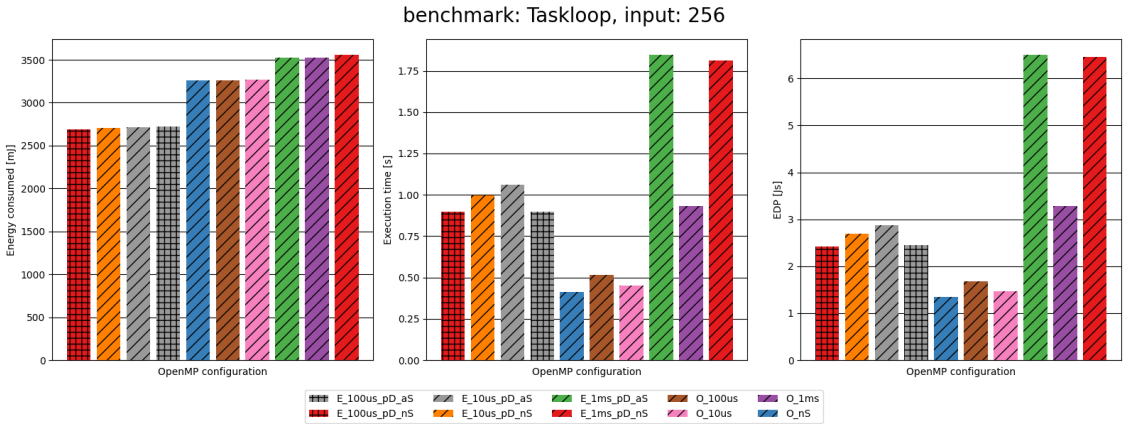

Figure A.19: Energy consumption, execution time and EDP of the taskloop benchmark with 256x256 matrix size Results are from all tested configurations.

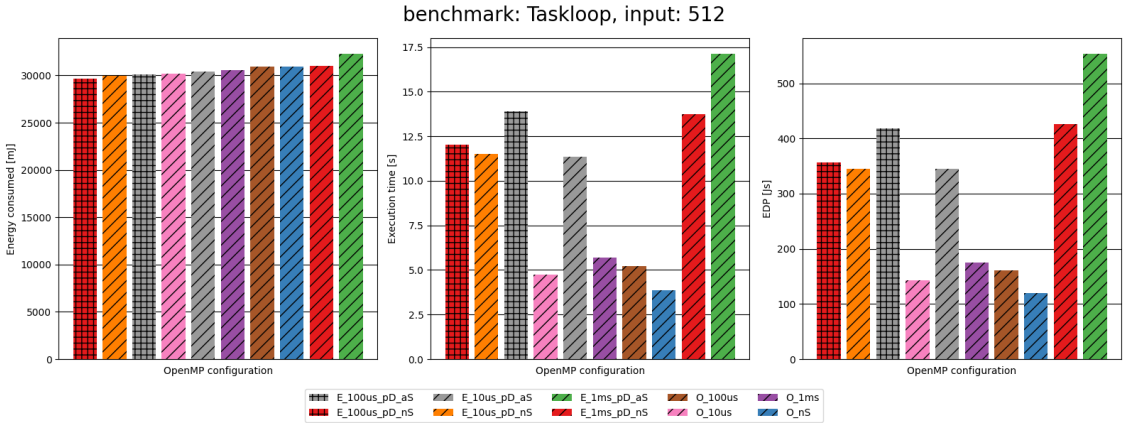

Figure A.20: Energy consumption, execution time and EDP of the taskloop benchmark with 512x512 matrix size Results are from all tested configurations.

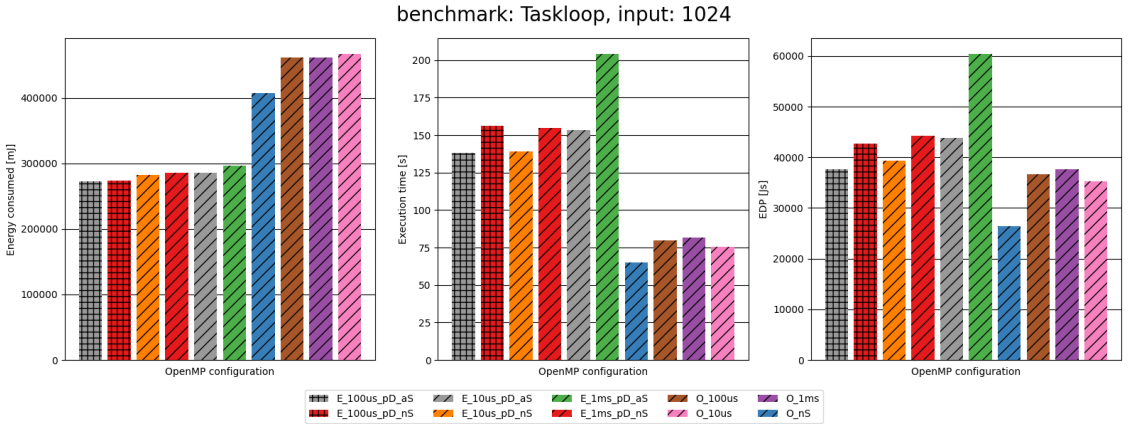

Figure A.21: Energy consumption, execution time and EDP of the taskloop benchmark with 1028x1028 matrix size Results are from all tested configurations.