Enhanced Scheduling Techniques through Lightweight Monitoring for OmpSs-2

By

ANTONI NAVARRO MUÑOZ
(antoni.navarro@bsc.es)

Advisors

VICENÇ BELTRAN QUEROL[†] (vbeltran@bsc.es)

EDUARD AYGUADÉ PARRA[†‡] (eduard@ac.upc.edu)

[‡] Computer Architecture Department (DAC)

[†] Barcelona Supercomputing Center (BSC-CNS)

Final Master Thesis
High Performance Computing Specialty
MASTER IN INNOVATION AND RESEARCH IN INFORMATICS

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BARCELONATECH

BARCELONA, 28 JANUARY 2019

Acknowledgments

First and foremost I want to thank both my advisors, Vicenç Beltran and Eduard Ayguadé, for the opportunity they gave me of working side by side on various challenging yet interesting projects, specifically this one. Not only do I want to express my gratitude towards them for the opportunity, but also for the constant support and dedication, which made everything easier.

I would also like to thank Josep M. Perez, who offered tireless help throughout my work, all the colleagues from the BSC Programming Models group, and everyone I met either at BSC or UPC with whom I had the pleasure to work with for many afternoons and late-nights, occasionally sharing beers (especially Ferran, Lluís, Gerard, all the Oriols, Diego, Miquel, all the Paus and Marcs, Adrià, Ying... sorry if I missed someone!)

I cannot end this note without thanking my whole family and especially my parents and brother, who supported me throughout not only this thesis, but also all my life.

Last but not least, I need to mention both my life-long friend Kevin Sala and my girlfriend Maureen van Osch for their constant help but most importantly for being two of the most patient and kind persons in my life, since they carry the burden of putting up with me not only through academic but also everyday life.

Abstract

Modern applications become larger and more complex with each passing day. To name a few, weather forecasting or particle simulation applications are examples of how applications may have significant differences in features, constraints, and limitations.

Users of these applications are usually faced with the task of optimizing or tuning them. To put the amount of work in perspective, experienced developers might take weeks to merely understand one of these large applications. Once experienced with it, they can begin tuning executions. Many aspects have to be taken into consideration, however. Input sizes, recursive depths, system workloads, internal status, or the underlying architecture onto which applications are running, are just a few.

Most runtimes supply users with functionalities to tune their executions. Optimizing through runtime-wise parameters by trial and error prevails as the default methodology in the field of parallel applications. Users often try different configurations until they stumble upon one which seems to yield the most performance. This is the so-called 'optimal' configuration. These are static as they cannot take the current state of the system into consideration. They also prove to be nonportable, as a slight change in any of the aspects mentioned before might yield undesirable negative effects in performance. When this occurs, users must try different configurations yet again.

In this project, we propose the addition of several monitoring modules to runtimes. These modules introduce precise information about the units of work these libraries must schedule. The extension of these libraries allows for accurate real-time predictions for present and future executions. Such predictions can be used to obtain better scheduling of future units of work automatically and, therefore, improve the overall performance of executions or the utilization of resources. All this, while being unnoticed by users, thus giving more power to the runtimes.

Through the evaluation provided, we demonstrate the precision of our predictions and how they can be used to optimize resource utilization among others. We integrate all the extensions mentioned above on an already existing runtime maintaining the vision of the integration being capable on any similar runtime or library.

Table of Contents

List of Figures

List of Tables

List of Code Samples

List of Algorithms

1 | Introduction

Current trends show that modern applications have become and keep becoming larger. As they grow, so does their computational intensity and complexity. A few good examples of relatively large applications or benchmarks are FTDock [2], which performs docking of biomolecules to make predictions, and WRF [3], which models weather forecasting. These and more applications use underlying computations that are intensive enough as to be considered benchmarks on their own. Some examples of these computations include fast Fourier transforms and LU decompositions like the Cholesky decomposition, first introduced in [4].

To fulfill these growing needs, applications are often ported to utilize other libraries or runtimes that offload work by fully exploiting parallelism. These ports aim to exploit intra-node parallelism [5–8], inter-node parallelism [9, 10], or both [11]. Most commonly, distributed-memory programming models such as MPI [12] are used to exploit inter-node parallelism. Other shared-memory programming models such as OpenMP [13] or OmpSs [14] are used for the intra-node counterpart.

The recent increase of interest in artificial intelligence brings more use cases for the aforementioned libraries. To name an example, at the lowest level of machine learning are various matrix operations that are computationally intensive. Recently, there have been successful attempts [15, 16] to introduce parallelism in AI-related fields.

Users of these applications, meaning users who work with and keep developing them, need their applications to execute as fast as possible. This is one of the most desired needs. However, resource utilization is also an essential factor. Using the least amount of resources as possible opens the door for more executions to be done in parallel. Taking into consideration both of these needs when scheduling is not an easy task – many aspects have to be taken into account:

- **Limitations or boundaries** – Is the application memory or compute bound? Does it perform a lot of Input/Output operations?
- **Input sizes** – The input size will have a direct effect while scheduling, although it is linked with other parameters.
- **Units of work** – Together with the input size they define in a rough way the available parallelism in an application. It can be the block size, the recursion depth or simply the number of units of work.

- **Current state of the system** – Possibly one of the most important aspects and frequently neglected due to not having runtime-related information in execution time from the application side.
- **Underlying architecture** – Also being an important aspect, the underlying architecture is an aspect that should be taken into account. As applications are portable, so should be their most significant executions. An alteration of the architecture from an execution to another could drastically decrease performance depending on the features of both.

One of the above stands out from the rest. Applications have no internal knowledge about the runtime. By knowledge we understand information such as the size of the execution queues or buffers. Runtimes, such as the ones that implement the OpenMP or OmpSs programming models, give means by which users can tune their applications. These are often in the form of environment variables, compilation parameters or even inner-code parameters such as block sizes in some of the directives used. Users can use these to specify a particular scheduling policy, granularity, etc.

Frequently, users will use the tools mentioned above to tune applications. Depending on the architecture where they execute their applications, users will have a set of parameters and variables to run with, that they have previously needed to study. This proves to be a tedious task, as these parameters will most likely have to be considered again when modifying any of the aforementioned aspects. As dull as it might be, users continue to manually tune their applications since they search for simplicity and maximum performance.

Being aware of the system's status is not a task that pertains to application developers, so in this project we aim to solve these issues focusing on what users search for – simplicity and performance. It must stand out that we believe the approach of *power to the runtime* to be the appropriate one for these cases, since it eliminates the need to study each application parameter in each architecture. Our proposal focuses on the extension of runtimes with monitoring modules capable of obtaining information in a lightweight manner without introducing distortion in executions. These monitoring modules would be capable of obtaining precise predictions about the incoming workload in systems to better schedule units of work or to exploit resource utilization better.

1.1 Contributions

The primary aim of this project is to extend programming models with tools by which they can easily identify units of work and their relative computational weight. This is so that runtime libraries based on these programming models can use the tools to enhance their scheduling policies.

Specifically, we aim to extend the OmpSs-2 programming model to serve as an example. Nonetheless, our approach should be replicable and adaptable for any kind of parallel programming model. A summary of the contributions of this thesis is listed next:

- Find non-intrusive information-gathering mechanisms.
- Extend the OmpSs-2 programming model with these mechanisms to easily identify or categorize units of work based on their features.
- Integrate all the extensions into a complete monitoring infrastructure that collects metrics about every element in the programming model.
- Use the aforementioned infrastructure to create tools that, in a lightweight manner, obtain information about the units of work. In OmpSs-2 these units of work are tasks.
- Compute predictions of any kind that might be worth using by the internal schedulers, based on metrics obtained by the monitoring infrastructure. These can be in the form of timing predictions, workload predictions, task's features-based predictions or resource utilization predictions.
- Generate the ability to provide real-time information about the metrics to external modules or libraries.
- Use predictions and metrics to enhance scheduling policies within the OmpSs-2 programming model.

With the contributions listed above, users would benefit from not having to tune their applications manually. They would also benefit from custom-shaped scheduling from the runtime side without having to study their applications, thus obtaining what they want more straightforwardly and transparently.

1.2 Document Structure

This document starts with the current chapter (1), which introduces the topic of our contributions and the aim of the project. It follows with state of the art in the related work chapter (2). After a brief of the related work in this field, chapters 3, 4, and 5 describe the literature of the environment of this project, as well as the OmpSs-2 programming model. After explaining all the elements that take part in this project, chapter 6 presents a complete description of all the tools and libraries used, as well as the methodology that was followed while working on this project.

Once the whole environment and tools have been described, chapters 7, 8, and 9 start describing the thesis itself. In chapter 7 we introduce the modules with which OmpSs-2 is extended. In chapter 8, the algorithms used to create predictions are discussed, and, in chapter 9, we showcase all the implemented enhancements to current scheduling techniques and policies. Once every aspect of the thesis has been discussed, chapter 10 follows by presenting an extensive evaluation of our contributions, properly introducing every application and architecture that has been tested since our contributions are independent of both.

To wrap it up, in chapter 11 we discuss the conclusions extracted from this thesis and finally in chapter 12 we briefly comment about all the options regarding future work after this thesis.

2 | Related Work

Finding scheduling policies that automatically adapt to both runtime and application needs is not a new topic. Many researchers have worked on finding optimal and adaptive techniques to solve the aforementioned issues. When it comes to automatic granularity control, Thoman et al. [17] presented an approach for recursive OpenMP [13] applications. Their approach uses a compiler to generate several versions of a task each with increasing granularity. At runtime, a specific version of the tasks is chosen taking into account the size of internal queues. Similarly, Cong et al. [18] propose a strategy to control the granularity of parallel tasks, adapting them to the size of internal queues. This last proposal is thought for an open-source runtime, the X10 Work Stealing framework.

The aforementioned techniques propose adaptive task granularities. These techniques might be well suited for some applications. However, they do not contemplate all applications and runtimes. Another approach to control runtime workloads is limiting task creation. When a new task is about to be created, the runtime takes a decision. If the task meets certain criteria, it is spawned. On the other hand, if it does not meet such standards, it is inlined in the caller or parent task. By inlining them, their parent tasks are made coarser, and the overhead of creating too fine-grained tasks disappears. This technique was first thought as an extension for OpenMP in work conducted by Duran et al. [19], through introducing a `final` clause in the programming language to force coalescing of excessively fine-grained tasks. This proposal was accepted and entered in OpenMP. However, it is a manual way of tuning applications as users must study the best parameter for this clause in all applications where they want to use it.

From some of the same authors came another proposal [20], where they add an automatic cut-off technique which imitates the behavior of the final clause. This proposal does not rely on programmers. Mohr et al. [21] were the first ones to introduce this concept in their article. Their concept, however, spawns tasks only if the system's workload is scarce, that is, resources are idle. This approach has flaws, since it only takes into account the current status of the system. Thus, very fine-grained units of work might be created if the system has idle resources, and if the overhead to create such units of work is too demanding, it might not be worth doing.

In a previous work [22], we also introduced the concept of limiting task creation. Even though our work focused on recursive applications, the heuristics we presented could be adapted to any application. Our work improved previous proposals by introducing the idea of a new `cost` clause, which only required from users to specify the relative computational cost of the units of work. Because of this, it cannot be defined as manual tuning. Our proposal concluded that through this clause, predictions of timing were precise and these helped automatically limiting granularity. Our previous work also related to a proposal introduced by Duran et al. [20], since both use internal profiling to obtain information at runtime. Other works such as the one conducted by Aharoni [23] propose algorithms that automatically discover worthwhile parallelism and insufficient parallelism in applications, without focusing on units of work.

Duran et al. explored adaptive scheduling techniques yet further in another proposal [24], where they derived at runtime the best scheduling policies for each parallel loop in applications. This technique is similarly based on information also gathered at runtime.

On another note, predicting resource utilization for better managing remains a visited topic as well, as it has not been out of the scope, but rather recently visited for virtual machine environments. Recent works [25, 26] discuss heuristics and statistical methods such as linear regression to predict the utilization of resources to optimize their usage for cloud computing platforms and virtual machines within servers.

Predicting resource usage is not trivial. It is often linked to timing predictions for workloads, which is what this project aims to achieve. Works such as the one conducted by Sadjadi et al. [27] aim to predict the execution time of long-lasting applications, at runtime. Other works like Sadeka et al. [28] propose using timing predictions to predict resource usage, thus linking both ideas. Their aim, however, is focused on cloud computing platforms.

A different approach presented by Qawasmeh et al. [29] suggests the use of machine learning and runtime APIs to both profile executions and try to cluster tasks. Their profiling APIs allow having at their disposal information about tasks by using hardware counters. With this information, they achieve to group and identify different types of tasks and then use scheduling policies to intertwine the execution of tasks with sufficient difference in features.

To the best of our knowledge, ours is the first approach that tries to not only combine but also improve and extend with new proposals most of the aforementioned works. Similarly to most of the articles, this thesis generically extends runtimes with monitoring modules to obtain information at runtime. Our approach, however, studies different methods to gather information to make it as lightweight as possible, to avoid the effect of overheads when profiling executions that shadow the potential enhancement of performance with our scheduling techniques.

We plan to use these profiling modules to make several different kinds of predictions – some of these related to execution time, some others related to resource usage, etc. Our approach also monitors the accuracy of predictions to have feedback for future predictions, which is something we believe none of the works mentioned above included. These predictions will be used to create several heuristics, some of them included in our previous work [22]. With these heuristics, we plan to have enhanced scheduling policies that also adapt automatically to any architecture or application tested. As a final remark, we believe to be the first ones to focus our work generically, not only on recursive applications.

As previously stated, we will exemplify our proposal in the context of the OmpSs-2 [30] programming model. However, our claims could be adapted to any task-based programming model.

3 | The OmpSs-2 Programming Model

OmpSs-2 [30] is the second generation of the OmpSs programming model. The name comes from the combination of names of two other programming models: **OpenMP** and **StarSs**. OmpSs takes its general ideas from the design aspects of these two models. It is a programming model formed by a set of directives and library routines. These, combined, allow developers to create concurrent applications with high-level programming languages such as C++ or Fortran.

With OmpSs, developers use annotations in codes to produce parallel versions of applications. This idea was based on OpenMP. These annotations do not have direct effects in programs. They allow an underlying compiler to generate extra code to enable parallel versions of the code to exist.

StarSs, or **Star S**uper**S**calar, is a family of programming models that also offer implicit parallelism through a set of compiler annotations. Some of the differences between StarSs and OpenMP are:

- StarSs uses a thread-pool execution mode. OpenMP on the other hand implements fork-join parallelism. A representation of both is shown in figure 3.1. The scheme in the upper part of the figure shows a fork-join model in which parallel regions are created and dismissed, and in which threads within the region execute tasks. The scheme below illustrates how threads poll tasks from a task queue and execute them, and how parallel regions are not created nor destroyed. Instead, a parallel region is implicit from beginning to end.

- StarSs includes features to target heterogeneous architectures through leveraging native kernels. OpenMP, however, targets accelerator support through the generation of direct compiler code.

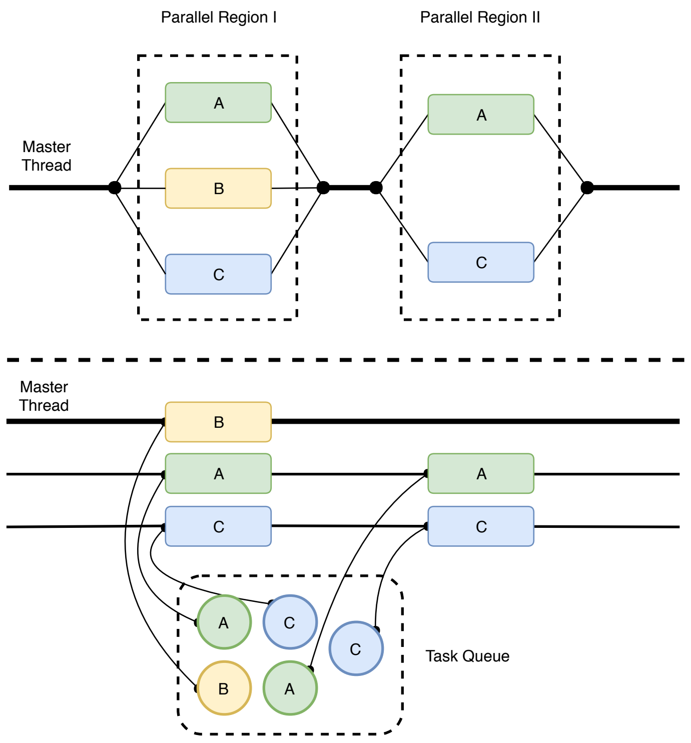


Fig. 3.1: Fork-join vs Thread-pool execution models

- StarSs offers asynchronous parallelism as the main mechanism of expressing parallelism. OpenMP only started implementing this feature since version 3.0.
- StarSs offers task synchronization through dependences as the main mechanism of expressing execution order. OpenMP only started including this mechanism since version 4.0.

When using both programming models, the differences between them are easy to grasp. In OpenMP, developers must first define regions of the code that are to be executed in parallel. After that, they express synchronization or work-sharing between the threads inside the parallel region. Once that is out of the way, developers might need to add directives to synchronize between different parallel regions, or even within them.

With StarSs, the process is more straightforward. StarSs simplifies parallelizing programs by implicitly defining parallelism from the beginning of the execution until the end. Developers need not control parallel regions or work sharing between threads. This is instead offloaded to the underlying runtime. Defining parallel code in StarSs is as simple as identifying units of work as tasks. A **task** is the minimum execution entity that can be managed independently by the runtime's scheduler. Tasks are units of work, or pieces of code, that can be executed in parallel. As aforementioned, synchronizing these units of work is much simpler. StarSs offers dependences for this purpose. **Dependences** allow expressing the correct order in which tasks must be executed to guarantee a proper program order. This, at the same time, allows exploiting resources more efficiently. This is because the underlying runtime can take any scheduling decision within the boundaries of the specified synchronization.

Rather than the misconception of being an extension of OpenMP, OmpSs tries to be the evolution that OpenMP needs to be able to target newer architectures. To do this, OmpSs takes the key design features from OpenMP and adds new ideas developed in the StarSs family. More about this is explained in section 3.1.

The reference implementation of OmpSs-2 is based on the Mercurium source-to-source compiler and the Nanos6 Runtime Library. Mercurium is a source-to-source compiler that provides support to transform high-level directives in parallel applications. More about Mercurium is explained in section 6.1.1.

The Nanos6 runtime provides services to manage the parallelism in user applications, including task creation, synchronization, and data handling, and provide support for heterogeneous architectures. As a big part of this project is carried in this runtime, it is explained further in chapter 4.

3.1 Influencing OpenMP since 2008

Many ideas from the OmpSs and StarSs programming models have been introduced into OpenMP. Figure 3.2 summarizes these contributions.

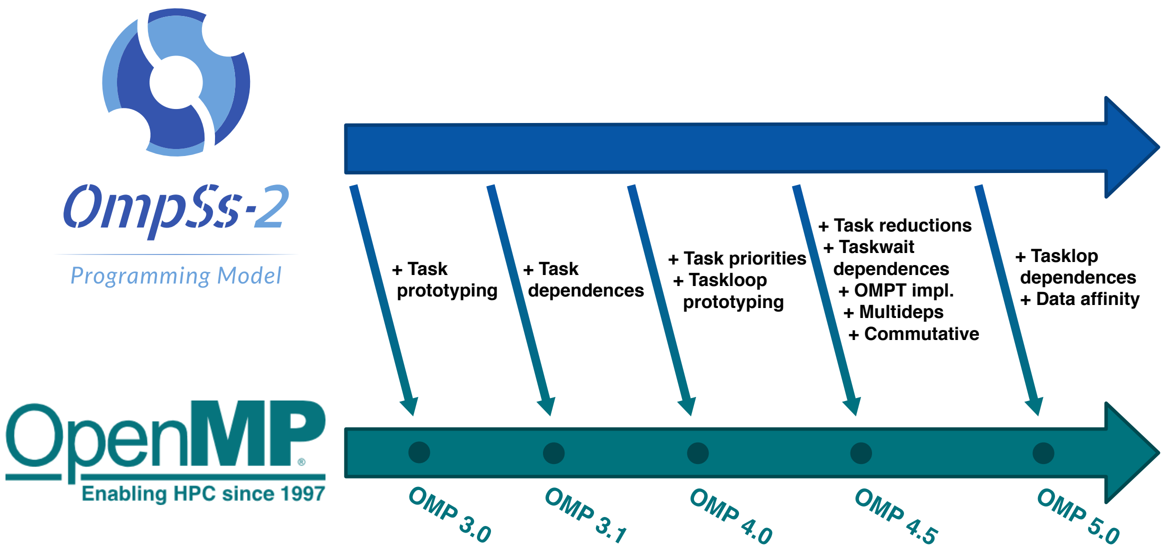


Fig. 3.2: Summary of the contributions from StarSs/OmpSs in OpenMP

Starting from version 3.0 released in May 2008, OpenMP included the support for asynchronous tasks. The reference implementation, which was used to measure the benefits that tasks provided to the programming model, was developed at Barcelona Supercomputing Center or BSC [31] and was done with the Nanos4 runtime library and the Mercurium source-to-source compiler.

The next contribution, which was included in OpenMP's version 4.0 released in July 2013, was the extension of the tasking model to support data dependences, one of the strongest points of OmpSs that allows defining synchronization between tasks. This feature was tested using Mercurium source-to-source compiler and the Nanos++ RTL.

In OpenMP's version 4.5, released in November 2015, the tasking model was extended with the taskloop construct. It used Nanos++ as the reference implementation to validate these ideas. BSC also contributed to version 4.5 adding the priority clause to task and taskloop constructs. BSC continues to influence OpenMP in their newest version 5.0 while there is more to come.

3.2 Main Features

OmpSs-2's main objective is to provide an environment to develop applications for modern High-Performance Computing systems. There are two ideas which make OmpSs-2 a productive model; the performance it provides, and the ease of use. Programs developed with OmpSs-2 must be able to deliver a reasonable performance when compared to other programming models targeting the same architecture(s). When it comes to ease of use, OmpSs-2 has been designed using principles that have been praised by their effectiveness. Some of the most remarkable features from the OmpSs-2 programming model include:

- **Lifetime of task data environment**: A task completes its execution when the last line of code of its body is executed. However, it does not become "deeply completed" until all of its children tasks have become deeply complete. If a task has no children, it becomes deeply completed when the last line of its body is executed. The data environment of a task is preserved until it is deeply completed. This environment has all the variables of the task when it is created.
- **Nested dependency domain connection**: Dependences of a task propagate to its children as if the task did not exist. When a task finishes, its outgoing dependences are replaced by those generated by its children.
- **Early release of dependences**: Once a task is completed, it will release all the dependences that are not included on any unfinished descendant task. If the `wait` clause is specified in the task construct, however, all its dependences will be released at once when the task becomes deeply completed.
- **Weak dependences**: The `weakin/weakout` clauses specify potential dependences only existent in children tasks. These do not delay the execution of the task.
- **Native offload API**: A new asynchronous API to execute OmpSs-2 kernels on a specified set of CPUs from any application, including Java, Python, R, etc.
- **Task Pause/Resume API**: A new API that can be used to suspend and resume the execution of a task programmatically. This API improves the interoperability and performance of hybrid MPI and OmpSs-2 applications.

3.3 Programming Model

As previously mentioned, a huge difference between OmpSs-2 and OpenMP is the lack of the `parallel` clause to specify parallel regions in user applications. This clause is needed in OpenMP as it uses a fork-join execution model where users need to specify where and when parallelism starts and ends. OmpSs-2 uses the model implemented by StarSs where parallelism is implicitly created at the start of executions. In other words, parallelism could be seen as a pool of threads – hence the name, thread-pool execution model – that the underlying runtime library uses during the execution. As the user has no control over resources, OpenMP-like methods such as `omp_get_num_threads()` are not available.

As previously mentioned, in OmpSs-2 a thread team is created since the beginning of the execution. This team is divided into a master thread and various worker threads. The master thread sequentially executes user code as a task, which is called the initial task. This task includes the whole program within it. The other threads (worker threads) wait while polling for concurrent tasks. These actively wait until any task is available. When executing an OmpSs-2 code sequentially (directives should be ignored), the program should correctly behave as a non-OmpSs-2 code.

3.4 Tasks

Parallelism is expressed through tasks in OmpSs-2. **Tasks** are pieces of code that can be executed concurrently (unless specified through directives) at run-time. When an execution reaches a point of the code where there is a task directive, instead of directly executing the region within the task, an instance containing that code is created. The execution of this instance or task is offloaded to the underlying runtime library. This runtime, at some point, will execute the task on the available resources. These resources are threads, which are assigned tasks when available. The execution of the task, as previously mentioned, might not be immediate. Within the constraints specified by users, through dependences, the scheduling policies can take whatever decisions they please. This, of course, always assuring the correct execution of the program. Hence why the task might be executed immediately or postponed until scheduling constraints are met.

Threads can suspend tasks at specific points, called scheduling points, to resume or begin executing a different task. Suspended tasks will be resumed later by the same thread in which they were suspended if these are ”tied” tasks. On the other hand, ”untied” tasks can be resumed by any worker thread.

Task declarations may appear within a task declaration itself. This is also named **task nesting** and allows defining multiple levels of parallelism. This may lead to performance improvements in applications since the runtime can exploit data or temporal locality. Multiple levels of parallelism are required, also, by recursive applications.

Specifying tasks is as easy as using the `task` construct shown below. This construct may appear inside the code in a program, or outlined in procedure calls. The usage will mark whatever is within the following statement as a task.

```
#pragma oss task [clauses]
{ ... }
```

Code 3.1: Snippet of code showing the usage of the `task` construct.

Next is a list of the possible clauses for the `task` construct. Below it is their usage and a brief explanation.

- `private(<list>)`
- `firstprivate(<list>)`
- `shared(<list>)`
- `depend(<type>:<memory-reference-list>)`
- `<depend-type>(<memory-reference-list>)`
- `priority(<expresion>)`
- `cost(<expresion>)`
- `if(<scalar-expression>)`
- `final(<scalar-expresion>)`
- `label(<string>)`
- `[ wait | weakwait ]`

The `private(<list>)`, `firstprivate(<list>)` and `shared(<list>)` clauses allow the specification of data sharing attributes of the list of variables inside the clause. More about these three clauses and their implicit attributes can be found in OmpSs-2 specification [30].

Synchronizing parallel tasks of an application is a must in order to create a correct execution. This is because tasks often depend on data computed by other tasks. OmpSs offers two ways of expressing this: data dependences, and explicit directives to set synchronization points. This can be achieved through the `depend` clause. Both the syntax and a thorough explanation of the `depend` clause can be found in section 3.6.

The `priority` clause is a hint for the runtime about a task. The bigger the value in this clause, the higher the priority. Lower numbers then, indicate a low priority. The default priority is 0. The expression in the clause is evaluated as a signed integer. This way, strictly positive priorities indicate a higher priority than the default, and negative priorities indicate a lower priority than the default.

Since the `cost` clause is one of the elements this project is based on, its features and description are discussed in later sections.

If the expression of the `if` clause evaluates to true, the execution of the new task can be deferred. On the other hand, if it evaluated to false, the current task must be suspended until the new task is completed.

As aforementioned in chapter 2, if the expression of the `final` clause evaluates to true, the newly created task will be a final task. This means that every code that generates a task will also generate final tasks. Also, when executing within a final task, all the task directives within the final task will be ignored and thus the code inside them will be executed immediately. As a final note, tasks created within a final task can use the data environment of its parent task.

Tasks with the `wait` clause will execute a taskwait-like operation right after exiting its code. As this is done outside the task code's scope, this occurs when the task has abandoned the stack. Due to this, the usage of the wait clause is restricted to tasks that, when exiting, have no subtasks accessing the variables of the parent task. Otherwise, regular taskwaits should be used.

The `label` clause specifies a string that has many uses. It can be used in any tool to identify the task with a far more human-readable format.

Snippet 3.2 shows a dummy example of some of the aforementioned constructs and clauses.

```c
#pragma oss task label(recurse) cost(N*N + N) inout(x, y)
void recurse(int * x, int * y, int N) {
   if (x == N) return;
   else {
      *x = (*x) + 1;
      *y = (*y) * (*y);
      recurse(x, y, N);
   }
}

int main () {
   int x, y, N = 10;
   #pragma oss task label(initialize) cost(2) priority(1) out(&x, &y) {
      x = 1;
      y = 2;
   }

   recurse(&x, &y, N);
   #pragma oss taskwait
}
```

Code 3.2: Snippet of code exemplifying OmpSs-2 constructs and clauses.

When the thread of execution reaches a `#pragma oss task` construct, a new task is created. As shown, it is an instance of a task with label `initialize`, as it is a task that initializes variables. This task has a higher priority than others since the default priority is 0. It has a cost value of 2 and two `out` dependences on variables `x` and `y`.

The task construct allows annotations of function declarations or definitions, as well as the already shown annotations of structured-blocks. If a function is annotated with the task construct, each call to the function becomes a task creation point. This is shown in the same snippet in function `recurse`. Any invocation of that function will generate a task with the restrictions and properties of the clauses specified in its declaration.

If the `taskwait` construct had not been specified in the snippet above, the execution could have terminated without knowing if all or any task had been executed. To ensure correct finalization of tasks this construct must be used.

3.5 Task Scheduling

When a task reaches a task scheduling point, the implementation may decide to switch from that task to another one. Task scheduling points may occur when:

- a task is generating code
- a taskwait directive is found
- a task has just been completed

Switching from one task to another is known as task switching. Task switching may imply the beginning of a non-previously executed task or resuming the execution of a paused task. Task switching is limited through the following constraints:

- The set of available tasks is initially formed by the set of tasks included in the ready task pool.
- If a thread has executed a tied task, and there is a task switch, the tied task will only be able to be resumed by the thread which performed the task switch (i.e. the set of available tasks for a thread does not include tied tasks that have been previously executed by a different thread).
- When creating a task with the `if` clause, if the expression within the clause evaluates to false the runtime must offer a mechanism to immediately execute this task (usually by the same thread that creates it).
- When executing in a `final` context, all the task directives within the ”final” task will be ignored. Instead, the code within the inner tasks will be executed immediately as simple routine calls

3.6 Synchronization of Tasks through Dependences

As mentioned, OmpSs-2 allows synchronizing parallelism between different tasks through data dependences (clauses in the task directive). Tasks often require data for their computations. The usual scenario is that tasks need some input data to perform operations, and with the input data they produce new data or modify existing data that can be used by other tasks.

OmpSs-2's underlying runtime uses these dependences along with the order of creation of tasks to perform an analysis. This analysis creates constraints that relate to execution order. These constraints are related to tasks, and they produce a correct order of execution for the applications.

There are several types of dependences. They can be categorized by order and operation into Read-after-Write (RaW), Write-after-Write (WaW) or Write-after-Read(WaR). When a task must be created, its explicit dependences are compared against the dependences of already existing tasks. If there is a match, newly created task becomes a successor of the matched task(s). This process generates a dependency graph. Tasks are scheduled to be executed as soon as all their predecessors have finished.

The dependency clauses allow the runtime to infer scheduling restrictions from the parameters within them. These restrictions are the so-called dependences. The syntax of this clause specifies first the dependence type and, after a colon, a memory reference list. The types allow the keywords `in`, `out`, `inout` and `concurrent`. After these keywords, there is a comma-separated list of memory references. All these clauses also admit a comma-separated list of elements (memory references), and an explanation for each, extracted from [30], is listed next:

- `in(memory-reference-list)`: If a task has an `in` clause that evaluates to a given lvalue, then the task will not be eligible to run as long as a previously created sibling task with an `out`, `inout` or `concurrent` clause applying to the same lvalue has not finished its execution.

- `out(memory-reference-list)`: If a task has an `out` clause that evaluates to a given lvalue, then the task will not be eligible to run as long as a previously created sibling task with an `in`, `out`, `inout` or `concurrent` clause applying to the same lvalue has not finished its execution.

- `inout(memory-reference-list)`: If a task has an inout clause that evaluates to a given lvalue, then it is considered as if it had appeared in an `in` clause and an `out` clause. Thus, the semantics for the `in` and `out` clauses apply.

- `concurrent(memory-reference-list)`: The `concurrent` clause is a special version of the `inout` clause where dependences are computed with respect to `in`, `out` and `inout` but not with respect to other `concurrent` clauses. As it relaxes the synchronization between tasks users must ensure that either tasks can be executed concurrently either additional synchronization is used.

```
void foo (int *a, int *b) {
   for (int i = 1; i < N; i++) {
      #pragma oss task in(a[i-1]) inout(a[i]) out(b[i])
      propagate(&a[i-1], &a[i], &b[i]);

      #pragma oss task in(b[i-1]) inout(b[i])
      correct(&b[i-1], &b[i]);
   }
}
```

Code 3.3: Snippet of code showcasing dependence clauses.

Snippet 3.3 above showcases a dummy example with some of the aforementioned clauses. To better grasp the whole view, however, snippet 3.4 is included, which showcases task nesting and dependences. This snippet was taken from a larger snipper from OmpSs-2's website [30].

```
#pragma oss task depend(inout:  a, b) // Task T1
{
   a++; b++;
   #pragma oss task depend(inout:  a) // Task T1.1
   a += ...;
   #pragma oss task depend(inout:  b) // Task T1.2
   b += ...;
   #pragma oss taskwait
}
#pragma oss task depend(in:  a, b) depend(out:  z, c, d) // Task T2
{
   z = ...;
   #pragma oss task depend(in:  a) depend(out:  c) // Task T2.1
   c = ... + a + ...;
   #pragma oss task depend(in:  b) depend(out:  d) // Task T2.2
   d = ... + b + ...;
   #pragma oss taskwait
}
#pragma oss task depend(in:a, b, d) depend(out:e, f) // Task T3
{
   #pragma oss task depend(in:a, d) depend(out:e) // Task T3.1
   e = ... + a + d + ...;
   #pragma oss task depend(in:b) depend(out:f) // Task T3.2
   f = ... + b + ...;
   #pragma oss taskwait
}
```

Code 3.4: Snippet of code showcasing dependence clauses within task nesting.

Next, we explain how the code above is parallelized. If we wanted to parallelize the original code, we would need to follow the next steps:

1. First we would add the directives (or pragmas from now on) of the outermost tasks. Since the data used within these conflicts with other outermost tasks, we would need to specify this using the `depend` clauses. Usually, it is a good practice to protect all the data accesses of tasks. This reduces the proneness to errors, improves the maintainability of the code and reduces conflicts in accesses.

2. Secondly, in each of the inner tasks we would identify separate computations or procedures. We would convert each of these (if desired) to tasks.

3. Finally, we would add taskwait directives at the end of the outermost tasks.

Synchronization in the previous example happens in two ways. First, the outermost tasks contain dependences that mimic the ones from their sub-children. This is to ensure that there are no data races between the inner tasks and any other task from the program. Also, to ensure that the dependences are not released too early, a taskwait is added at the end of each outermost task. This is so that the dependences of the outermost task are not released until all its children have completed their execution.

The synchronization in the previous example is correct, as it creates an execution order that meets the requirements of the program. However, inserting memory references in the depend clauses of outermost tasks that these do not need for themselves is not a good practice. This delays the execution of outermost tasks until the dependences they specify are met. Subsequently, the creation of its children tasks is also delayed. Also, the taskwait directive causes all the dependences specified to be released at the same time. To avoid these negative effects, OmpSs-2 introduces other clauses such as:

- **Weak dependences**: When it comes to the memory reference list specified in the depend clause, it may happen that the task that specifies the list needs such references. It may also happen that they are needed by one or more subtasks of the task. In a third scenario, both subtasks and the task itself might need the data. In the event that only subtasks need the data, as previously mentioned, it might only be specified in the parent task as a synchronization mechanism. To optimize this scenario, the dependence system of OmpSs-2 was extended to define the `weak` counterparts of the `in`, `out` and `inout` dependences. Their semantics are symmetrical to the ones used by clauses without the `weak` prefix. However, the weak variants specify that tasks do not perform any action with the data specified in the clause, their subtasks do.

- **Extended lvalues**: All the clauses from the dependence system allow extended lvalues from those of C and C++. Two extensions are allowed:

  - Array sections allow specifying multiple items of an array in a single expression. There are two ways of specifying this through array sections:

    * `a[lower : upper]`: Through this syntax, all the elements from `a` in the range specified – from lower to upper both included – are referenced. If 'lower' is missing, the default value is 0. If the reference is applied to an array and 'upper' is missing, the default value is the last element of the dimension of the array.

    * `a[lower ; size]`: This syntax specifies that the referenced elements from `a` are within the range that starts at `lower` and ends at `lower + size - 1` (both limits included).
  - The other extension is 'shaping expressions', that allow recasting pointers into arrays to recover the dimension sizes. Shaping expressions are one or more expressions of the form `[size]` before a pointer.

- **Taskwait dependences**: In addition to dependences, it is possible to introduce synchronization points in an OmpSs-2 application. These are defined through the `taskwait` directive. When the execution of a task reaches one of these points, it is halted until all previous sibling tasks are completed. This can be used through introducing the dependence clauses (any kind) in the taskwait directive.

- **Multidependences**: Multidependences is a powerful feature. It allows defining a dynamic number of dependences. A multidependence consists of two parts:

  - an lvalue expression that contains references to an iterator. The iterator does not exist in the program.
  - the definition of the previously mentioned iterator and its range of values. Depending on the programming language, the syntax is different:
    * `dependence-type(reference-list, iterator-name=lower;size)` for C / C++.
    * `dependence-type([reference-list, iterator-name=lower,size])` for Fortran.

4 | The Nanos6 Runtime

Nanos6 [32] is a runtime that implements the OmpSs-2 [30] parallel programming model, developed by the Programming Models group at the Barcelona Supercomputing Center [31]. Nanos6 applications can be executed as is. The number of cores that are used is controlled by running the application through the taskset command. For instance, `taskset -c 0-2,4 ./app` would run `app` on cores 0, 1, 2, and 4.

When it comes to this project, Nanos6 alongside OmpSs-2 would be the core of it. Everything we propose is exemplified using Nanos6 to serve as a runtime.

4.1 Tracing

Nanos6 applications, unlike its predecessor Nanos++, do not require recompiling their code to generate extrae traces or to generate additional information. This is instead controlled through environment variables, at run-time. Generating **extrae** (section 6.1.3) traces is as simple as running the application with the `NANOS6` environment variable set to 'extrae' (`NANOS6=extrae`). More details about this and other environment variables are explained below in section 4.2.

It is important to note that the resulting traces show the activity of threads instead of the activity at each CPU.

4.2 Environment Variables

Next is a list of some of the most important environment variables when using Nanos6. As a note, other variables unspecified in this list are found later in this thesis. These are yet to be added to the current public version of Nanos6.

- `NANOS6`: Possibly the most important variable, it lets users select the variant of the runtime that will be used. Next are the currently available variants and a brief about them:
  - **optimized**: This is the default value and selects the standard runtime based on pthreads.

  - **debug**: Runtime compiled without optimization and with all assertions turned on.
  - **extrae**: Instrumented to produce extrae traces.
  - **verbose**: Instrumented to emit a log of the execution.
  - **verbose-debug**: Instrumented to emit a log of the execution and compiled without optimization and with all assertions turned on.
  - **graph**: Instrumented to produce a graph of the execution. Only practical for small graphs.
  - **profile**: Instrumented to produce a function and source code execution profile.
  - **stats**: Instrumented to produce a summary of metrics of the execution.
  - **stats-papi**: Instrumented to produce a summary of metrics of the execution including hardware counters.

- `NANOS6_STACK_SIZE`: Nanos6 by default allocates stacks of 8 MB for its worker threads. In some codes, this may not be enough. For instance, when converting Fortran codes, some global variables may need to be converted into local variables. This may substantially increase the amount of stack required to run the code and may surpass the space that is available. To solve that problem, the stack size can be set through this environment variable. Its value is expressed in bytes but it also accepts the K, M, G, T and E suffixes, that are interpreted as power of 2 multipliers. An example illustrating this variable could be: `export NANOS6_STACK_SIZE=16M`.

- `NANOS6_LOADER_VERBOSE`: This variable controls the verbosity of the Nanos6 Loader. By default (value 0) it is quiet. If the environment variable has value 1 it will emit to standard error the actions that it takes and their outcome.

  By default the loader will attempt to load the actual runtime library from the path determined by the operating system (taking into account `rpath` and the `LD_LIBRARY_PATH` environment variable). If it fails to load the library, then it will attempt to locate the library at the same location as the nanos6 loader.

  The default search path can be overridden through the `NANOS6_LIBRARY_PATH` environment variable. If it exists the first attempt at loading the runtime will be performed at the directory specified in that variable. The loader does not accept multiple directories in that variable.

  The nanos6 loader resolves the addresses of the API functions to the actual runtime implementation. In addition it also checks for the implementation of some features, and if they are not found, it will either complain or emit a warning and fall back to a compatible but less powerful implementation. More specifically, the loader accepts running applications that make use of weak dependencies and will fall back to strong dependencies if the runtime does not have support for them.

- `NANOS6_SCHEDULER`: The scheduler can be specified through an environment variable called `NANOS6_SCHEDULER`. Currently it accepts:
  - **default, priority**: The default priority-aware scheduler with one immediate successor reservation per CPU.
  - **naive**: A very simple scheduler in LIFO mode.
  - **fifo**: A very simple scheduler in FIFO mode.
  - **immediatesuccessor**: A scheduler that reserves an immediate successor for each CPU.
  - **iswp**: A scheduler that reserves an immediate successor for each CPU and that when starved, leaves one thread polling for new work.
  - **iswpfifo**: A scheduler that reserves an immediate successor for each CPU and that when starved, leaves one thread polling for new work. This is the FIFO version.

4.3 Runtime Variants

Through the `NANOS6` environment variable, several runtime variants can be selected. This section discusses the features of some of the most valuable variants for developers.

- **Verbose**: By default this variant produces a lot of information. This can be controlled through the `NANOS6_VERBOSE` environment variable. This variable can contain a comma separated list of areas. These areas are shown in table 4.1.

Section	Description
AddTask	Task creation
Blocking	Blocking and unblocking within a task through calls tot he blocking API
ComputePlaceManagement	Starting and stopping compute places (CPUs, GPUs, . . .)
DependenciesByAccess	Dependencies by accesses
DependenciesByAccessLinks	Dependencies by the links between the accesses to the same data
DependenciesByGroup	Dependencies by groups of tasks that determine common predecessors and common successors
LeaderThread	Execution of the leader thread.
LoggingMessages	Additional logging messages
TaskExecution	Task execution
TaskStatus	Task status transitions
TaskWait	Entering and exiting taskwaits
ThreadManagement	Thread creation, activation and suspension
UserMutex	User-side mutexes (critical)

Table 4.1: Different information available in the verbose variant of the runtime

- **Graph**: This variant is used to generate a graph which represents the dependencies between tasks. By default, the graph nodes include the full path of the source code. To remove the directories, the `NANOS6_GRAPH_SHORTEN_FILENAMES` environment variable has to be set to 1.

  The resulting file is a PDF that contains several pages. Each page represents the graph at a given point in time. With `NANOS6_GRAPH_SHOW_DEAD_DEPENDENCIES`, when it is set to value 1, it forces future and previous dependencies to be shown with different graphical attributes.

  The `NANOS6_GRAPH_DISPLAY` environment variable, if set to 1, will make the resulting PDF to be opened automatically. The default viewer is xdg-open, but it can be overridden through the `NANOS6_GRAPH_DISPLAY_COMMAND` environment variable.

- **Stats**: To enable collecting statistics, applications must be ran with the `NANOS6` environment variable set to either `stats` or `stats-papi`. The first collects timing statistics and the second also records hardware counters. By default, the statistics are emitted standard error when the program ends. The output can be sent to a file through the `NANOS6_STATS_FILE` envar. The contents of the output contain the average for each task type and the total task average of metrics such as: number of instances, mean instantiation time, mean pending time (not ready due to dependencies), mean ready time, mean execution time, mean blocked time (due to a critical or a taskwait), mean zombie time (finished but not yet destroyed), mean lifetime (time between creation and destruction). The output also contains information about: number of CPUs, total number of threads, mean threads per CPU, mean tasks per thread, mean thread lifetime, mean thread running time.

5 | Intel® Resource Director Technology

Intel® Resource Director Technology [33] is a software package that provides support for Cache Monitoring Technology (CMT), Memory Bandwidth Monitoring (MBM), Cache Allocation Technology (CAT), Code and Data Prioritization (CDP) and Memory Bandwidth Allocation (MBA).

It provides a hardware framework to both monitor and manage shared computing resources, like cache or memory bandwidth. Large amounts of workloads running concurrently on a single system increase the demand for shared resources. This lowers the overall performance of the system. Intel® RDT technologies can monitor and control the usage or allocation of crucial shared system resources to help improve these scenarios.

5.1 Motivation

The features it provides/support make this library one of the most complete libraries when it comes to hardware counters related to cache and memory. By allowing not only monitoring but also certain allocation features, this library becomes even more interesting. Table 5.1 shows a comparison of supported features between similar libraries in the field. Because of all the features and capabilities it provides, the usage of Intel's® CMT-CAT APIs is integrated into this project. This is to obtain a complete monitoring, which will be further discussed in chapter 7.

	Core	**Task**	**CMT**	**MBM**	**L3 CAT**	**L3 CDP**	**L2 CAT**	**MBA**
intel-cmt-cat	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Intel(R) PCM	Yes	No	Yes	Yes	No	No	No	No
Linux perf	Yes	Yes	Yes	Yes	No	No	No	No
Linux cgroup	No	Yes	No	No	Yes	No	No	No
Linux resctrl	Yes	Yes	No	No	Yes	Yes	Yes	No

Table 5.1: Comparison of support for different monitoring & allocation technologies between common profiling libraries

Through the use of CMT, applications can be monitored simultaneously while running on a platform. Studies conducted by Intel [1] showcased a number of applications running on a 14-core Intel® Xeon® E5-2600 v3 processor-based system with RMIDs pinned to each core. This can be seen in the article cited from these studies, in a plot marked as figure 4. As applications run, their cache occupancy can be sampled periodically. Periodic spikes in occupancy (green line) are visible from a periodic operating system task. In the middle of the plot, a memory streaming application is invoked on a core, which quickly consumes all of the L3 cache and then terminates. Using CMT, this aggressor application can be detected, and if its behavior is found to interfere with more important applications, the aggressor application could be moved to another processor or another node. If the aggressor application is simply resource-hungry but high-priority then its true cache sensitivity can be measured over time using CMT.

Last level cache occupancy curves collected for various applications could be used to build long-term histories of applications and schedule optimally across sockets. For instance as shown on the left side of figure 5.1, if two compute-intensive applications are co-located on a processor with small working sets, then applications could be rebalanced across sockets to optimize L3 cache utilization and potentially increase performance.

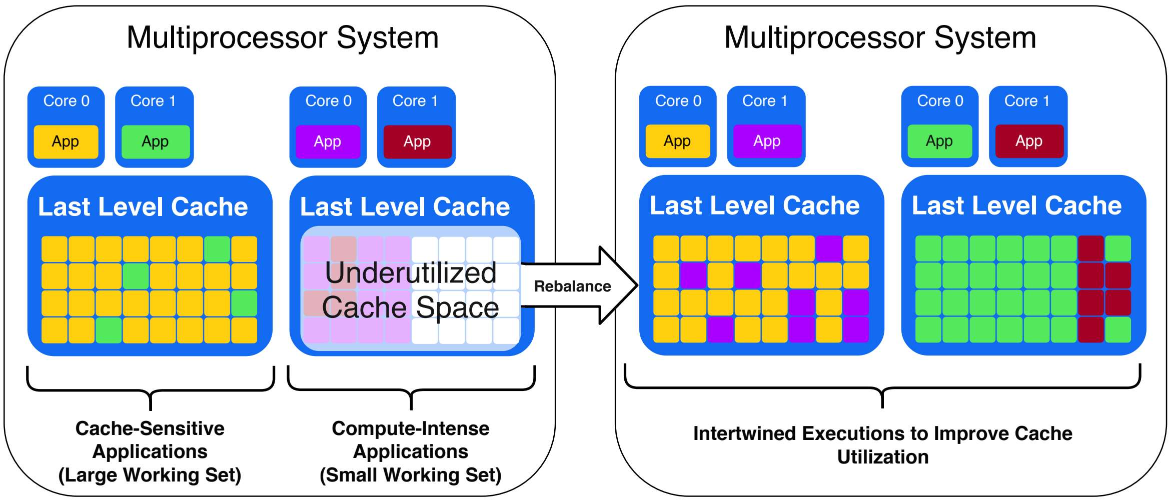

Multiprocessor System
Core 0
Core 1
App
Last Level Cache
Underutilized Cache Space
Rebalance
Cache-Sensitive Applications (Large Working Set)
Compute-Intense Applications (Small Working Set)
Multiprocessor System
Intertwined Executions to Improve Cache Utilization

Fig. 5.1: Rebalancing applications across processors for optimal cache utilization using CMT [1]

These use cases can easily be seen as, instead of processes, rebalancing tasks or intertwining them when detecting that they are memory-hungry. These capabilities bring, therefore, new ideas to the table which this project might benefit from. Apart from monitoring, it would be interesting to integrate allocation techniques into a runtime.

5.2 Support

Support for Intel® RDT in the Linux kernel was originally obtained through Linux perf system call extensions for both CMT and MBM. Recently, the Resctrl interface added CAT, CDP, and MBA support. Intel-cmt-cat [34], their software package, works seamlessly in all Linux kernel versions. Table 5.2 shows the availability of the library's features concerning different processor models.

	CMT	MBM	L3 CAT	L3 CDP	L2 CAT	MBA
Intel® Xeon® processor E5 v3	Yes	No	Yes	No	No	No
Intel® Xeon® processor D	Yes	Yes	Yes	No	No	No
Intel® Xeon® processor E3 v4	No	No	Yes	No	No	No
Intel® Xeon® processor E5 v4	Yes	Yes	Yes	Yes	No	No
Intel® Xeon® Scalable Processors	Yes	Yes	Yes	Yes	No	Yes
Intel® Atom® processor for Server C3000	No	No	No	No	Yes	No

Table 5.2: Availability of Intel® PQoS features on different processors

5.3 Interoperability

Using intel-cmt-cat or Intel® PCM software along with Linux perf and cgroup is not currently allowed. Use of Linux perf for CMT & MBM and intel-cmt-cat for CAT & CDP is not permitted. This is due to Linux perf overriding existing CAT configuration during its operations. Table 5.3 illustrates the current status of interoperability between different libraries and PQoS.

	intel-cmt-cat	Intel® PCM	Linux perf	Linux cgroup	Linux resctrl
intel-cmt-cat	Yes	Yes	Yes	No	Yes
Intel® PCM	Yes	Yes	No	No	No
Linux perf	Yes	No	Yes	Yes	Yes
Linux cgroup	No	No	Yes	Yes	No
Linux resctrl	Yes	No	Yes	No	Yes

Table 5.3: Intel® PQoS software interoperability matrix

5.4 Monitoring & Other Features

This library, also known as Intel® Perceived Quality of Service or PQoS for short, allows two modes of operation or interfaces. These are the MSR and the OS interfaces.

MSR or Model Specific Registers are used to configure the underlying platform by programming these registers directly. This interface requires no kernel support for Intel® RDT, but it is limited to per-core monitoring and managing of resources.

The second interface is the OS interface. When using this interface, the library leverages Linux kernel extensions to program the aforementioned technologies. This enabled monitoring and managing resources on a per-process basis as well as the per-core basis. Hence why this interface should be used when available. Currently, this interface returns invalid data on some architectures. Thus, it is a must for developers to keep up to date with any news by checking the compatibility tables provided by Intel®, like the one shown above.

Even though intel-cmt-cat can be integrated into runtimes with their API to merely fetch the desired counters, they offer pre-built tools. Next, some of the potential use cases this library can provide to developers are explained, as well as a thorough description and example of all the technologies. These examples are shown using the `pqos` binaries installed by default with the library. There are more use cases available in [33], however, the previously mentioned cases are the ones this project will benefit the most from.

5.4.1 Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM)

The library allows monitoring both cache and memory bandwidth usage. As mentioned above, this could be done separately when using their APIs. However, for simplicity, both explanations are discussed in this section as the `pqos` binary includes both these technologies into the same command.

CMT is a feature that allows operating systems (OS) or hypervisors/virtual machines (VMM) to obtain the cache usage made by applications running on their platform. Currently, this feature allows L3 cache monitoring (last level cache in most platforms). CMT provides an API:

- To detect if the platform supports this monitoring capability (via CPUID).
- For OS or VMMs to create software-defined IDs for each application or VM that is scheduled to run on the platform. This ID is named the Resource Monitoring ID (RMID).
- To monitor cache occupancy on per-RMID.
- For OS or VMMs to read LLC occupancy for each RMID.

By analyzing cache utilization, the OS or VMM can optimize scheduling policy decisions to improve overall system performance. Cache monitoring allows cache utilization to be simultaneously tracked for many concurrently running independent threads, applications or VMs at runtime, enabling advanced optimization techniques to be applied in real-time.

Applications may over-utilize shared resources. Detecting these so-called “noisy neighbor” applications is an essential feature. Memory Bandwidth Monitoring (MBM) helps in solving this issue by providing per-thread memory bandwidth monitoring for all threads at the same time. MBM uses the same underlying infrastructure as CMT. This includes feature enumeration (via CPUID), and new event codes to poll memory bandwidth from the local memory controllers, or from all controllers including remote ones. CMT and MBM share the following mechanisms:

- A mechanism to enumerate the presence of the RDT Monitoring capabilities within the platform (via a CPUID feature bit).
- A framework to enumerate the details of each sub-feature (including CMT and MBM, as discussed later, via CPUID leaves and sub-leaves).
- A mechanism for OS or hypervisors to indicate a software-defined ID (RMID) for each software thread (applications, VMs, ...) that is scheduled to run on a core. Figure 5.2 shows the infrastructure of monitoring when using Resource Monitoring IDs (RMIDs).
- Mechanisms in hardware to monitor cache occupancy and bandwidth statistics as applicable to a given product generation on a per software-id basis.
- Mechanisms for OS or hypervisors to read collected metrics such as LLC occupancy or memory bandwidth for a given software ID in real time.

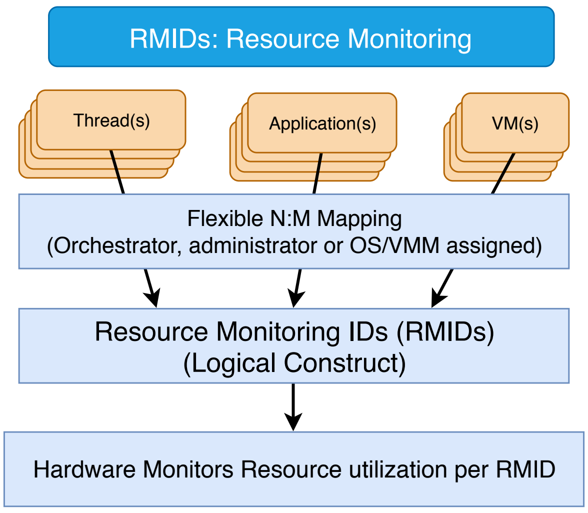

RMIDs: Resource Monitoring
Thread(s)
Application(s)
VM(s)
Flexible N:M Mapping
(Orchestrator, administrator or OS/VMM assigned)
Resource Monitoring IDs (RMIDs)
(Logical Construct)
Hardware Monitors Resource utilization per RMID

Fig. 5.2: RMIDs can be used to track the resource usage of threads, apps, VMs or containers. Software assigns RMIDs based on specific monitoring needs [1]

Example of Usage

Using the `pqos` command, Instructions Per Cycle, Last Level Cache misses, Last Level Cache usage (in KB), Memory Bandwidth (Local, in MB/s) and Memory Bandwidth (Remote, in MB/s) can be monitored. This information is gathered in real time. Thus, if integrated with a runtime, this library can provide runtime schedulers with valuable information about application behaviors and bottlenecks.

The following command – `pqos -m "all:[0-2],[3-5],[6-8];"` – would monitor (`-m` option) all events (`all` keyword) on the list of cores shown, in groups as delimited by the corresponding brackets. The list of available keywords currently includes `llc`, for Last Level Cache usage monitoring, `mbl`, for Memory Bandwidth usage monitoring in the Local node, and `mbr`, for Memory Bandwidth usage monitoring in Remote nodes. The previous command could generate an output such as the following:

```
CORE    IPC    MISSES    LLC[KB]   MBL[MB/s]   MBR[MB/s]
 0-2   0.28     7893k      383.2       901.2       430.8
 3-5   0.28       45k       25.3    361282.6        22.4
 6-8   0.26    89468k     6778.8     43904.3         4.3
```

5.4.2 Cache Allocation Technology (CAT)

While shared resources provide good performance scalability and throughput, specific applications such as background video streaming over-utilize cache, which reduces the overall performance of critical applications. For example in figure 5.3, the “noisy neighbor” (on core zero) consumes excessive last-level cache, which is allocated on a first-come-first-served basis. This can cause performance loss in higher-priority applications (shown in green on core one).

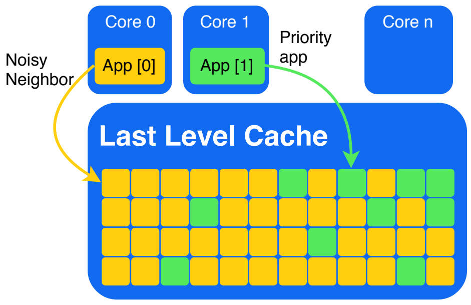


Fig. 5.3: A “noisy neighbor” on core zero over-utilizes shared resources in a platform, causing performance inversion [1]

Cache Allocation Technology (CAT) provides an API to control the usage of cache space consumed by a given thread, application, VM, or container. This allows operative systems to protect important processes, or hypervisors to prioritize important VMs even in noisy environments. The basic mechanisms of CAT include:

- The ability to enumerate the CAT capability and the associated LLC allocation support via CPUID.

- Interfaces for operative systems or hypervisors so that they can group applications into classes of service (CLOS). These groups then can be set a certain limit of last-level cache occupation. These interfaces are based on Model-Specific Registers.

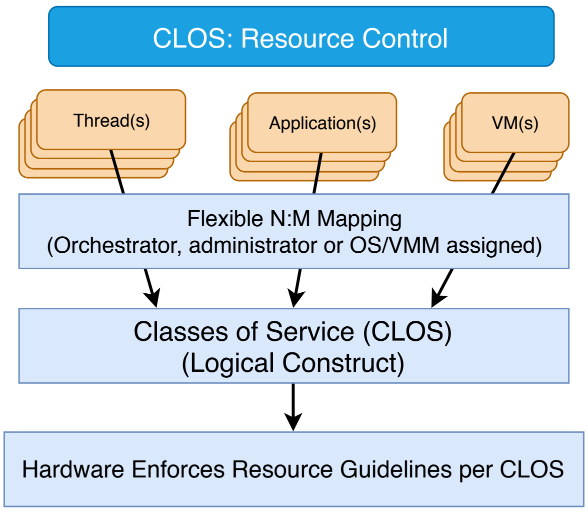
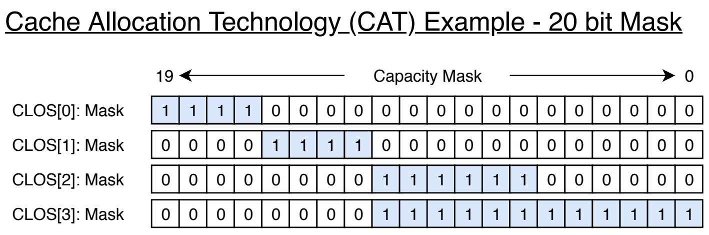

CLOS: Resource Control
Thread(s)
Application(s)
VM(s)
Flexible N:M Mapping
(Orchestrator, administrator or OS/VMM assigned)
Classes of Service (CLOS)
(Logical Construct)
Hardware Enforces Resource Guidelines per CLOS

Fig. 5.4: CLOS enable flexible control over threads, apps, VMs, or containers [1]

Similarly to the RMIDs mentioned in section 5.4.1, figure 5.4 shows how an OS may group applications into these classes of service (CLOS). CAT also has capacity bitmasks (CBMs) for each CLOS. These bitmasks limit the amount of cache that threads in a CLOS can allocate. The values within the bitmasks indicate the amount of cache available. As shown in figure 5.5, overlapping or isolating regions is possible. In this figure, CLOS [1] has less cache availability than CLOS [3], thus it could be considered that CLOS [1] has a lower priority.


Cache Allocation Technology (CAT) Example - 20 bit Mask

19 ← Capacity Mask → 0

CLOS[0]: Mask	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CLOS[1]: Mask	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
CLOS[2]: Mask	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
CLOS[3]: Mask	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

Fig. 5.5: Capacity Bitmasks overlap and isolation across multiple CLOS [1]

Example of Usage

Also using the aforementioned `pqos` command, classes of services for cores can be set, and bitmasks can be assigned to these. `pqos -e "llc:1=0x000f;llc:2=0x0ff0;"` – would setup 2 classes of service. The first one, CLOS1, is set to the first 4 ways of the LLC. The next one, CLOS2, is set to the next 8 cache ways. The next command –

`pqos -e "llc:1=0x000f;llc@0,1:2=0x0ff0;`" – would set CLOS1 on all sockets and CLOS2 on socket 0 and 1. Next a possible output of this last command is shown.

```
L3CA CLOS definitions for Socket 0:
    L3CA CLOS0 => MASK 0xfffff
    L3CA CLOS1 => MASK 0xf
    L3CA CLOS2 => MASK 0xff0
    ...
L3CA CLOS definitions for Socket 1:
    L3CA CLOS0 => MASK 0xfffff
    L3CA CLOS1 => MASK 0xf
    L3CA CLOS2 => MASK 0xff0
    ...
```

Once the CLOS are created, all that is left to do is assign cores to these. This can also be done with the `pqos` command like so: `pqos -a "llc:1=0,2,6-10;llc:2=1;"`. The previous command would associate cores 0, 2, and 6 to 10 with CLOS 1 and core 1 to CLOS 2.

5.4.3 Monitoring with the OS Interface

Every feature mentioned above can also be executed with the OS interface. Monitoring with the OS interface is similar to the previous. One of the notable differences is that classes of service (CLOS) are bound to process IDs (PIDs) instead of sockets/cores. To illustrate an example of what monitoring with the OS interface looks like, the following command can be executed – pqos -I -p all:116,119-121 –, and it would monitor with the OS interface (`-I` option), for process IDs (`-p` option), all kind of events (`all` keyword) on processes with PID 116, and 119 to 121. The output of such command could look like the following:

```
PID   CORE    IPC   MISSES    LLC[KB]  MBL[MB/s]  MBR[MB/s]
116    N/A   1.82    2140k     1344.0        1.9        0.0
119    N/A   1.29     288k      704.0        2.2        0.0
120    N/A   2.16     657k      256.0        9.6        0.0
121    N/A   1.27     718k      192.0       63.4        0.0
```

6 | Tools & Methodology

In this chapter we aim to briefly describe all the tools left unmentioned in previous chapters as well as the methodology we adopted when developing this project.

6.1 Tools

In this section, we introduce some of the more frequently used tools in this thesis. Only a select few of these tools are briefly discussed, however. These are the Mercurium compiler, and two tools used for analysis and evaluations, Paraver and Extrae.

The explanations of OmpSs-2 and Nanos6 are omitted, as they've already been introduced in chapter 3 and chapter 4 respectively.

6.1.1 Mercurium

Mercurium [35] is a source-to-source compilation infrastructure aimed at fast prototyping. Currently supported languages are C, C++ and, Fortran. Mercurium is mainly used in the Nanos environment to implement OpenMP, but since it is quite extensible, it has been used to implement other programming models or compiler transformations. Examples include Cell Superscalar, Software Transactional Memory, Distributed Shared Memory or the ACOTES project, to name a few.

Extending Mercurium is achieved using a plugin architecture, where plugins represent several phases of the compiler. These plugins are written in C++ and dynamically loaded by the compiler according to the chosen configuration. Code transformations can be implemented in terms of source code (there is no need to modify or know the internal syntactic representation of the compiler).

6.1.2 Paraver

Paraver [36], a tool developed by Barcelona Supercomputing Center, is a very flexible data browser that is part of the CEPBA-Tools toolkit. Its analytical power is based on two main pillars.

First, its trace format has no semantics; extending the tool to support new performance data or new programming models requires no changes to the visualizer, just to capture such data in a Paraver trace.

Secondly, metrics are not hardwired on the tool; they are programmed. To compute them, the tool offers a large set of time functions, a filter module, and a mechanism to combine two timelines. This approach allows displaying a vast number of metrics with the available data.

To capture the expert's knowledge, any view or set of views can be saved as a Paraver configuration file. After that, re-computing the view with new data is as simple as loading the saved file. The tool has been demonstrated to be very useful for performance analysis studies, giving much more details about the application's behaviour than most performance tools.

Paraver is not tied to any programming model as long as the model can be mapped in the three levels of parallelism expressed in the Paraver trace. An example of two-level parallelism would be hybrid MPI + OpenMP applications. The runtime measurement system Extrae that generates Paraver traces currently supports programming interfaces such as MPI, OpenMP, pthreads, OmpSs and CUDA.

6.1.3 Extrae

Extrae is a package devoted to generating Paraver trace-files for post-mortem analysis. Extrae, a tool also developed by Barcelona Supercomputing Center, uses different interposition mechanisms to inject probes into target applications to gather information regarding performance.

Extrae does not only offer the possibility to instrument application codes but also provides sampling mechanisms to gather performance data. While adding monitors into specific locations of the application produces insight which can be easily correlated with source code, the resolution of such data is directly related with the application control flow.

Currently, Extrae sports two different sampling mechanisms. The first mechanism are the old-fashioned signal timers, which fire the sampling handler at specified time intervals. The second sampling mechanism uses processor performance counters to shoot the sampling handler at a specified range of events interval.

While the first mechanism can provide uncorrelated samples with the application code, the second mechanism, using the appropriate performance counters, can give insight of the application but still present some correlation with the application code/performance.

6.2 Methodology

When it comes to methodology, I believe there is no such thing as a miracle plan that works in any case. I think companies should offer tools so that employees follow in a generical manner a particular methodology or set of methodologies, always tweaking them to refine them.

For this project, a combination of the conventional scientific research methodology with the agile methodology used in development is used.

6.2.1 Research Methodology

The common research methodology follows a ”number-of-steps” plan which can contain more or fewer steps. The general view, however, consists in:

- **Rationale**: Finding a rationale behind the study to be conducted.
- **Problem**: Define the problem. Describe what the problem is, why it is a problem, and the reasons why it is worth solving.
- **Research Objectives**: Define the objectives of the research towards solving the problem. Often, this includes a study of related work if existent. In this study, researchers must be convinced of the flaws of related work and that their research, if successful, will end up generating a better solution to the problem.
- **Research Hypothesis & Design**: This step consists in creating a hypothesis and begin designing the plan towards finding the aforementioned solution.
- **Testing**: Once the plan has been set and followed, testing or experiments must be done.
- **Evaluation**: Collect data, analyze the results and consider the limitations and future work of the study.

On its own, the methodology mentioned above would be chaotic for the computer science field, as it is a method that takes a long time to complete and inconveniences such as bugs would ruin it, so it was combined with the methodology explained next.

6.2.2 Agile Methodology

Agile software development is an approach to software development under which requirements and solutions evolve through the collaborative effort of self-organizing and cross-functional teams and their end users. It advocates adaptive planning, evolutionary development, early delivery, and continual improvement, and it encourages rapid and flexible response to change.

The term agile (sometimes written Agile) was popularized, in this context, by the Manifesto for Agile Software Development [37]. The values and principles discussed in this manifesto were derived from a broad range of software development frameworks, including Scrum and Kanban.

There is significant anecdotal evidence that adopting agile practices and values improves the agility of software professionals, teams and organizations; however, some empirical studies have found no scientific evidence.

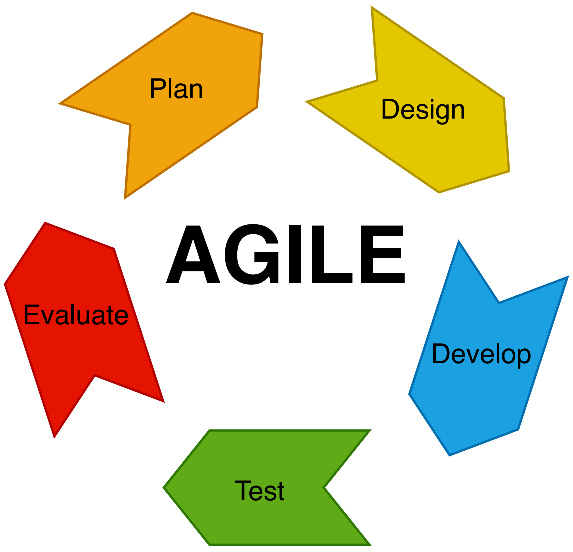


Fig. 6.1: A graphical representation of the steps in Agile

Figure 6.1 shows a graphical representation of the steps in the agile methodology that were followed. Typically, there is an extra step called the *release* phase, in which the part of the project that was worked on gets released to the end-users. In this work, the agile methodology was introduced in each step of the scientific research methodology.

To put it in simpler words, first, the existing problem and its flaws are discussed and developed into – manual tuning. Then a solution was theorized – automatically tuning applications through enhanced scheduling policies. To develop this solution, a plan that divided the work into several phases of the project was created. Then, in each of this phases, Agile development was introduced – first the phase was planned, then it was designed, developed, and tested, and finally, the phase was evaluated on its own before moving on to the next phase. This was accompanied by bi-weekly meetings to keep our focus on track with the global scope of the project.

7 | The Monitoring Infrastructure

The first phase of this thesis consists in extending OmpSs-2, more specifically the Nanos6 runtime, with a lightweight monitoring infrastructure. Throughout the following sections, we describe everything about this infrastructure or module.

7.1 Enhancing Profiling in Nanos6

Currently, Nanos6 includes a variant called the `profile` variant. This version of the runtime, as explained in section 4.3, has plenty of counters that profile the basics of the runtime, from tasks to threads.

Since this variant is a complete profiling of every internal feature of the runtime, it introduces substantial overhead. As this version should not be used for performance purposes, the overhead introduced is not a significant problem. This is because this variant should only be used for information-gathering purposes, just like the `verbose` or `debug` variants.

Nanos6 was developed in a way that allows it to be extensible with plugins. In other words, developing a module which is independent of everything else in the runtime is relatively simple. The current instrumentation or profiling modules that Nanos6 presents can be considered plugins. They use static function calls in certain locations of the core of the runtime, in order to instrument it. This is visualized in figure 7.1.

Due to the extensibility it presents, our chosen strategy is to create a new monitoring module that pinpoints the critical features of the runtime just the way the current profiling does. That is, by introducing static independent procedure calls in specific locations of the core of the runtime. However, since we aim to enhance scheduling policies in Nanos6, monitoring runtime features in a lightweight manner must be the primary objective. Using the profiling existing in Nanos6 however is not a viable option, as it introduces overhead and it monitors more features than the ones needed in this work.

Apart from the issues mentioned above, one of our other goals is extending monitoring to integrate the usage of Intel's® PQoS library. Because of this, a new module tailored to our needs must be created.

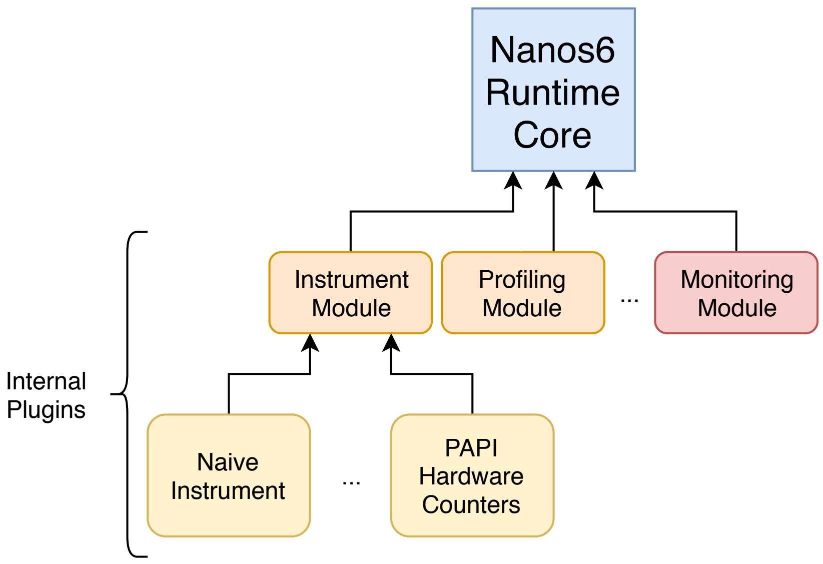


Fig. 7.1: Monitoring and other modules shown as internal plugins of Nanos6

7.2 Previous Monitoring Module

In our previous work [22] a monitoring module had to be implemented as well. We conducted a study to measure the overhead of obtaining timing measures with several counters, however it was not complete. We gathered information from several sources, tested all of them and used the best one.

Since this project had to start over with a new monitoring module which takes into account every element of this thesis, and that is extensible itself, we conducted a study of timing measures yet again. This is further explained in the next section, where we discuss every detail of the new improved module.

7.3 Improved & Lightweight Monitoring Module

In this section, we describe every detail of the new monitoring module. As mentioned above, a study to choose between different options to record time had to be conducted. The study began by picking up where the previous one left off and by adding several other sources. Since one of the most important requirements is for our module to measure metrics in a lightweight way, first we studied the pros and cons of every available option.

7.3.1 Choosing Timers

Listed next is every option we studied with the most remarkable features and issues they present.

- **C - <sys/times.h>**: This library provides fairly simple usage by instantiating `clock_t` structures which contain timers. Afterwards, simply using the `times()` procedure – `times(clock_t)` – records timestamps. When it comes to precision, it records user time through system calls, up to milliseconds. Due to using system calls and having low precision, we discarded this option.

- **C - <sys/time.h>**: Also using the same library, it is possible to record timing using the `getrusage()` function. This option measures not only time between calls, but also memory, I/O, CPU, and thread metrics. In precision, it exceeds the previous one, as it records up to microseconds. Nonetheless, since it records lots of metrics, the overhead added is excessive, so we discarded this option as well.

- **C - <ctime>**: With simple calls to `std::clock()` to start and stop timers, this option gives an estimation of timing with precision up to seconds, which made us discard this option even though the overhead was negligible.

- **C - <time.h>**: The usage of this option is fairly simple as well. It consists in calling the `clock_gettime(id, struct)` function, where `id` refers to the type of timers and `struct` refers to the timer itself. We also discarded this option since it measured more metrics than just timing, which added unnecessary overhead. This option had a precision of up to nanoseconds, which is more precision than all the previous options had.

- **C++11 - <chrono>**: Using this library is less simple than other options. A lot of casting between types has to be done. However, there is barely any overhead since every metric has its own calls and they can be combined as desired. Precision can also be tweaked as desired as well, since it supports different precisions (nanoseconds, microseconds, etc.).

- **C/C++ - Hardware-based counters**: The FFTW library for computing Fourier transforms [38] includes a file called `cycle.h`. This file contains definitions of timers depending on the architecture. The precision is matchless, as it measures time using ticks. The usage is fairly simple as well since every architecture has its own definitions and they all match naming-wise. Due to this, it is the option that seemed the most reasonable, as it introduces little to no overhead. This is because it uses assembler instructions that read special registers.

Our decision then, biased by overhead introduced into measuring, ease of use, and precision, is to use two of the options mentioned above to complement each other. The first is **hardware-based counters** as it has both the best precision and less overhead. However, the second type of timers `std::chrono` are used as an alternative as well. This is mainly due to two reasons:

Firstly, we need counters able to work on any existing architecture. When it is detected that the appropriate definitions for an architecture are not available, structs switch to the alternative. This alternative can also be used when in a specific architecture there is no permission to access timing registers, which would disable hardware-based timers.

Secondly, hardware-based timers use CPU ticks (frequency clocks) to measure time. These recorded 'ticks' must be converted to time, which is done through measuring how many ticks are counted between a certain time range. To do this, another type of timer must be used. For this purpose, two types of timers are used.

Clock Synchronization

Nowadays processors include functionalities to vary their frequency on demand. These are dynamic-frequency CPUs, and the timers used in this work must adapt to these types of processors as well, since the approach taken must be independent from architectures.

To adapt to these needs, a synchronization mechanism must be implemented. This mechanism starts up as soon as the runtime initializes. While the mechanism starts up and until a tick-to-time ratio is obtained, the second type of timers – `std::chrono` – are used. When the runtime initializes, a service is created. This service implements the mechanism, called `TickConversionUpdater`, which fires every certain amount of time. This mechanism has two phases:

The first phase is shown in snippet 7.1. The updating service contains one of the alternative timers, named `_t1`, which measures the time between phases. First, a boolean is set to true to mark that an update has started, so that no more services can be started and collide with the current update. Then, the hardware-based timer – `_c1` – is restarted in case any leftover data from previous updates were present. This only happens in further updates to keep the tick-to-time ratio in sync. After that, both timers start measuring time.

```
inline void beginUpdate()
{
      _updateStarted = true;
      _c1.restart();
      _t1.start();
      _c1.start();
}
```

Code 7.1: Start phase of the tick-to-time updating service

The second phase is shown in snippet 7.2. First, both timers are stopped. Then, we calculate and update the ratio between clock ticks and seconds. Lastly, we mark the update as completed with the same boolean that we marked as true in the first phase. Once the update is completed, the hardware-based timers can be used as the runtime timers. The first synchronization happens within nanoseconds of the start of the runtime initialization. This means by the time timers are needed for monitoring, hardware-based ones can be used.

```
inline void finishUpdate()
{
      _c1.stop();
      _t1.stop();
      double rate = ((double) _t1) / _c1.getAccumulated();
      ChronoArchDependentData::_tickConversionFactor = rate;
      _updateStarted = false;
}
```

Code 7.2: Ending phase of the tick-to-time updating service

7.3.2 Monitoring Locations

Current modules in Nanos6, such as the profiling module, insert their calls in various locations of the runtime. Before these modules existed, developers had to study the key locations in the runtime core where these calls could be inserted. The most common locations are those in which tasks, threads, and other elements that constitute the runtime, suffer an important change in their internal status. Calls to the profiling module API are already in perfect locations, which can be reutilized by our new monitoring module.

```
void WorkerThread::handleTask(CPU *cpu)
{
      _task->setThread(this);
      ...
      Instrument::startTask(taskId);
      Instrument::taskIsExecuting(taskId);

      Monitoring::taskChangedStatus(...);

      // Run the task
      std::atomic_thread_fence(std::memory_order_acquire);
      _task->body(...);
      ...
}
```

Code 7.3: Location of profiling and monitoring calls when a task starts execution

Snippet 7.3 shows a fragment of the code of a worker thread when it is about to execute a task. When that happens, right before the execution, the profiling module inserts a call recording the change of status. The monitoring module should imitate this behavior, which is why it is shown right after the Instrument module's calls. For ease of reading, unrelated pieces of code have been replaced with an ellipsis.

Not all calls to profiling must be replicated, however. Changes such as tasks entering 'zombie' timing – when a task is completed but not finalized – are unnecessary for the monitoring module, and would only add unwanted overhead. Figure 7.2 illustrates all the changes which tasks might go through in the runtime. These are just the desired changes to measure with the monitoring module.

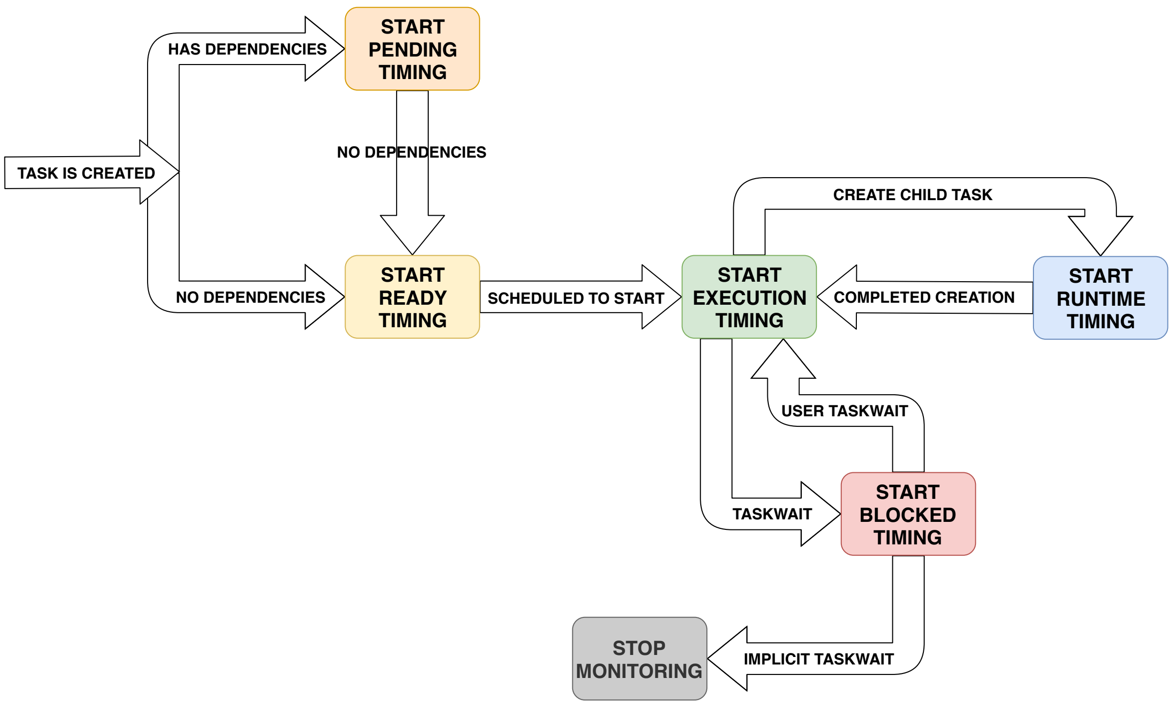


Fig. 7.2: Representation of the lifetime of tasks in Nanos6 through status changes

Tasks start in the pending or ready status, depending on whether they have unmet dependences. Once the scheduler assigns the task to a thread, the status becomes executing. While executing, a task can change to the runtime status if it must create a child task, or blocked if it must due to an explicit user-defined taskwait. Once the task completes due to the implicit end-of-task taskwait, monitoring ceases for that task.

For threads, there are only two status. The first one is idle. Threads begin as idle, and when they poll the scheduler for tasks, two events may happen. In the first one, the scheduler returns a valid task, which switches the status into the second one, executing. If the scheduler has no work available for the thread, this one remains idle. A thread only switches to the idle status when it is executing and no more tasks are available.

In the next sections we fully describe every feature and metric that the monitoring module measures. We give a thorough description of every metric, which includes why the metric must be monitored and how it is monitored. Afterward, we display an example of the output of monitoring, showing the potential it could provide to any runtime. An extensive evaluation of the overhead and the usefulness is discussed later in chapter 10. Each metric is described by organizing them into four levels of abstraction: runtime metrics, CPU metrics, thread metrics and task metrics. We explain each of these metrics in their respective sections.

7.3.3 Task Metrics

To make any sort of timing predictions about tasks, the monitoring module must be cappable of providing precise timing metrics concerning every task status shown in figure 7.2. For that purpose, in each task, timer objects are allocated for every possible task status. For simplicity when making predictions, these timing metrics are also accumulated into two kind of groups. The first is per-task status. Then, they are also aggregated in a per-tasktype basis. In other words, regarding task metrics, so far monitoring includes:

- **For every task instance, a timer per each task status**. That translates to 5 timers, one for the ready status, another for the pending status, and so on until the blocked status.
- Also **for every task instance, a timer per each task status** accounting the aggregated timing of children tasks. That translates to 5 timers as well, one for the accumulated ready time of children, another for the accumulated executing time, etc.
- Even though this does not pertain to the level of abstraction of tasks but types of tasks, the aforementioned task timers are aggregated into task-type counters. That means, for **every type of task**, a counter specifies the **amount of elapsed time** that pertains to that specific type of task.
- Related to the previous one, the **number of instances** of a certain **type of task** is also monitored.

7.3.4 Thread Metrics

When it comes to threads, holding two timers per thread recording idle and executing time is sufficient to make timing predictions. Apart from this, however, the integration of Intel's PQoS® library must be done at a threading level, since monitoring on a per-core basis is still under development as specified in their latest release.

Even though our integration is at a threading level, since threads are the elements that integrate PQoS, we offload data in tasks. That means that, when a task is bound to a thread, this task flags the thread asking for its events values, and the thread polls the data using PQoS' API. When the task stops execution, the task flags the thread again. This keeps happening until tasks reach completion. Next, we explain how the integration of PQoS was carried, and which kind of metrics are stored.

Intel® PQoS Integration

Similarly to other monitoring metrics, PQoS event calls are measured when threads suffer a change in status. These are the scenarios in which PQoS events are polled:

- When a thread begins executing a task or the task resumes execution because it was previously paused. At this point, the task will poll PQoS events from the thread which, at its turn, will poll hardware events using Intel's® PQoS API. The task will then gain its starting values for all events.
- When a task must be blocked and thus PQoS event monitoring must stop, or when it finalizes execution and therefore PQoS event monitoring must end. In this scenario, the task will poll its events values right before being blocked or finalized. Then, the paused event values will be subtracted from the initial event values, which will result in the accumulated values between the interval time. This accumulated value is stored in the task's structures.

Section 7.3.8 shows the API that tasks use to poll events from threads. Next, we explain which and how PQoS events are stored in tasks. Firstly, per-task, we store the following metrics:

- The **elapsed time** of the task, obtained from monitoring. This is to swiftly obtain a task's timing, to then compute combined metrics such as an event's value over time.
- A collection of snapshot-like events. Currently, per task, **we store only last level cache usage** (`llc_usage`) as a snapshot event. A snapshot event behaves as a hardware event that needs its average values to be computed using all measurements. That means that all the values must be accumulated on its own as every time a thread polls the hardware event, it is reset back to 0.
- A collection of accumulated events. An accumulated event behaves as a hardware event which does not lose information by polling. In other words, we must ensure the correct computation of the accumulated value of these events by measuring start and stop values and computing the difference. Currently, per task, **we store four kinds of these events**:
  - The memory bandwidth of the local node – `mbm_local_delta`.
  - The number of last level cache misses – `llc_misses_delta`.
  - The number of retired instructions – `ipc_retired_delta`.
  - The number of unhalted clock cycles – `ipc_unhalted_delta`.

  These last two are not used on their own, they are combined to form the Instructions Per Cycle event. Also by combining the retired instruction with the last level cache misses, we obtain the last level cache miss ratio.

All the events mentioned above form the collection of four events, which names are self-explanatory – `llc_usage`, `ipc`, `mem_bandwidth`, and `llc_miss_rate`. To easily predict values, we also aggregate these events in a per-tasktype basis once tasks finish their execution. That translates to having these four kinds of events replicated and accumulated into every type of task, along with the number of instances used in the aggregation.

7.3.5 CPU Metrics

CPU metrics only include timers to measure the percentage of idle time on each core individually. Handling the measurement of these metrics is fairly simple, as it only requires inserting start & stop calls whenever a CPU is put to sleep or woken up. Concerning CPU metrics then, monitoring includes:

- **For every CPU, two timers**, one to measure idle timing and a second one to measure active timing.

7.3.6 Runtime Metrics

To correctly predict runtime features, information from lower levels of abstraction must be combined. For this purpose, structs hold information about:

- The average load of the runtime with respect to the number of ready or executing tasks. This is the currently available workload of the system.
- The average load of the runtime with respect to the number of microseconds executing/to be executed.
- The aggregated elapsed execution time of tasks that have finalized their execution.
- The number of task instances taken into account for each runtime load. Chapter 8 further discusses what `runtime loads` are, why these metrics are needed, and what they are used for.

As mentioned above, the fourth metric is explained in the next chapter. The third metric is merely the accumulated elapsed time that has already been executed. The first and second metrics relate to the average loads in number of tasks or time, both over specific intervals of time. Currently, monitoring includes four metrics of this kind, however, more could be computed as explained next.

- `average_load_10` – The average load of the runtime in number of tasks, averaging intervals of 10 seconds.
- `average_load_60` – The average load of the runtime in number of tasks, averaging intervals of 60 seconds.
- `average_timing_load_10` – The average load of the runtime in microseconds executed, averaging intervals of 10 seconds.
- `average_timing_load_60` – The average load of the runtime in microseconds executed, averaging intervals of 60 seconds.

The two first metrics give an abstract view about the number of tasks executed over a range of time. This translates to the size of the internal task queues. These are only used in the reports given as an output. They are similar to Linux' average loads given with the `top` command. The two last metrics however, approximate the same counters using time, which provides a clearer vision of the amount of workload executed in the past.

Linux Load Averages

The `top` command is an addition to the UNIX set of commands that ranks processes according to the amount of CPU time they consume. It produces outputs such as the following:

```
top - 10:31:53 up  1:29,  3 users,  load average: 0.45, 0.42, 0.33
Tasks: 223 total,   1 running, 222 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.7 us,  0.5 sy,  0.0 ni, 96.5 id,  0.2 wa,   ...
KiB Mem : 16306604 total, 11040148 free,  3069464 used, ...
...

  PID USER      PR  NI ... S  %CPU %MEM     TIME+ COMMAND
  524 avahi     20   0 ... S   6.7  0.0   0:00.27 avahi-daem+
 2199 anavarr1  20   0 ... S   6.7  2.2   1:25.61 chromium
 3282 anavarr1  20   0 ... S   6.7  2.2   2:14.14 chromium
 4116 anavarr1  20   0 ... R   6.7  0.0   0:00.01 top
    1 root      20   0 ... S   0.0  0.0   0:00.84 systemd
  ...
```

In each of these commands, there are three numbers reported as part of the `load average` output. Quite commonly, these numbers show a descending order from left to right. Combining several sources [39–41], the definition of this metric could be one like the following:

The load average tries to measure the number of active processes at any time. As a measure of CPU utilization, the load average is simplistic, poorly defined, but far from useless [40](page 726).

(...) high load averages usually mean that the system is being used heavily and the response time is correspondingly slow [40](page 720).

The load average is the sum of the run queue length and the number of jobs currently running on the CPUs [41](page 97).

The load average metrics are not the usual kind of averages. They are time-dependent averages. Not only that, but they are damped time-dependent averages [39]. Next is a thorough explanation of how these are computed.

The Linux kernel's code includes the procedure shown in snippet 7.4. The `LOAD_FREQ`, or the frequency by which loads are updated, equals 5 HZ. If 1 HZ is 100 clock ticks, and 1 tick equals 10 milliseconds, that means that CALC_LOAD is executed every 5 seconds.

CALC_LOAD is a macro defined in another file of the Linux kernel called sched.h, and it is shown in snippet 7.5.

```c
static inline void calc_load(unsigned long ticks)
{
      unsigned long active_tasks; /* fixed-point */
      static int count = LOAD_FREQ; /* 5 HZ */

      count -= ticks;
      if (count < 0) {
            count += LOAD_FREQ;
            active_tasks = count_active_tasks();
            CALC_LOAD(avenrun[0], EXP_1, active_tasks);
            CALC_LOAD(avenrun[1], EXP_5, active_tasks);
            CALC_LOAD(avenrun[2], EXP_15, active_tasks);
      }
}
```

Code 7.4: calc_load function from the Linux kernel

```c
extern unsigned long avenrun[]; /* Load averages */

#define FSHIFT 11 /* nr of bits of precision */
#define FIXED_1 (1<<FSHIFT) /* 1.0 as fixed-point */
#define LOAD_FREQ (5*HZ) /* 5 sec intervals */
#define EXP_1 1884 /* 1/exp(5sec/1min) as fixed-point */
#define EXP_5 2014 /* 1/exp(5sec/5min) */
#define EXP_15 2037 /* 1/exp(5sec/15min) */

#define CALC_LOAD(load,exp,n) \
      load *= exp; \
      load += n*(FIXED_1-exp); \
      load >>= FSHIFT;
```

Code 7.5: CALC_LOAD macro from the Linux kernel

What this chaotic code tries to do is calculate load averages in fixed-point representation, to avoid costly floating-point operations. Numbers such as 1884, 2014 and 2037 are constants used to fake the fixed-point representation. Using the 1 minute sampling as an example, the conversion of exp(5/60) (since the averages needed are of 1 minute, or 60 seconds, and in 5 second intervals) into base-2 with 11 bits of precision, occurs like so:

$$e^{\frac{5}{60}} = \frac{e^{\frac{5}{60}}}{2^{11}}$$

However, EXP_N represents the inverse function (exp(-5/60)) therefore these numbers can be computed using the following expression:

$$EXP\_N = \frac{2^M}{2^{\frac{f*log2(e)}{N}}}$$

where M is the number of precision bits, f is the frequency of update in seconds, and N is the number of seconds to average.

Computing Load Averages

To sum up, the three load average numbers provided by top in Linux intend to provide some information about how much work has been done on the system in the recent past (1 minute), the past (5 minutes) and the distant past (15 minutes). However, some issues remain:

- The load is not an utilization metric but queue length.
- They are samples of three different time series.
- They are exponentially moving averages. This will be mentioned in later sections like section 8.

Once cleared, all that is left to know is how to convert the aforementioned expressions to compute monitoring load averages. For this purpose, since the module specifies averages of the last 10 and 60 seconds, 15 bits of precision and 0.01 seconds of interval between measures, the expressions become:

$$EXP\_10 = \frac{2^{15}}{2^{\frac{0.01*log2(e)}{10}}} \quad EXP\_60 = \frac{2^{15}}{2^{\frac{0.01*log2(e)}{60}}}$$

Which once computed, become the following constants inserted in the monitoring code:

```
#define UPDATE_FREQ 0.01 // 10 ms intervals
#define EXP_10 32735     // 1/exp(0.01s/10s) as fixed-point
#define EXP_60 32763     // 1/exp(0.01s/60s) as fixed-point
```

Using these constants and functions such as the ones aforementioned in snippets 7.4 and 7.5, precise load averages are computed in a lightweight manner, without the need for floating point operations. All the needed code is collapsed into one single procedure that is used as a service to update monitoring loads every specific amount of time (frequency of update). Pseudocode of this service can be seen in pseudocode 7.1.

Algorithm 7.1 Pseudocode of the service that updates monitoring average loads

```
if must_update then
    for all average_loads as load[i] do
        activeTasks ⇐ getActiveTasks(ready + executing);
        load[i] ⇐ load[i] × EXP_M[i];
        load[i] ⇐ load[i] + (activeTasks × (FIXED_1 - EXP_M[i]));
        load[i] ⇐ load[i] >> FSHIFT;
    end for
    for all average_timing_loads as load[i] do
        elapsed ⇐ getElapsedTime();
        load[i] ⇐ load[i] × EXP_M[i];
        load[i] ⇐ load[i] + (elapsed × (FIXED_1 - EXP_M[i]));
        load[i] ⇐ load[i] >> FSHIFT;
    end for
end if
```

7.3.7 Monitoring API

Snippet 7.6 shows the API that the whole monitoring infrastructure uses to communicate with the runtime. The purpose and actions of every function are detailed within the snippet. For simplicity, we have omitted getters and setters. As written, every action to perform regarding CPU, runtime, and task metrics, is found in or propagated by task-targetting functions such as `taskCreated()`, `taskChangedStatus()` or `taskFinished()`. Functions are described using the `brief`, `param` and `return` labels in Doxygen-styled comments. The actions taken are discussed using the `actions` label.

```
//****************
//** MONITORING **

//! \brief Initialization of monitoring
static void initialize();

//! \brief Destroy the monitoring module
static void shutdown();

//! \brief Print a report of all the CPU statistics
static void displayCPUStatistics();

//! \brief Print a report of the global runtime statistics
static void displayRuntimeStatistics();

//! \brief Print a report of task timing information
static void displayTaskStatistics();

//**************************
//** TASKS, CPUS & RUNTIME **
```

```
//! \brief Gather basic information about a task when it is created
//! \actions Propagate task creation to PQoS
//! \actions Switch to the appropriate timing status
//! \actions Switch timing to the appropriate runtime load
//! \param[in] task The task to gather information about
static void taskCreated(Task * task);

//! \brief Propagate monitoring operations after a task has changed its
//! execution status
//! \actions Propagate task status change to PQoS (only in some scenarios)
//! \actions Switch to the appropriate timing status
//! \actions Switch timing to the appropriate runtime load
//! \param[in] task The task that's changing status
//! \param[in] execStatus The new execution status of the task
//! \param[in] cpu The cpu onto which a thread is running the task
static void taskChangedStatus(Task * task, int execStatus, ComputePlace *
    cpu = nullptr);

//! \brief Propagate monitoring operations after a task has finished
//! \actions Propagate task finalization to PQoS
//! \actions Stop and accumulate timing statistics
//! \actions Subtract timing from the current runtime load
//! \param[in] task The task that has finished
//! \param[in] cpu The cpu onto which a thread was running the task
static void taskFinished(Task * task, ComputePlace * cpu);


//*************
//** THREADS **

//! \brief Propagate monitoring operations when a thread is initialized
//! \actions Begin/resume timing for the thread
//! \actions Propagate the call to the PQoS API when available
static void initializeThread();

//! \brief Propagate monitoring operations when a thread is shutdown
//! \actions Pause/finish timing for the thread
//! \actions Propagate the call to the PQoS API when available
static void shutdownThread();
```

Code 7.6: Monitoring Infrastructure's API

7.3.8 PQoS API

Since PQoS is integrated as a submodule of the infrastructure of monitoring, its calls are bound to the ones shown in the previous section. This means that when monitoring, specific scenarios rise on which a change asks for an update in hardware events. If those scenarios show up, the integrated PQoS API, shown in snippet 7.7 is called through the API shown above.

```
//**********
//** PQOS **

//! \brief Initialization of PQoS
static void initialize();

//! \brief Destruction of the PQoS library
static void shutdown();

//! \brief Prints a report of PQoS information from tasks
static void displayTasksSummary();

//***********
//** TASKS **

//! \brief Predict metrics and gather information about a task
//! \actions Initialize PQoS structures
//! \param[in] task The task to predict metrics for
static void taskCreated(Task * task);

//! \brief Start monitoring PQoS events for a task
//! \actions Gather starting or resuming PQoS event values
//! \param[in] task The task to start PQoS for
static void startPQoS(Task * task);

//! \brief Stop monitoring PQoS events for a task and accumulate
//! events from its threads into the task's statistics structures
//! \actions Gather pausing or stopping PQoS event values
//! \param[in] task The task to stop PQoS for
static void stopPQoS(Task * task);

//! \brief Finish monitoring PQoS events for a task and accumulate
//! the events into accumulators
//! \actions Accumulate PQoS events into global structures
//! \param[in] task The task that has finished
static void taskFinished(Task * task);

//*************
//** THREADS **

//! \brief Initialization of PQoS for the current thread
static void initializeThread();

//! \brief Shutdown of PQoS for the current thread
```

```
static void shutdownThread();
```

Code 7.7: API of the integration of PQoS within the Monitoring Infrastructure

7.3.9 Output Statistics

In this section we show a real example of an output obtained from executing Cholesky on the SSF machine. Every application and machine is thoroughly documented in section 10. In this execution, we used a 56 core processor with 2 sockets. The size of the matrixes for the factorization was 32768 * 32768, and the block size was 512 * 512. Next we explain every part of the output that the monitoring module provides. Since this example is a large one with a lot of task types, we have decided to omit tasks such as `flat2tile`, the `main` task and other initialization tasks, which only had a single task instance and therefore did not provide much insight.

```
---------------------
STATS PQOS TASK-TYPE (INSTANCES) gemm (41664)
STATS PQOS LLC Usage (KB)        ACC / SUM / AVG / STDEV   72.23% / 393.77197    / 0.00945 / 0.04771
STATS PQOS IPC                   ACC / SUM / AVG / STDEV   79.42% / 54835.61397  / 1.31614 / 0.14963
STATS PQOS Local Mem BW (KB/ms)  ACC / SUM / AVG / STDEV   99.52% / 179787.50802 / 4.31518 / 72.3497
STATS PQOS LLC Miss Rate         ACC / SUM / AVG / STDEV   88.08% / 155.46249    / 0.00373 / 0.00031
---------------------
STATS PQOS TASK-TYPE (INSTANCES) potrf (64)
STATS PQOS LLC Usage (KB)        ACC / SUM / AVG / STDEV   69.84% / 0.80078      / 0.01251 / 0.04720
STATS PQOS IPC                   ACC / SUM / AVG / STDEV   79.69% / 53.34096     / 0.83345 / 0.22895
STATS PQOS Local Mem BW (KB/ms)  ACC / SUM / AVG / STDEV  100.00% / 0.00000      / 0.00000 / 0.00000
STATS PQOS LLC Miss Rate         ACC / SUM / AVG / STDEV    7.47% / 0.11995      / 0.00187 / 0.00070
---------------------
STATS PQOS TASK-TYPE (INSTANCES) syrk (2016)
STATS PQOS LLC Usage (KB)        ACC / SUM / AVG / STDEV   75.94% / 26.02148     / 0.01291 / 0.05575
STATS PQOS IPC                   ACC / SUM / AVG / STDEV   72.52% / 2526.11641   / 1.25303 / 0.22589
STATS PQOS Local Mem BW (KB/ms)  ACC / SUM / AVG / STDEV   99.75% / 4380.99185   / 2.17311 / 45.5738
STATS PQOS LLC Miss Rate         ACC / SUM / AVG / STDEV   71.73% / 8.38135      / 0.00416 / 0.00056
---------------------
STATS PQOS TASK-TYPE (INSTANCES) trsm (2016)
STATS PQOS LLC Usage (KB)        ACC / SUM / AVG / STDEV   71.89% / 23.71289     / 0.01176 / 0.05453
STATS PQOS IPC                   ACC / SUM / AVG / STDEV   74.69% / 2657.42096   / 1.31817 / 0.27264
STATS PQOS Local Mem BW (KB/ms)  ACC / SUM / AVG / STDEV   99.51% / 8111.55847   / 4.02359 / 68.5341
STATS PQOS LLC Miss Rate         ACC / SUM / AVG / STDEV   42.20% / 7.88417      / 0.00391 / 0.00073
---------------------
STATS PQOS ALL TASKS
STATS PQOS LLC Usage (KB)        SUM / AVG    444.50228       /   NA
STATS PQOS IPC                   SUM / AVG    60074.55225     /   NA
STATS PQOS Local Mem BW (KB/ms)  SUM / AVG    192280.05847    /   NA
STATS PQOS LLC Miss Rate         SUM / AVG    171.90376       /   NA
---------------------
```

The piece of output above shows a summary of PQoS events per type of task. It shows statistics such as the average and standard deviation of each event, the sum of events from all tasks of a certain task and the accuracy of predictions, which we discuss in further chapters. Also, a summary aggregating all tasks is shown at the very bottom. The amount of tasks of each type is shown right after every type of task's label.

```
+-----------------------------+
|       TASK STATISTICS       |
+-----------------------------+

STATS MONITORING TASK-TYPE (INSTANCES)    gemm (41664)
STATS MONITORING UNITARY COST AVG / STDEV 0.00009    / 0.00003
STATS MONITORING PREDICTION ACCURACY (%)  96.51%
STATS MONITORING AVERAGE PARALLELISM      8.86988
+-------------------------+
STATS MONITORING TASK-TYPE (INSTANCES)    potrf (64)
STATS MONITORING UNITARY COST AVG / STDEV 0.00006    / 0.00002
STATS MONITORING PREDICTION ACCURACY (%)  84.98%
STATS MONITORING AVERAGE PARALLELISM      0.03928
+-------------------------+
STATS MONITORING TASK-TYPE (INSTANCES)    syrk (2016)
STATS MONITORING UNITARY COST AVG / STDEV 0.00004    / 0.00002
STATS MONITORING PREDICTION ACCURACY (%)  74.01%
STATS MONITORING AVERAGE PARALLELISM      0.91271
+-------------------------+
STATS MONITORING TASK-TYPE (INSTANCES)    trsm (2016)
STATS MONITORING UNITARY COST AVG / STDEV 0.08984    / 0.02393
STATS MONITORING PREDICTION ACCURACY (%)  72.45%
STATS MONITORING AVERAGE PARALLELISM      0.83023
```

This piece of output, on the other hand, shows information about timing on a per-task type basis. Most of the metrics shown are explained in further chapters. The average parallelism stands for the number of tasks of that type executed within the execution time of the application. This is a rough approximation to the parallelism present in a certain type of task. However, this metric does not take into account dependences or intertwining between different types of tasks.

```
+-----------------------------+
|     RUNTIME LOADS (µs)      |
+-----------------------------+
Instantiated Runtime Load (45764)       896657630.00 µs
Blocked Runtime Load (0)                0.00 µs
Ready Runtime Load (0)                  0.00 µs
Executing Runtime Load (0)              0.00 µs
Finished Runtime Load (45764)           896657630.00 µs
+-----------------------------+

+-----------------------------+
|        AVERAGE LOADS        |
+-----------------------------+
Average Load (# of tasks) last 10 s     511.20
Average Load (# of tasks) last 60 s     157.39
+-----------------------------+

+-----------------------------+
|     AVERAGE TIMING LOADS    |
+-----------------------------+
Average Load (ms) for the last 10 s     865.96
Average Load (ms) for the last 60 s     135.30
+-----------------------------+

+-----------------------------+
|     GENERAL STATISTICS      |
+-----------------------------+
Total Execution Time                    55.65 (s)
+-----------------------------+
```

The last metrics shown in the output refer to runtime loads, explained in the next chapter, the aforementioned average loads in number of tasks, and the average loads in seconds. These are only in the output to show the potential of the runtime. Any of these metrics, or the ones mentioned in previous pieces of output, can be polled at any time during the execution of applications. Hence why, the instantiated and finished runtime loads have the same value both in instances and time executed.

7.3.10 Current Monitoring Structure

Figure 7.3 shows how the Monitoring module includes the underlying module of each kind of metrics. Runtime, CPU, thread, and task-related metrics. Task and thread metrics are also incorporated by the PQoS module, which, at its time, is included in the Monitoring module. Finally, each module uses the pertaining chronometer structures, which must be updated using the appropriate service.

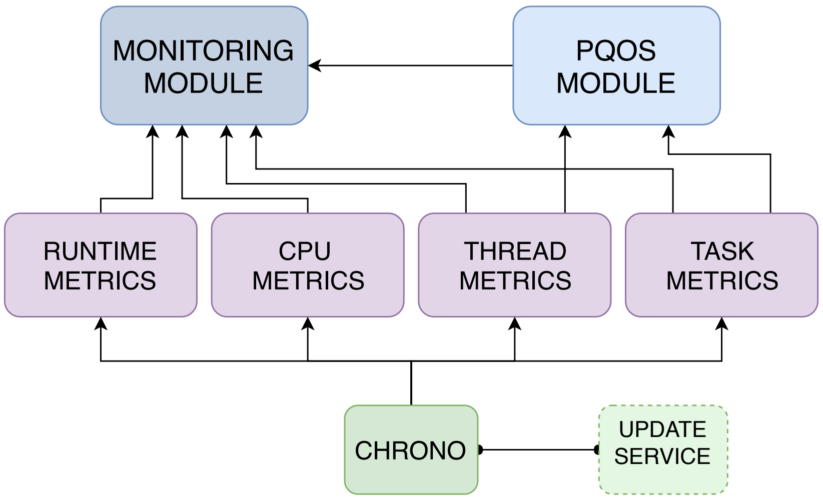


Fig. 7.3: Scheme showing the monitoring module and its underlying structure

All these modules are controlled by the environment variables listed next:

- `NANOS6_MONITORING_ENABLE`: Used to control whether monitoring is enabled. As it is a boolean, the only possible values this variable can take are 0 and 1. By default, it takes value 0.
- `NANOS6_MONITORING_VERBOSE`: Used to control whether statistics are output. It is a boolean, so the only possible values this variable can take are 0 and 1. By default, it takes value 0. This variable is only valid if monitoring is enabled. Currently, it only allows the specified values, however, it is thought to be extended so that any combination of the four different kinds of metrics can be output.
- `NANOS6_PQOS_ENABLE`: Used to control whether PQoS event monitoring is enabled. Also being a boolean, the possible values are 0 or 1. By default, it takes value 0.

- `NANOS6_PQOS_VERBOSE`: Used to control whether PQoS event statistics are output. Also being a boolean, the possible values are 0 or 1. By default, it takes value 0. This variable is only valid if PQoS monitoring is enabled.

Even though it is not possible at the time of writing, we plan to extend and further separate these modules. This is to allow only certain metrics to be monitored or output, and so that any combination of metrics can be chosen. All the metrics and APIs aforementioned in previous sections are used to make predictions, we discuss this in the next chapter.

8 | Predictions through Monitoring

With this chapter, we aim to explain how all the aforementioned metrics are used to obtain precise predictions. We begin by introducing the `cost` clause. We follow by explaining which kind of predictions we compute, how they are computed, and where they are stored. At the end of this chapter, we briefly explain how the error in predictions is computed, and why it is computed the way it is.

8.1 The Cost Clause

In a utopic scenario, every task would pertain to a group or type of tasks. In each of those types, tasks would share the same features and would be almost identical to each other in attributes such as the computational intensity or the memory operations they perform. However, this is not the case for task-based programming models. In these models, each task may be grouped with other tasks, but that does not mean they share the same granularity.

One of the best ways to group tasks is by their objective in an application. To serve as a simple example, we explain an appropriate organization of tasks by types in Mergesort. Mergesort is a recursive sorting application which we further discuss in chapter 10. In this application, we would separate tasks into three types. The first type of task would be the **initialization** tasks, which initialize array positions. The second type of tasks would be the recursive or **merge_sort** tasks, which create other tasks and recursively call themselves. The third kind would be **merge** tasks, which take two chunks of an array and merge them into a single one.

These tasks differ in computational intensity since some have linear cost and others have exponential cost. Regardless, we need a way to make predictions without implicitly knowing this. For this purpose, we introduced the `cost` clause into OmpSs-2.

The `cost` clause tries to eliminate the need of, from the runtime-side, identifying underlying features of tasks. It serves as a help or a hint given to the runtime, by application developers. We encourage developers to use this clause to point features such as the computational intensity of a task, the amount of I/O operations if the task is I/O bound, or the amount of memory transactions if it is well known and the task is memory-bound.

The relative computational weight of a task is often bound to input sizes. This is why the `cost` clause should and is in fact, parametrized. The clause accepts variables, constants, or function calls. Next, we show the syntax of this clause in the aforementioned example application. After discussing the example, we list the most remarkable features of the newly introduced clause.

```c
void merge_sort(double *a, size_t start, size_t end)
{
     if (start >= end) return;

     size_t mid = (start + end) / 2;
     size_t n1 = (mid - start) + 1;
     size_t n2 = (end - mid);

     #pragma oss task label(merge_sort) cost(n1*log2(n1))
     merge_sort(a, start, mid);

     #pragma oss task label(merge_sort) cost(n2*log2(n2))
     merge_sort(a, mid + 1, end);

     #pragma oss taskwait

     #pragma oss task label(merge) cost((end-start) + 1)
     merge(a, start, end);
}
```

Code 8.1: Snippet of an OmpSs-2 Mergesort code using the cost clause

As shown in snippet 8.1, the `merge` procedure has a linear cost. However, `merge_sort` has a logarithmic computational weight. Having this information allows us to normalize a task's cost using its elapsed execution time for future tasks. If the application has a linear or no algorithmic function, this normalization through the `cost` clause will not cause a negative effect, as we will be averaging a task's time. On the contrary, if tasks follow an algorithmic expression such as exponential, the average will be normalized and, therefore, much more truthful and precise. These are some of the most remarkable features concerning the `cost` clause:

- The clause accepts variables, symbols, and calls to functions even from external libraries.
- The cost of a task is normalized (i.e., using the task's elapsed execution time).
- This normalized or **unitary cost** is used to make predictions in a fair way.
- These can be later on used in real time, as we will discuss in chapter 9.

TASK	ALGORITHM INPUT	COST EXPRESSION	ROLLING AVERAGE(ms)	UNITARY COST(ms)
QuickSort	N = 200.000	*O(N*log(N))*	0.6306	$0.5948*10^{-12}$
	N = 1.000.000		3.5352	$0.5892*10^{-12}$
InsertionSort	M = 200.000	*O(M*M)*	184.80	$4.6202*10^{-9}$
	M = 1.000.000		4604.40	$4.6044*10^{-9}$

Table 8.1: A brief experiment using the `cost` clause to normalize time

To demonstrate the effectiveness of the cost clause, table 8.1 is presented. Tasks from the same type with similar costs may provide accurate predictions. However, tasks that differ in cost by a lot may not behave conventionally. In the table, we can see the features obtained from executing two different sorting algorithms. The second column represents the size of the input, or the number of elements to sort. The third one shows the algorithmic expression that defines the task's computational weight. The fourth one shows an average of the execution time of all tasks, and the fifth one shows the unitary or normalized cost computed by the monitoring module.

What we try to show is that, no matter the input size of the application or the algorithmic expression of tasks, the unitary cost will always provide a normalized and precise metric. However, non-normalized averages do not take into account exponential nor logarithmic growth, which is why predictions won't be as exact.

8.2 Prediction Metrics

Having the cost clause allows us to have means by which we can normalize other metrics, not only time. In this section we specify every prediction we obtain using these modules.

8.2.1 Predictions Related to Monitoring

When it comes to predictions related to the metrics we obtain from monitoring, we get three types.

Firstly, we could save the average unitary cost of every type of task to obtain predictions. When a new task would be created, all we would have to do would be computing the product of the cost of the newly created task by the average unitary cost. However, at certain points of an application, unitary costs might vary due to different scenarios. To serve as an example, a task could be using a shared cache which is being overutilized by other tasks or programs, which would cause the elapsed execution time of the task to increase, and so would the unitary cost. Due to this and other scenarios, we decided it was best if we introduced boundaries in the average, transforming it into a moving or rolling average. That way, only the 'n' more recent statistics are used to compute the average unitary cost. To sum it up, **for every type of task** we save the 'n' latest timing measures, to compute the **average unitary cost**. As a note, obtaining the average unitary cost has little to no overhead, as we use accumulators from the **boost** [42] library, which is optimized for the purposes we need. Also as a side note, this 'n' parameter can be tunned through an environment variable called `NANOS6_ROLLING_WINDOW_SIZE`.

Secondly, we use the aforementioned unitary costs to make predictions, which are aggregated into what we call the `runtime load` metrics. These metrics aggregate an approximation of the workload in each status in real time. The approximation is due to aggregating task predictions, and not the actual elapsed time of tasks. The **ready, blocked, paused, instantiated, and finished task statuses** then, have their own predictions of aggregated cost as a runtime metric, which are combinations of predictions.

These runtime workload predictions per task status are further optimized by being updated once tasks complete their execution. When a task reaches completion, its elapsed time is subtracted from the prediction of the parent if it had one. This is limited, however, by updating these metrics once again when the parent finishes its execution, to avoid side effects due to the imperfections of predictions (i.e., the last prediction of a child task could leave the parent task with a negative predicted elapsed execution time if the children tasks' timing were overpredicted).

Lastly, runtime loads are combined predictions that take advantage of task timing predictions, and with these combined predictions we create **forecasts of CPU utilization**. These are fired periodically, as a method of producing an approximated value for the CPU utilization that is needed for the next period. To put it in an exemplified scenario, at timestep '`i`', we poll runtime workload metrics to know the predicted amount of work left in queues. Then, with several heuristics, we compute our prediction of CPU usage for timestep '`i+1`'. At timestep '`i+2`', apart from doing everything previously mentioned, we also compute the error in the prediction of timestep '`i+1`', and we use it to tune our predictions further.

8.2.2 Predictions Related to PQoS

Since we monitor not only runtime related metrics but also hardware events through Intel's® PQoS library, we provide predictions for these as well. Similarly to the computing of other predictions, once tasks reach completion, their events are normalized by the cost specified. Then, these unitary event values are averaged in the same way unitary cost is for timing predictions, through a moving average.

The normalized values for each event are then used to predict events for future tasks. These predictions can be used to apply different schedule techniques. As an example, if we anticipate that a set of tasks is going to use a huge amount of shared cache, maybe scheduling policies that intertwine these with tasks that use almost no shared cache would be a great idea.

8.3 Preliminary Results

Before we could use any of the predictions mentioned in previous sections, first we had to assess if the predictions were accurate. To do this, before doing any complete evaluation, we assessed the accuracy of timing predictions in tasks using two applications with different features on the same machine. Our initial experiments had the next features:

- They were all executed in the SSF machine. Details about this machine are given in section 10.
- The first application tested was **Mergesort**. The size of the array to sort was 100 million elements, and the final depth size was 1 million elements. This creates thousands of tasks, with mixed granularities. More about this application is discussed in section 10.
- Since we wanted to test different features, the second application tested is **Cholesky**. The size of the factorized matrixes was 32768 columns by 32768 rows, and the block size was 2048 by 2048. This creates a small number of tasks but with coarse granularity. More about this application is discussed in section 10.

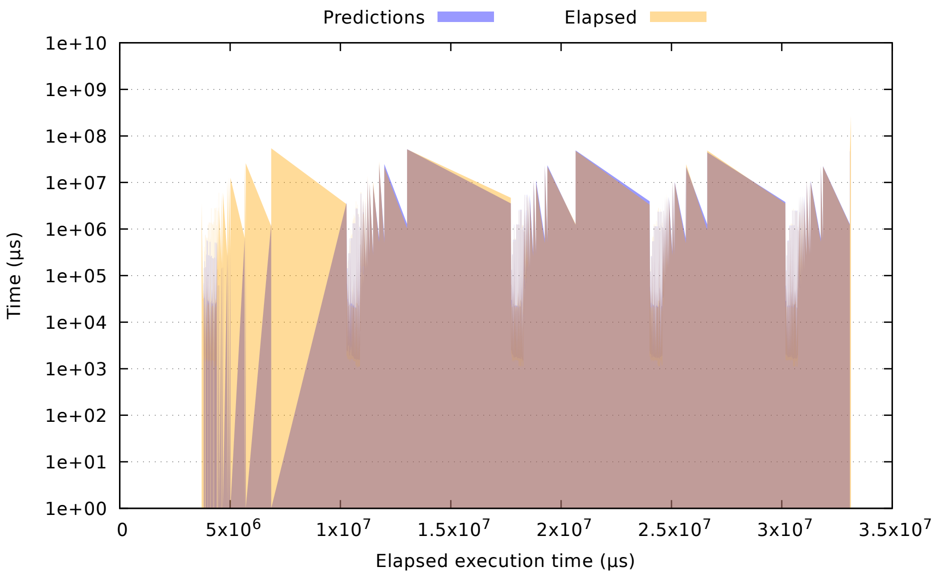


Fig. 8.1: Predictions compared to real timing values of tasks in a Mergesort execution

Our first test was conducted with the Mergesort application. Shown above, in figure 8.1, pairs of values are plotted throughout the execution (elapsed execution time – x-axis). These pairs are the predicted task time and the actual task time and they are plotted at the exact moment the task completes its execution in the application. To better comprehend the accuracy of predictions, we have filled the area beneath these two series. Clear orange areas mean there were either no predictions or underpredictions (since the blue of predictions does not clash in these areas). Clear blue areas mean there were overpredictions. The slightly purple areas are where the two pairs meet, thus accurate predictions.

Shown above, we see that at the start of the application, predictions are inaccurate. As soon as the monitoring has obtained more metrics, these predictions start being accurate (around the second large spike in the plot). However, since thousands of tasks are shown, it is hard to grasp any clear ideas, which is why we include the two plots shown below, which are zoomed areas of the previous plot.

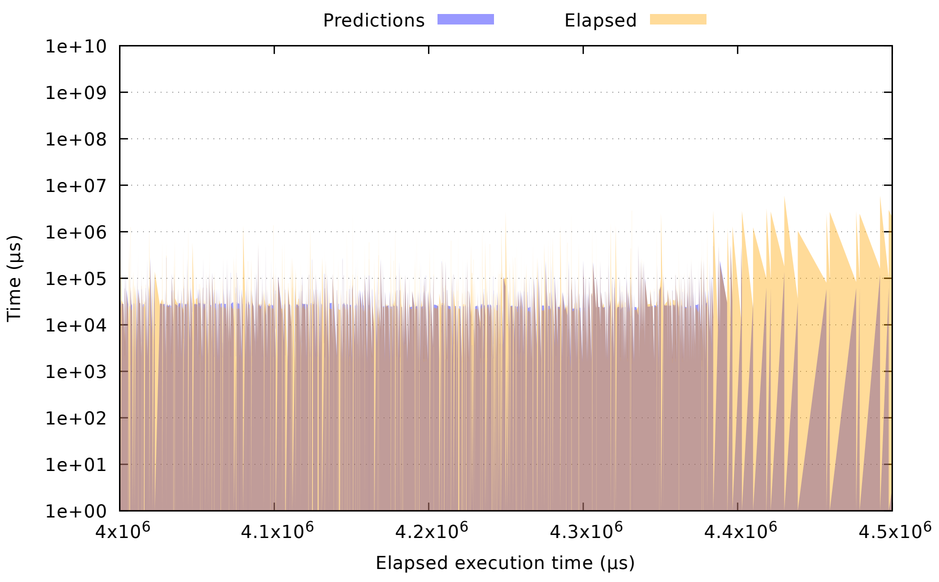


Fig. 8.2: Zoomed in view at a start zone of plot 8.1

As shown in figure 8.2, at the start, there are lots of missing predictions. In fact, there should only be missing predictions, since, at the beginning of the application, no task has completed yet; thus unitary costs cannot be computed. This means that all the missing predictions from both plots should be clustered at the start. However, as mentioned above, pairs of predicted versus real time are plotted once tasks complete, and it is quite common that the tasks that are created first (the ones that have a nonexistent prediction) do not complete until later in the execution.

However, as we can see in figure 8.3, once a few tasks have completed their execution, predictions get more and more accurate as time goes, and after a few iterations, we can grasp how predictions become extremely accurate. The elapsed execution time of the application is a little over 32 seconds, and the average prediction accuracy for timing was **78.50%** for `merge` tasks, and **89.84%** for the `merge_sort` tasks.

For completeness, we decided to execute Cholesky to evaluate if the inaccurate predictions were also found at the start of this application. To our surprise, in executions of Cholesky with a small number of tasks ($\tilde{5}00$ tasks), no predictions were found, even though the tasks had coarse granularities. We studied this case and found out that it was not until well into the application when tasks completed, and by the time that

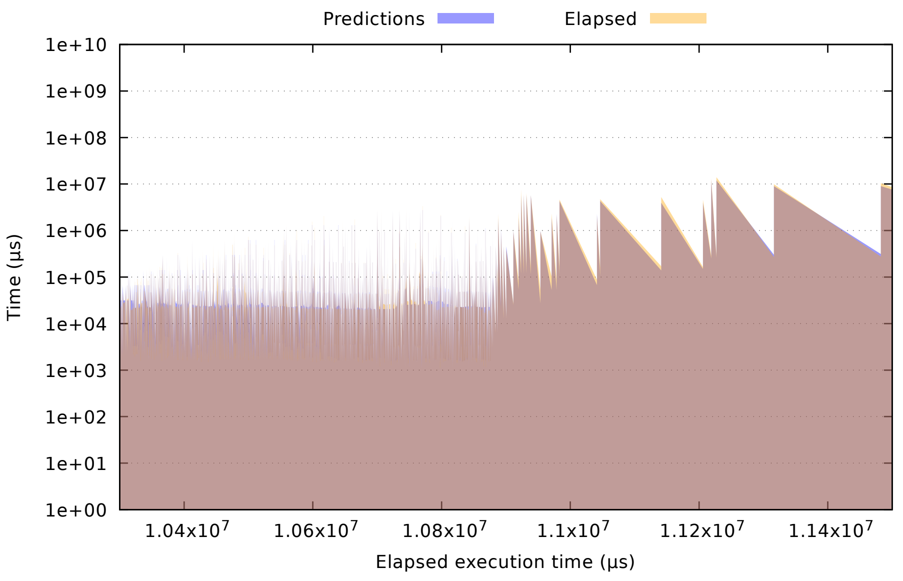


Fig. 8.3: Zoomed in view at a mid-zone of plot 8.1

enough tasks completed, all tasks had been spawned and thus no predictions could be made for future tasks.

Due to the initial inaccurate predictions and the case of Cholesky, where no predictions could be found at all, we decided to optimize our approach by adding the Wisdom mechanism, which we explain in the next section.

8.4 Wisdom Mechanism

Through the preliminary results we obtained, we decided it was best if we thought of a mechanism that allowed the runtime to save some general hints for future applications. These hints are in the form of unitary costs. When this mechanism is enabled, at the end of an application the average unitary costs of each type of task are saved in a file. Since these unitary costs are normalized, they are able to work in any scenario of input. At the start of an application, the previously mentioned file is checked. If such file exists and unitary costs are available, these are loaded into runtime statistics so that predictions can be done since the start.

To obtain a fast mechanism with files in a human-readable format, we save these statistics in a JSON file, with a tree-like structure. To further optimize the whole process, we also store information such as the average prediction accuracy. The file is modified always comparing the previous and current accuracies.

This mechanism can be controlled through the `NANOS6_MONITORING_WISDOM_ENABLE` and `NANOS6_MONITORING_WISDOM_PATH` environment variable. The first enables or disables the mechanism, and the second specifies the path where the file should be searched and saved.

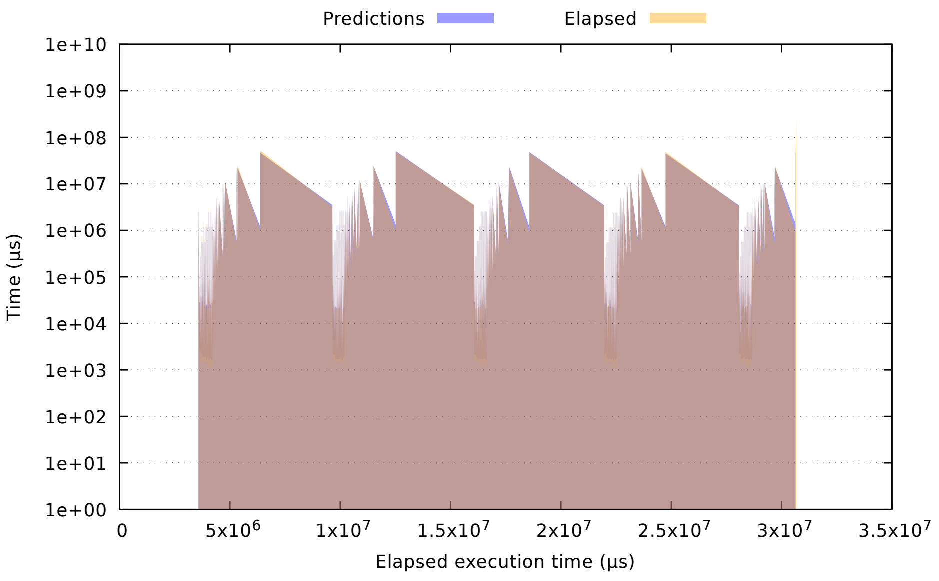


Fig. 8.4: Predictions compared to real timing values of tasks in a Mergesort execution with the Wisdom mechanism enabled

With this mechanism, we decided to rerun the preliminary results, this time with wisdom enabled. Figure 8.4 shows the previous experiment using Mergesort. This time, however, we can see that the predictions are far more accurate than before at the start. To easily grasp this, we zoomed in at the start zone of this plot as well. This is shown in figure 8.5.

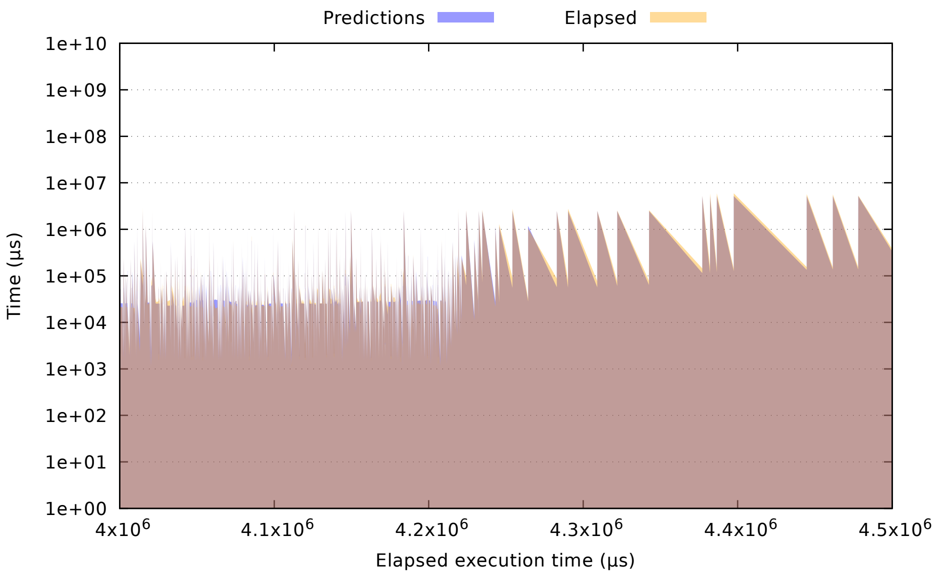


Fig. 8.5: Zoomed in view at a start zone of plot 8.4

The execution times were identical to the executions with Wisdom disabled, around 30 to 31 seconds, and the average prediction accuracy for timing was **81.23%** for `merge` tasks, and **91.78%** for the `merge_sort` tasks.

Granted that the mechanism works and it improves the accuracy of timing predictions, and for completeness, we also tested it in Cholesky to see if, this time, at least we obtained predictions. This is shown in figure 8.6.

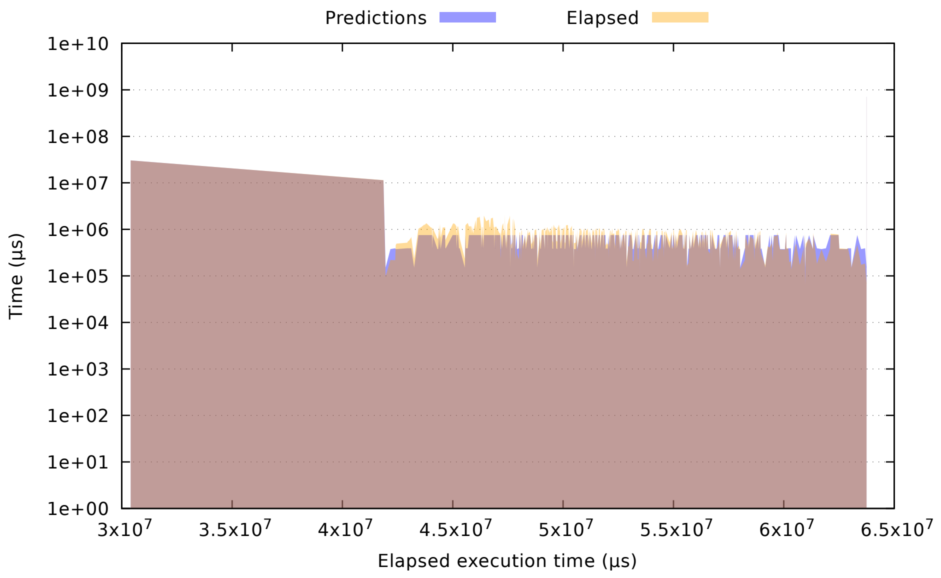


Fig. 8.6: Predictions compared to real timing values of tasks in a Cholesky execution with the Wisdom mechanism enabled

When it comes to statistics for the execution above, its elapsed time was 64 seconds and the average accuracy of timing predictions was **72.81%** for `gemm` tasks, **75.25%** for `potrf` tasks, **70.37%** for `syrk` tasks, and **76.98%** for `trsm` tasks.

8.5 Prediction Error Analysis

We have shown how predictions are made and preliminary results about their accuracy, but an equally important matter is to discuss how we compute the error of predictions, which inversely relates to the accuracy plots we will show in the next chapter.

To this date, many models tailor the error made in predictions or forecasts. A common approach is the mean percentage error, which, in a set of predictions, takes each real value, subtracts the prediction, and divides the difference by the real value. In other words, this computes an expression that relates the error of the prediction to the actual value. All these computations are then averaged and converted into a percentage to obtain the **M**ean **P**ercentage **E**rror [43]. The next expression shows its formula, where a_i stands for the actual value and f_i for the forecast or predicted value.

$$MPE = \frac{100\%}{n} \sum_{i=1}^{n} \frac{a_i - f_i}{a_i}$$

Although commonly used, this is not the appropriate model for our approach. This is mainly because, even though negative percentages can be avoided by getting the absolute value of the `MPE`, huge differences may show errors of over 100%.

Another approach suggests modelling prediction errors through the **S**ymmetric **M**ean **A**bsolute **P**ercentage **E**rror [44]. This measure relates the difference between the predicted and actual value to the average value between them. It is called symmetric due to using absolute values and relating them to the average between the used values. The expression to compute the **SMAPE** is shown next, where the aforementioned syntax still holds.

$$SMAPE = \frac{100\%}{n}\sum_{i=1}^{n}\frac{|a_i - f_i|}{(|a_i| + |f_i|)/2}$$

However, before deciding to go with **SMAPE** we found an error model that fit our needs. This is the **R**elative **P**ercentage **E**rror [45], which relates the error with a function tailored by the user, as shown in the next expression.

$$RPE = \frac{100\%}{n}\sum_{i=1}^{n}\frac{|a_i - f_i|}{|f(a_i, f_i)|}$$

This $|f(a_i, f_i)|$ function, as cited in [46], is not easy to define:

Defining relative difference is not as easy as defining relative change since there is no "correct" value to scale the absolute difference with. As a result, there are many options for how to define relative difference and which one is used depends on what the comparison is being used for. In general, we can say that the absolute difference $|\triangle|$ is being scaled by some function of the values x and y, say $f(x,y)$.

Common choices for the function include:

- Well-known ranges
- $max(|a_i|, |f_i|)$
- $min(|a_i|, |f_i|)$
- $max(a_i, f_i)$
- $min(a_i, f_i)$
- $(a_i + f_i)/2$
- $(|a_i| + |f_i|)/2$ (just like in the SMAPE approach)

For our implementation, in predictions related to timing we use the absolute maximum value between both measures – $max(|a_i|,|f_i|)$. However, for CPU utilization predictions, since we know the range of the prediction must always be between zero and the maximum amount of available CPUs by the runtime, we use this range.

Finally, our expressions to measure the error in predictions of task timing and predictions of CPU utilization are, respectively, the next ones:

$$TASK-PREDICTION_{Error} = \frac{100\%}{n} \sum_{i=1}^{n} \frac{|a_i - f_i|}{max(|a_i|,|f_i|)}$$

$$CPU-PREDICTION_{Error} = \frac{100\%}{n} \sum_{i=1}^{n} \frac{|a_i - f_i|}{MAX\_CPUs}$$

And the accuracy we show in the plots from further sections is obtained by simply subtracting the relative percentage errors from 100%.

9 | Enhancing Scheduling in OmpSs-2

Having a large number of statistics and metrics through the monitoring module provides, not only information to the runtime, but also capabilities to enhance existing scheduling policies or even create new ones. In this chapter, we explain what all the aforementioned metrics and modules are currently used for. In chapter 12, we continue by proposing new ideas that are in the roadmap. First, we begin by briefly explaining a technique introduced in previous works. Afterward, we discuss the newly created techniques.

9.1 Autofinal

From the final clause, and through normalized cost, came the idea of creating a module called the **autofinal** module. This module can determine at task creation time if the task should be a `final` task. This is possible through monitoring information and predictions, and it is determined by several heuristics which contain various environment variables to tweak and control this module.

Although this module is present and uses monitoring, as it is not part of this thesis, we decided it was best not to explain it thoroughly. A detailed description and evaluation of this module, as well as the techniques it introduces, can be found in our previous work [22].

9.2 Time-To-Completion Infrastructure

One of the secondary objectives of this thesis was to offer users an API with which they could poll the most interesting metrics from monitoring. Combining some metrics, we created the Time-To-Completion infrastructure, which combines predictions and metrics to approximate the estimated time until an application completes.

This infrastructure optimizes how predictions are made and combined so that the estimation of time is as precise as possible. If predictions were accounted from the start until the end of tasks, this would mean that in the end, the prediction of a task would be subtracted from the estimation of overall time. If this were the case, instead of being an estimation that continually decreases at an expected smooth rate, it would remain immutable while tasks do not finish execution, and once they ended, it would suddenly drop quickly.

These are some of the most remarkable features of this infrastructure:

- For every task type, the accumulated cost is saved separately from the average unitary cost. This due to the variability of unitary costs. Once users want to poll the estimation, the accumulated cost is multiplied by the average unitary cost at that period.
- Users either poll the time at a certain moment or set an automatic output with a refresh rate. Users control the frequency of output.
- Once a task completes its execution, its elapsed execution time is subtracted from the parent task if existent. This is so that the overall estimation is as accurate as possible.

This estimation can be used by any external libraries, such as Slurm [47], to optimize scheduling of executions. Slurm is an open source cluster management and job scheduling system for Linux clusters. It allocates access to resources for users for a specific duration, so that their jobs can be executed. It also provides a mechanism to start, execute, and monitor parallel jobs. Finally, it schedules resource usage by managing queues of work or jobs.

Libraries such as Slurm could highly benefit from our infrastructure. Oftenly, users of these clusters will input a substantial amount of duration so their jobs are not killed. Two main scenarios might occur in which both users and resource utilization benefit from libraries having accurate estimations of the real duration of the jobs in their queues.

First, if the user fell short in the specified duration of his or her job, the estimation of time provided by the runtime allows Slurm to modify the job to add more duration if it made sense. This is to avoid wasting the time the user already spent, as the user would need to restart his/her execution from scratch. On the other hand, if the user specified more time than his or her job actually needed, Slurm can benefit from this by anticipating an earlier completion of the job.

9.3 Dynamic CPU Activation

As mentioned in section 8, timing predictions for tasks are used to predict runtime workloads. These are used to predict CPU utilization. With these, we can achieve one of the primary objectives of this thesis, which is to manage CPU usage automatically.

Our approach towards this objective is to extend the default scheduler of Nanos6 with the capabilities to do the aforementioned tasks. The focus of this approach is due to the scheduler being the element that controls CPUs through a CPU manager. The scheduler is in charge of providing idle CPUs to threads when work becomes available, and also to disable CPUs when there is a lack of tasks to suffice the currently available CPUs. Next, we show pseudocodes with every primary function in the scheduler interface that interacts with CPU managing.

Algorithm 9.1 Pseudocode of the `addTask` function of the scheduler interface

```
add_task(task, get_idle)
    spinlock ⇐ lock;

    queue.push(task);
    if get_idle then
        idle_cpu ⇐ get_idle_cpu();
    else
        idle_cpu ⇐ $\emptyset$;
    end if

    spinlock ⇐ unlock;
    return idle_cpu;
```

Pseudocode 9.1 shows the basic operations when adding a task. The task is inserted into the task queue and, if the thread had asked for it, an idle CPU is returned. This last action must be removed, so this function will not be able to return idle CPUs anymore. Pseudocode 9.2 shows the actions taken when a thread asks for a task. When this happens, if the queue is not empty, a task is returned. On the contrary, if there is permission to do it, we mark the CPU (`compute_place`) which the thread was executing on as idle. This action must be removed as well since schedulers no longer manage CPUs.

Algorithm 9.2 Pseudocode of the `getTask` function of the scheduler interface

```
get_task(compute_place, can_idle)
    spinlock ⇐ lock;
    task ⇐ queue.get_task();
    spinlock ⇐ unlock;

    if task $\neq \emptyset$ then
        return task;
    end if

    if can_idle then
        cpu_becomes_idle(compute_place);
    end if
```

Another action that must be withdrawn from the schedulers is the ability to return idle CPUs with the function shown in snippet 9.3. Worker threads called this function.

Algorithm 9.3 Pseudocode of the `getIdleCPU` function of the scheduler interface

```
get_idle_cpu(force)
  idle_cpu ⇐ ∅;

  spinlock ⇐ lock;
  if force || queue ≠ ∅ then
    idle_cpu ⇐ get_idle_cpu();
  end if
  spinlock ⇐ unlock;

  return idle_cpu;
```

All the aforementioned actions must be replaced with a call to a service. In other words, in every line of code where schedulers managed CPUs, instead, we activate a service controlled by CPU utilization predictors. This service takes into account the prediction for the next period of time and takes the actions it must to ensure that only the sufficient CPUs are utilized, in order not to waste resources. Pseudocode of the service is shown in snippet 9.4.

Algorithm 9.4 The Dynamic CPU Activation service

```
dynamic_cpu_activation_service(prediction)
  if predict_next_period then
    workload ⇐ get_runtime_loads();
    instances ⇐ get_num_instances();

    utilization ⇐ get_current_cpu_utilization();
    accuracy ⇐ RPE(utilization, prediction);

    prediction ⇐ min(instances, workload / PREDICT_RATE);
    prediction ⇐ min(prediction, MAX_NUM_CPUS);
    prediction ⇐ max(prediction, 1.0);            // Minimum 1 CPU
    return prediction;
  end if
```

With the predictions from this service, we call the now extended CPU manager, telling it the operations that must be done. That is, idling CPUs that will be unused for the next period, or waking up CPUs that will poll tasks from the predicted workload. A detailed and extensive evaluation of predictions and these enhanced scheduling techniques is given in the next chapter. Timing predictions for tasks and the assessment of the wisdom mechanism, however, is already discussed in chapter 8.

10 | Evaluation

In this chapter we finalize by showcasing the extensive evaluation of our proposals. First, we describe the applications used in our experiments. Then, we discuss the architectures used. Once that is out of the way, we start the experiments by analyzing the overhead of the introduced modules and infrastructures. Afterward, we assess the accuracy of CPU utilization predictions. Then, we show the enhanced usage of resources provided by the dynamic CPU activation infrastructure. Finally, we extend the evaluation done in chapter 8 for the wisdom mechanism.

10.1 Applications

To assess the accuracy of predictions, we prepared a set of applications with noticeable differences. Next we list each of the applications in the set, briefly commenting on all of their features.

- **Cholesky**: The Cholesky factorization is a decomposition of a positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, which is useful e.g. for efficient numerical solutions. The benchmark was executed with an **input size** (matrix size) of **32768 by 32768** elements. It is an **iterative** implementation, and the **block size** chosen for the executions was **2048 by 2048** elements. These parameters make this application have **coarse grained tasks**. The **parallelism** available is **scarce**, as not many tasks are created due to the relatively big block size, compared to the input size.

- **Mergesort**: Mergesort is a sorting algorithm in which tasks are subdivided until a certain point. The implementation of this application was **recursive**. We executed this application choosing an **input size** of **100 million** elements to be sorted. The **depth of recursion** was controlled through the final clause with a value of **1 million**. This means that the application subdivided tasks until each chunk of the vector to sort had 1 million elements. These parameters create a reasonable amount of tasks compared to Cholesky. The tasks created have **medium to coarse granularities**. To test the accuracy of our predictions, each execution contained **5 iterations**, thus sorting and randomizing the array five times in every execution.

- **NQueens**: NQueens is a problem that finds solutions to place N non-attacking queens on an 'N' by 'N' chessboard. Solutions exist for all natural numbers 'N' with the exception of N=2 and N=3. The **size** of the chess board chosen was **17 by 17**. It is a **recursive** application. **Recursion depth** is controlled through the final clause, with a value of **4**. Thus, when the column index reaches 4, tasks become final. Since it has exponential growth in tasks, this application creates a **lot of tasks** ranging from **low to medium granularities**.

- **Strassen**: Strassen is an algorithm for matrix multiplication. It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrixes. Our implementation is **recursive**. The **size of the matrixes** was **8192 by 8192**, as it consumes more memory than Cholesky. Recursion was controlled through the final clause as well. The maximum depth of recursion was when the **block size** reached **256 by 256** elements, thus creating a large number of tasks, approximately as many as in NQueens. The **small granularities** in this benchmark created a **lot of available parallelism**.

- **Multisaxpy**: Multisaxpy is an **iterative mini-app** that combines scalar multiplication and vector addition. SAXPY stands for "Single-Precision A*X Plus Y". It is a function in the standard Basic Linear Algebra Subroutines (BLAS) library. To get representative executions, we had to execute this application with arrays of **100 million** elements in **size**. The **chunk size** we chose was **1 million** elements, similarly to Mergesort. However, we had to execute **a thousand iterations** per execution to get meaningful executions. This application also created a **lot of parallelism** through tasks ranging from **low to medium granularities**.

- **Heat**: Our last application in the set was the Heat simulation, which uses an iterative Gauss-Seidel method to solve the Heat equation. The Heat equation is a parabolic partial differential equation that describes the distribution of heat (or variation in temperature) in a given region over time. We executed this application with a **size of 32768 by 32768** and a block size of **1024 by 1024**. Even though this application creates much parallelism as it creates around half a million tasks, these range from **low to coarse granularities**. In order to get long executions, each of them executed **500** timesteps of the simulation. The simulation had **2 heat sources**.

As shown above, with our set we can test applications that range from low parallelism to high parallelism. The full spectrum of granularities is also widely tested, as these vary between applications. When it comes to execution time, as shown in table 10.1, the average elapsed execution time ranges from as little as 12 seconds to as much as 500 seconds for some applications. We believe that getting high accuracies in short-lasting applications proves the correctness of our approach, as these executions are the hardest to make predictions for. However, we complement our evaluations with long-lasting executions as well.

Application	# of Tasks	Average Elapsed (SSF)	Average Elapsed (CTE-Power)	Average Elapsed (CTE-KNL)	Average Elapsed (Marenostrum4)
Cholesky	~500	22 seconds	497 seconds	29 seconds	10 seconds
Mergesort	~15k	13 seconds	12 seconds	27 seconds	14 seconds
NQueens	~30k	55 seconds	214 seconds	123 seconds	101 seconds
Strassen	~40k	20 seconds	16 seconds	26 seconds	18 seconds
Multisaxpy	~100k	22 seconds	12 seconds	29 seconds	13 seconds
Heat	~500k	129 seconds	71 seconds	163 seconds	103 seconds

Table 10.1: Features of the application set used to evaluate our proposals

In the mentioned table, we see the average elapsed execution times for each application and architecture tested. These, as well as every result in this thesis, were obtained out of averages of 5 executions per data point. When it comes to the accuracy shown in plots, each data point shows the accuracy at a specific timestep. Accuracy is reported at a frequency of 100 microseconds. Thus, every 100 microseconds, the accuracy for the previous period is saved in a buffer so it can be displayed later on.

10.2 Architectures

We claim that our proposals are independent of applications, input parameters for these and architectures or number of CPUs. Due to this, we tested our contributions in several architectures. We believe that the MxN mesh resulting from testing the aforementioned M applications in these N architectures gives our contributions a fair and complete evaluation. The most remarkable and basic features of each architecture are listed next.

- **SSF**: The SSF cluster contains 8 x Intel® Xeon® Phi E5-2690v4 nodes. Each of these is equipped with 16 GB of memory, 56 cores divided into 2 sockets, and 240 GB of Intel® SSD DC S3520 Series. The cores in this machine are equipped with Intel® RDT technologies, allowing us to use PQoS monitoring. For the evaluation of this thesis, a node from this cluster was exclusively dedicated to all our purposes, so we had access to install any tool or kernel we needed. The operating system is SUSE Linux Enterprise Server.
- **CTE-Power9**: CTE-POWER is a cluster based on IBM Power9 processors. Its operating system is Red Hat Enterprise Linux Server 7.4, with an Infiniband interconnection network. Its compute nodes are composed by 52 nodes each of them with 2 x IBM Power9 8335-GTG @ 3.00GHz, with 20 cores per socket and 4 threads per core, totaling 160 threads per node. For this thesis, we used a whole node with exclusive access through sending jobs to the job queue system. Each of the nodes has 512GB of main memory distributed in 16 dimms of 32GB each @ 2666MHz. The storage was composed by 2 x SSDs of 1.9TB as local storage and 2 x 3.2 TB of NVME. Using this machine allowed us to test a different architecture with far more cores than any other machine in this list.

- **CTE-KNL**: CTE-KNL is a cluster based on Intel® Xeon® Phi Knights Landing (KNL) processors, a SUSE Linux Enterprise Server 12 SP2 Operating System and an Intel® OPA interconnection. It has 16 compute nodes, each with 1 x Intel® Xeon® Phi CPU 7230 @ 1.30GHz 64-core processor. Even though hyperthreading exists in these processors, it was disabled and thus unusable, so 64 cores were used. Each node contains 96GB of memory distributed in 8 x 2GB MCDRAM @ 7200 MHz dimms. The storage was composed by 120GB of SSD. A node with exclusive access was used in this cluster, also through the internal job queue system. Several optimized compilation options were enabled for every application compiled, including AVX–512 instructions and other memory-alignment flags.

- **Marenostrum4**: MareNostrum4 is a supercomputer based on Intel® Xeon® Platinum processors from the Skylake generation. It is a Lenovo system composed of SD530 Compute Racks, an Intel® Omni-Path high performance network interconnect and running SuSE Linux Enterprise Server as operating system. It consists of 48 racks which house 3456 nodes, totaling 165888 processor cores and 390TB of memory. Each compute node contains 2 x sockets of Intel® Xeon® Platinum 8160 CPU with 24 cores @ 2.10GHz each, totaling 48 cores per node. Each node also contains 200GB of local SSD and 96 GB of main memory.

10.3 Overhead

A previous evaluation of the overhead of each module was conducted. Shown below in figure 10.1 we can see four series for four applications. Above each of these four series is the slowdown introduced by activating the monitoring modules. We decided to do an extensive evaluation using these four applications as it completed the spectrum of features. Each of the series is computed out of an average of five executions in each of the four machines. We averaged different architectures since the difference in overhead between these was hardly noticeable.

As shown, measuring PQoS events seems to be the lightest between the PQoS module and the monitoring one. In some scenarios, both modules present a slowdown of 5%. However, when combining the usage of both, since the atomic structures are combined as well, the slowdown is not doubled (10%) but rather slightly worse, hardly ever reaching more than 5% of slowdown.

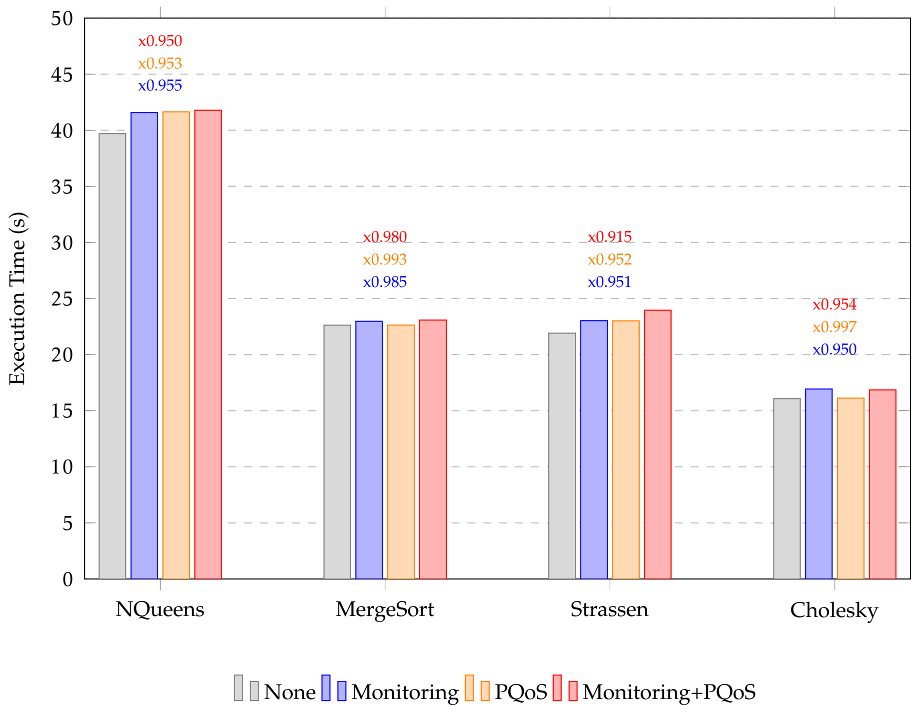


Fig. 10.1: Different series show the overhead introduced by monitoring, PQoS, and both modules

One scenario that stands out is the usage of both modules for Strassen, which presents a slowdown of 8.5%. We believe this is due to the fine granularity of tasks. We also believe, however, that the slowdown shown in these series is worsened by having to track down the overhead itself – that is, having to print out these statistics. If we take all of this into account, and we consider all the information gathered and the potential provided to the runtime, the usage of both these modules presents a negligible overhead.

10.4 CPU Usage – Prediction Accuracy

In this section we present different figures each with the accuracy of predictions for CPU utilization. These are important since we base the dynamic CPU activation mechanism on the accuracy of them. We present six figures, each with one of the benchmarks listed above. In these six figures are four series in each, each representing the execution of the application with the parameters mentioned above in a certain architecture. With these plots, we test the whole spectrum of applications and architectures. To better comprehend these figures, we specify in tables the average accuracy of CPU utilization predictions for all applications and benchmarks.

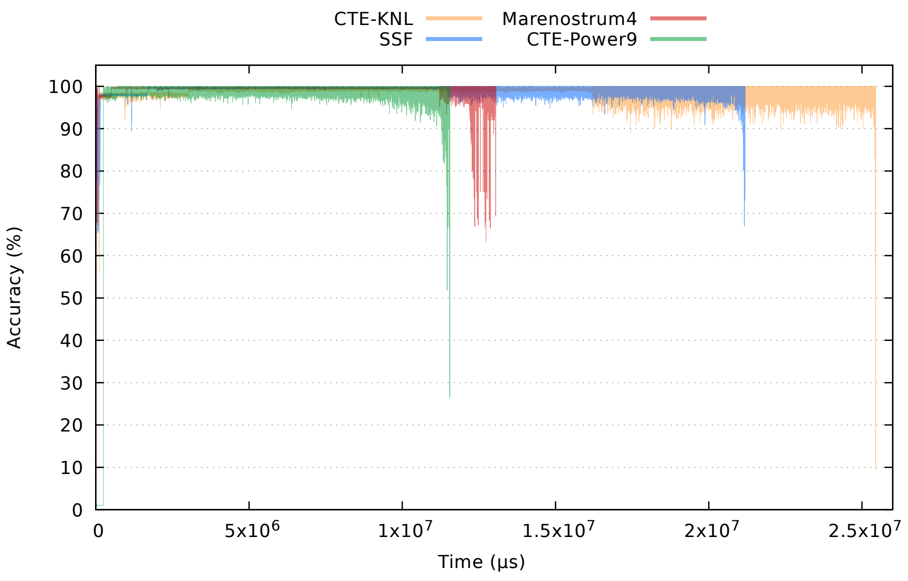


Fig. 10.2: Accuracy of CPU usage predictions for Multisaxpy

	SSF	**Marenostrum4**	**CTE-Power**	**CTE-KNL**
Average Accuracy	99.52%	99.18%	97.73%	99.08%

Table 10.2: Average accuracy of CPU usage predictions for figure 10.2

Figure 10.2 shows the accuracy of CPU predictions for the **Multisaxpy** application over time. Each series plots the accuracy of each prediction made for every period. As shown, there is a drop of accuracy right before the end of the application in each of the series. This will be a recurrent scenario in most experiments, since, even though our predictions are immediate, the runtime might take some time to accommodate to the current workload. In other words, once no more tasks are available, the mechanism forecasts that the utilization will instantly drop. However, what will happen is that, over a certain period, CPUs will become idle. This will happen when threads running on active CPUs try to poll tasks and no more are available. The average accuracy for predictions for this application can be seen in table 10.2.

A similar thing happens at the start of the application. Even though tasks might be ready to be executed, CPUs will not become active until threads are woken up on them and poll tasks. Both these scenarios can also be seen in the **Heat** application, shown in figure 10.3. However, the accuracy this time is higher as the runtime is quicker to react to changes in the available workload. The average accuracy is shown in table 10.3, below the mentioned figure.

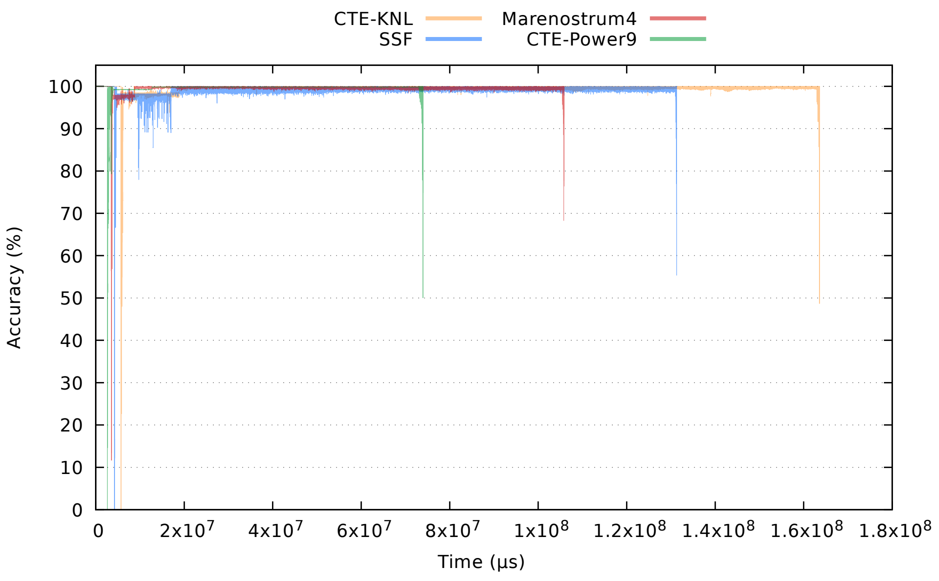


Fig. 10.3: Accuracy of CPU usage predictions for Heat

	SSF	**Marenostrum4**	**CTE-Power**	**CTE-KNL**
Average Accuracy	99.39%	99.60%	99.15%	99.65%

Table 10.3: Average accuracy of CPU usage predictions for figure 10.3

The results of our next application, **Cholesky**, are shown in figure 10.4. For this application, as mentioned in previous sections, we had to use the wisdom mechanism to ensure that predictions are made. This is because the input we chose for this application creates a small number of tasks. These tasks, due to the application's behavior, do not complete until deep into the execution. These issues cause for monitoring not to normalize costs, which means predictions cannot be done for task timing. This, at the same time, translates into being unable to make runtime workload predictions. Finally, this means that CPU utilization predictions cannot be done.

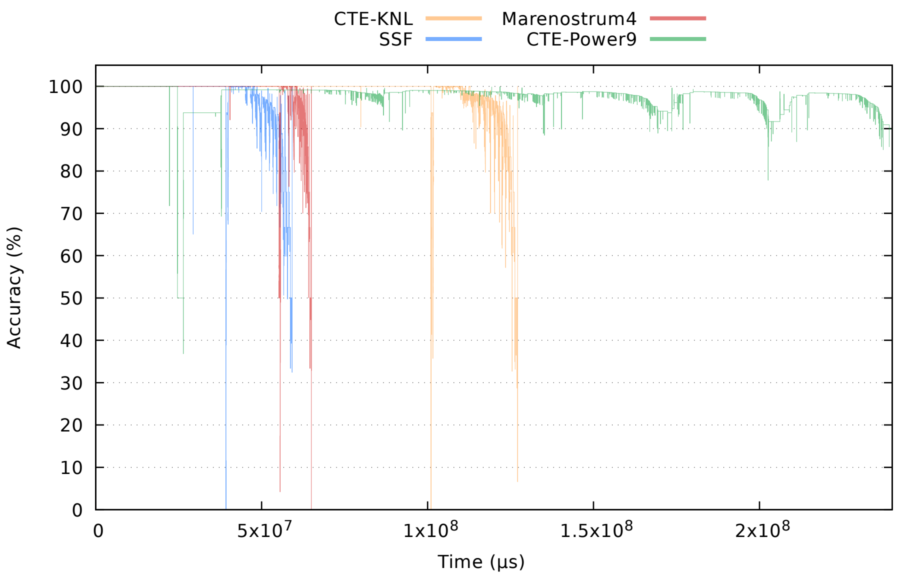


Fig. 10.4: Accuracy of CPU usage predictions for Cholesky

	SSF	Marenostrum4	CTE-Power	CTE-KNL
Average Accuracy	97.38%	94.24%	90.89%	94.74%

Table 10.4: Average accuracy of CPU usage predictions for figure 10.4

This application shows a relatively good average prediction. However, the scenario previously commented is worsened at the end of the application. Since there are almost no tasks and the ones left executing are huge, CPUs take a while to become idle. Due to this, even though the utilization should be lower, the runtime reports active CPUs when they should be in idle mode. This causes the error to be significant at the end. Nonetheless, the average accuracy of predictions shown in table 10.4 demonstrates that the accuracy, although lower than in other application, is still high. One of the series that stands out the most is CTE-Power. The accuracy is around 91%, which is worsened by the error being moderately bigger due to the amount of CPUs in that machine (160).

Figure 10.5 shows the evaluation for the **Mergesort** application. At first glance we see different behaviour than in other figures, however, this is due to having five iterations of Mergesort instead of one. This can be seen more clearly in figure 10.6, where we show the same plot with only the SSF series. Taking this into account, the series look much more similar to the ones displayed in other plots. As shown, at the end of each iteration in every series, there is a sudden drop in accuracy. This is caused, yet again, by runtime CPUs and threads not responding quick enough to the lack (or existence, at the beginning) of tasks. With all this, the plot is similar to the three previously discussed.

Table 10.5 shows that the accuracy is quite impressive for this application as well.

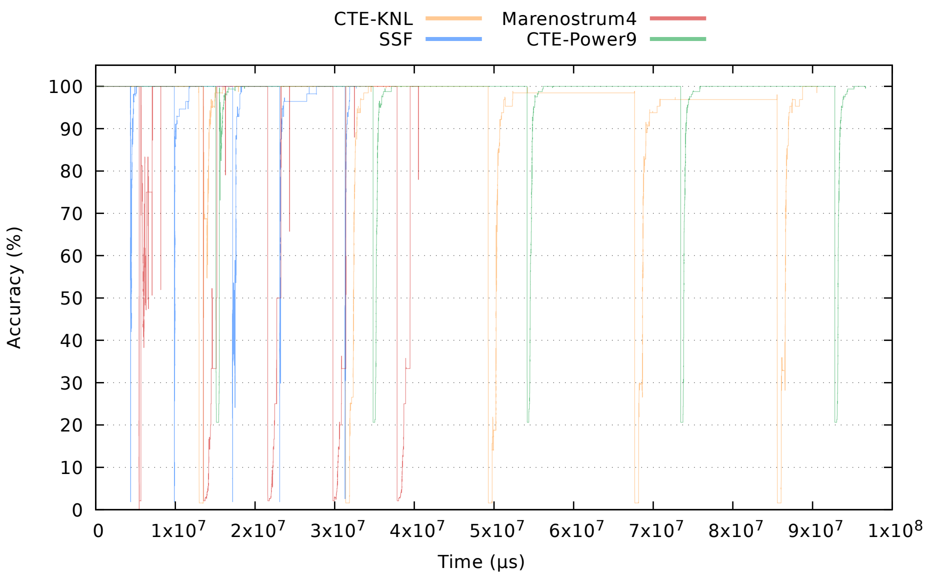


Fig. 10.5: Accuracy of CPU usage predictions for Mergesort

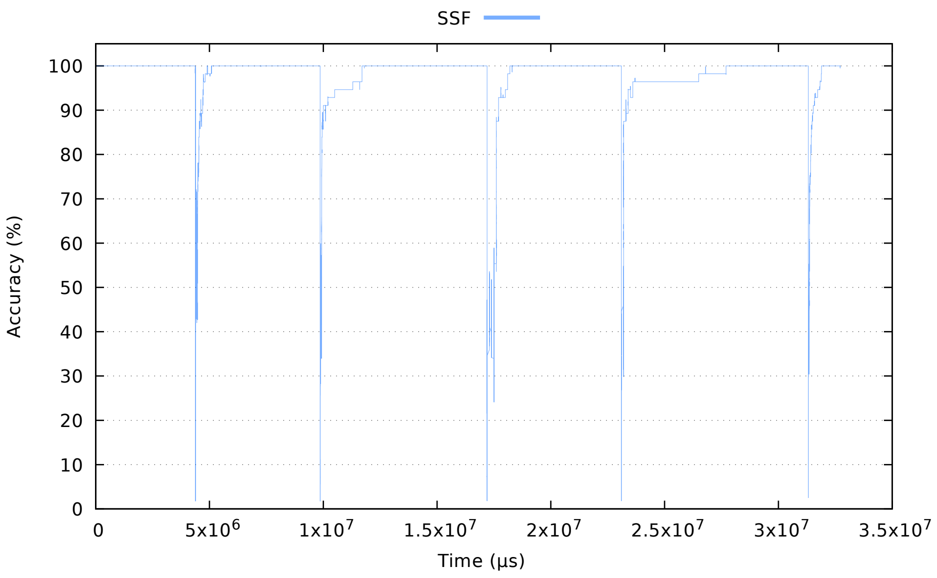


Fig. 10.6: Accuracy of CPU usage predictions for Mergesort for the SSF machine

	SSF	Marenostrum4	CTE-Power	CTE-KNL
Average Accuracy	96.77%	88.86%	97.16%	96.95%

Table 10.5: Average accuracy of CPU usage predictions for figure 10.5

After Mergesort, we tested the accuracy of our predictions with the **NQueens** application. Upon inspecting the results, we saw clear differences compared to the previous four applications. A similar plot would have resulted messy to inspect. To easily visualize the accuracy, we decided to split the plot into four subfigures, separating each architecture into one. These are shown in figure 10.7. As shown, the two upper architectures, SSF and CTE-Power9, present results similar to previous applications. This includes the characteristic downwards spike of accuracy right at the end of executions. However, in the run performed in Marenostrum4, we can grasp a few spikes of mispredictions during the execution. In CTE-KNL, these are present more often.

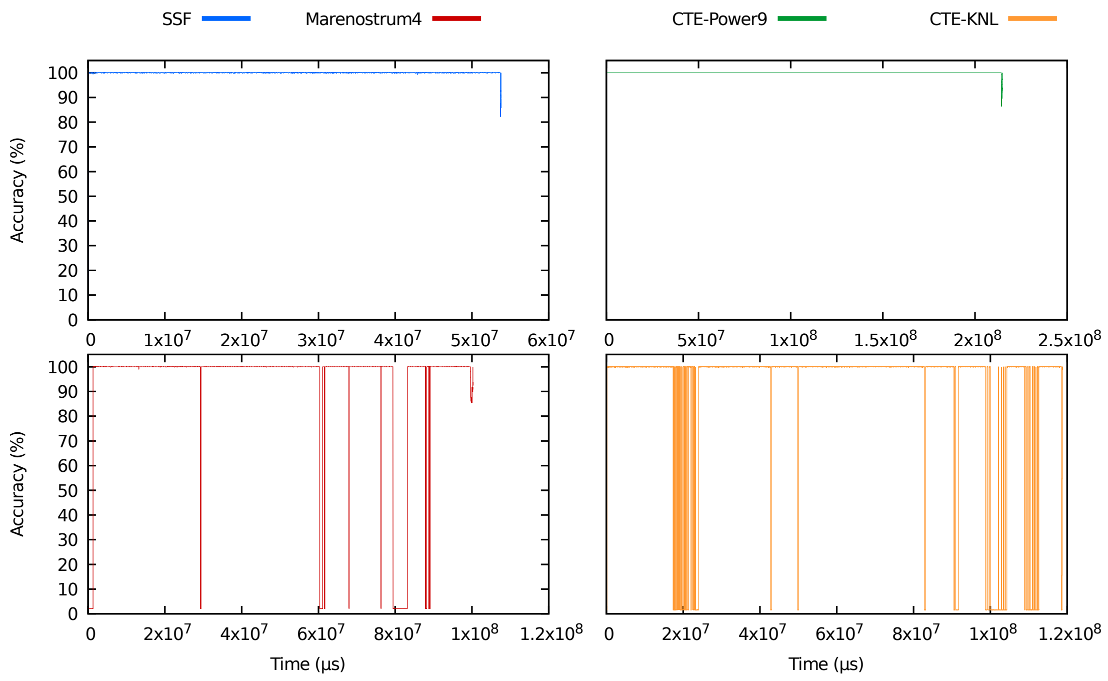


Fig. 10.7: Accuracy of CPU usage predictions for NQueens

Upon inspecting the collected data, we saw two interesting features. First, the overall average accuracy seemed to be reasonably similar to executions that did not present spikes. Second, even though tasks were being executed at various points throughout the execution, these had no predictions. Because of this, the predicted CPU utilization for subsequent periods was very low, however since tasks were being executed, the real CPU utilization was high. We concluded that this was due to NQueens being a recursive application. Initial tasks have no predictions, however these are not completed until well into the execution.

In other words, these spikes correspond to runtime workload metrics where the information is composed **only** by tasks with no predictions. Upon running these executions more times in all architectures, we saw this was a recurrent scenario for all architectures. However, some machines were more prone to present these spikes than others.

Figure 10.8 shows similar subfigures. These, however, represent the average accuracy over time instead of the current accuracy. To put it differently, figure 10.7 shows the immediate accuracy of every timestep in which CPU utilization predictions are computed. Figure 10.8 on the other hand shows the accumulated average accuracy over the whole execution. Thus, in this last figure, the accuracy presented at timestep i is the averaged accuracy using the immediate accuracies from timestep 0 to timestep i. These still let us know where spikes are present, but showcase that the average accuracy does not variate much as they represent a small portion of all the timesteps. The average accuracies for all architectures are shown in table 10.6.

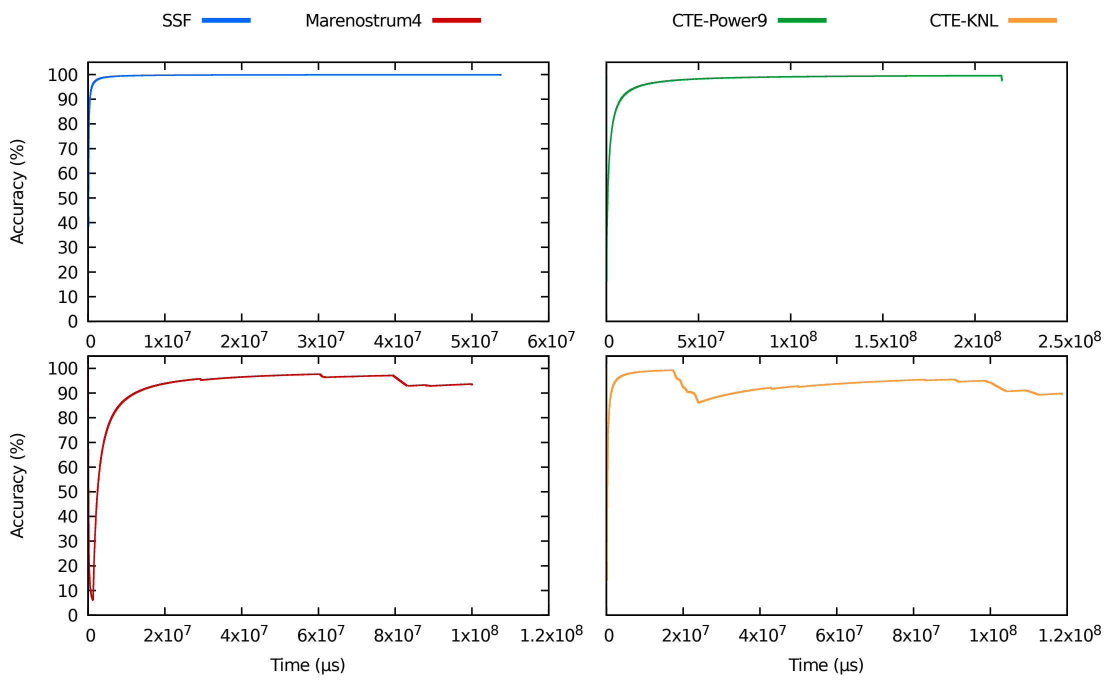


Fig. 10.8: Accumulated average accuracy of CPU usage predictions for NQueens

	SSF	Marenostrum4	CTE-Power	CTE-KNL
Average Accuracy	99.72%	93.46%	95.50%	89.41%

Table 10.6: Average accuracy of CPU usage predictions for figure 10.7

Our last experiment was conducted with the **Strassen** algorithm for matrix multiplication. The accuracy results are shown in figure 10.9. The initialization of matrixes is a large portion of the execution of this benchmark. During that phase, the accuracy was constantly at 100% for all machines, since only one CPU was used and predicted. Due to this, and to not undermine the error in the computation phase, we filtered the initialization from our data, as it does not provide a fair comparison with other applications.

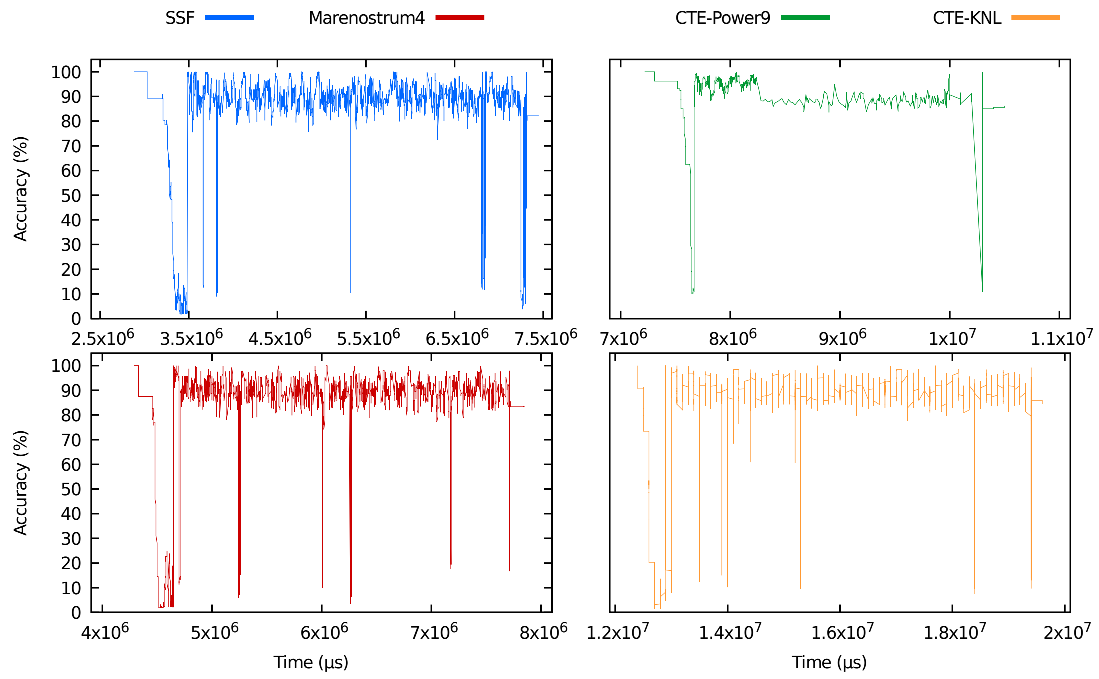


Fig. 10.9: Accuracy of CPU usage predictions for Strassen

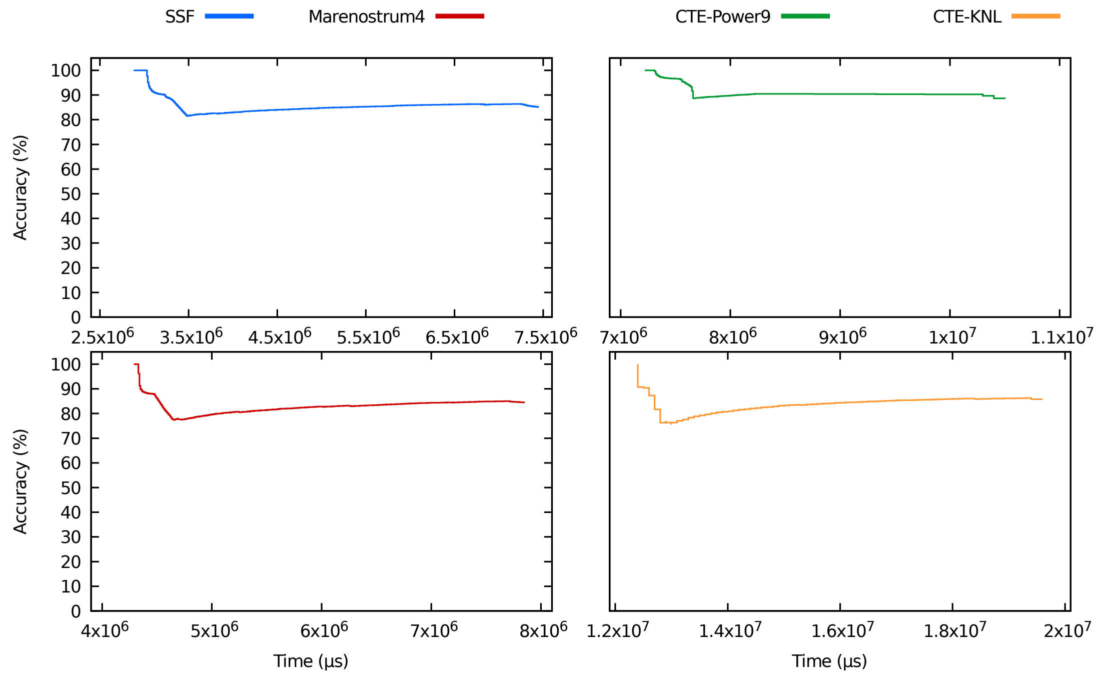


Fig. 10.10: Accumulated average accuracy of CPU usage predictions for Strassen

To better comprehend the results of Strassen, we also divided the series between four different plots. Filtering the initialization phase causes great differences in time-range across different architectures. As we can observe, these figures do not show significant similarities compared to other applications. The prediction accuracy seems to fluctuate between 80% to 100% for the entire duration. We believe this is due to the excessively fine granularity of tasks and the low execution time of this application. The smaller tasks are, the harder it is to predict their behavior through normalization. This application presents granularities as small as processing blocks of 256 elements.

The four series present similarities between themselves, however. Just like in previous executions, the starting and ending spikes are present. Similarly to the NQueens executions, a small number of spikes are found spread between the executions in Marenostrum4 and CTE-Power9. Even though the accuracy of predictions seems to fluctuate, the average seems to be around 90%, which is similar to the average accuracy for other applications. This is shown in table 10.7. To visualize the average accuracy over time, we also include figure 10.10, which, like for NQueens, shows the accumulated average accuracy of predictions at every timestep.

We show table 10.7 for completeness. In this table we combine all the average accuracies shown above, for a faster inspection. At the end of the table we compute the average of all predictions, using all the results gathered in this section.

Average Accuracies	**SSF**	**Marenostrum4**	**CTE-Power**	**CTE-KNL**
Multisaxpy	99.52%	99.18%	97.73%	99.08%
Heat	99.39%	99.60%	99.15%	99.65%
Cholesky	97.38%	94.24%	90.89%	94.74%
Mergesort	96.77%	88.86%	97.16%	96.95%
NQueens	99.72%	93.46%	95.50%	89.41%
Strassen	85.14%	84.52%	88.40%	85.81%
Total Average	**96.32%**	**93.31%**	**94.81%**	**94.27%**

Table 10.7: Average accuracies of CPU usage predictions for all machines & applications

One of the recurrent scenarios in all previous figures is the spikes that signal accuracy drops. These, as shown through the average accuracy, are a minor part of the overall execution. Oftenly, they show up either at the beginning or the end of executions. If needed, these can be solved like they were for Cholesky: by activating the wisdom mechanism. The potential of this mechanism is further discussed in section 10.6 of this chapter.

Finding these spikes spread throughout executions is another scenario we found particularly interesting. However, it does not present a big risk. To solve these temporary spikes, instead of taking actions at every timestep, scheduling policies could be built. These would gather information about the `N` latest timesteps, combine these predictions and then decide the overall utilization of CPU for the next `N` timesteps through heuristics. This can be further optimized by playing around with the frequency of predictions. For the shown results, we used a frequency of 100 microseconds, as reported above. Next we show our CPU utilization predictions in action, comparing them against the already existing default scheduler.

10.5 Dynamic CPU Activation

To test our predictions, we created the Dynamic CPU Activation mechanism, introduced in section 9.3. Then, we compared two schedulers. The first is the current default scheduler in Nanos6, the PriorityScheduler, which wakes up CPUs and they remain woken up until the end of the execution. This is a no-idle policy. The scheduler uses producer-consumer-based spinlocks that minimize thread contention, hence why CPUs can remain non-idle and constantly poll for tasks. Although very efficient while executing, this policy could utilize resources in a better way with our mechanism.

The second scheduler we tested is an extension of the PriorityScheduler with the Dynamic CPU Activation mechanism. When CPU usage is predicted to drop for future timesteps, CPUs are disabled instead of stuck in an endless polling loop. This is so that these resources can be better utilized for other purposes or by other libraries.

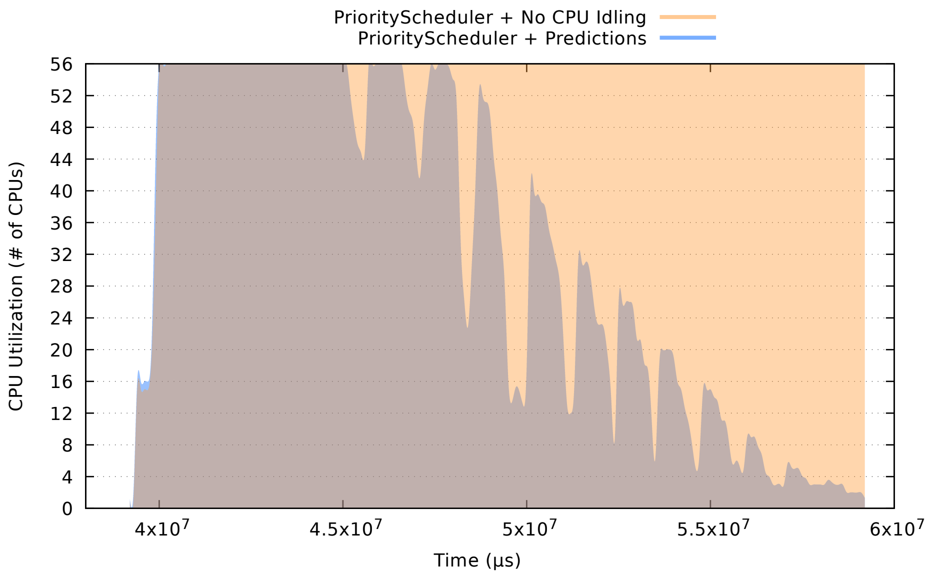


Fig. 10.11: Comparison of different scheduling techniques through resource usage

Figure 10.11 shows two executions of Cholesky in the SSF architecture, with the initialization of matrixes being filtered. Both had the same parameters and lasted similar execution times. We attribute this similarity to both the high accuracy of our predictions and also not underestimating workloads, since our approach predicts a similar higher utilization than the real one almost always. As we can observe, once the factorization starts in Cholesky, all the CPUs are working in the default scheduler. After a few moments, the utilization starts to drop and there are peaks of utilization.

As we are in a discrete domain – every prediction is made at a specific timestep – we can compute the whole area of the plot between the first and last timestep, and the total area between the two series. This gives us a number which, once converted to a percentage, allows us to visualize the amount of resources that could be reutilized by other libraries. Thus, computing the area between the blue and orange series, we obtain an area of about **14.12%** of the total area, which tells us that our approach utilizes resources 14% better than default policies for our experiment.

10.6 Normalized Costs

This section aims to demonstrate the effectiveness of normalized costs fully. To do this, we exemplify our approach using the wisdom mechanism. The best case scenario to test this is the Cholesky application. This is because, as shown before, with the parameters chosen, it needs the wisdom mechanism to have predictions.

To test this, we used the Cholesky application in the SSF machine. We first executed Cholesky with the size of the matrixes being **32768 by 32768** elements as usual, and the block size **1024 by 1024**. We used this block size instead of the usual one to be able to have some predictions.

Once the first execution is completed, a file is created with the normalized costs of every type of task. With this file, our second test is able to provide predictions since the beginning of the execution. This second execution had the same parameters as the first one, just for completeness.

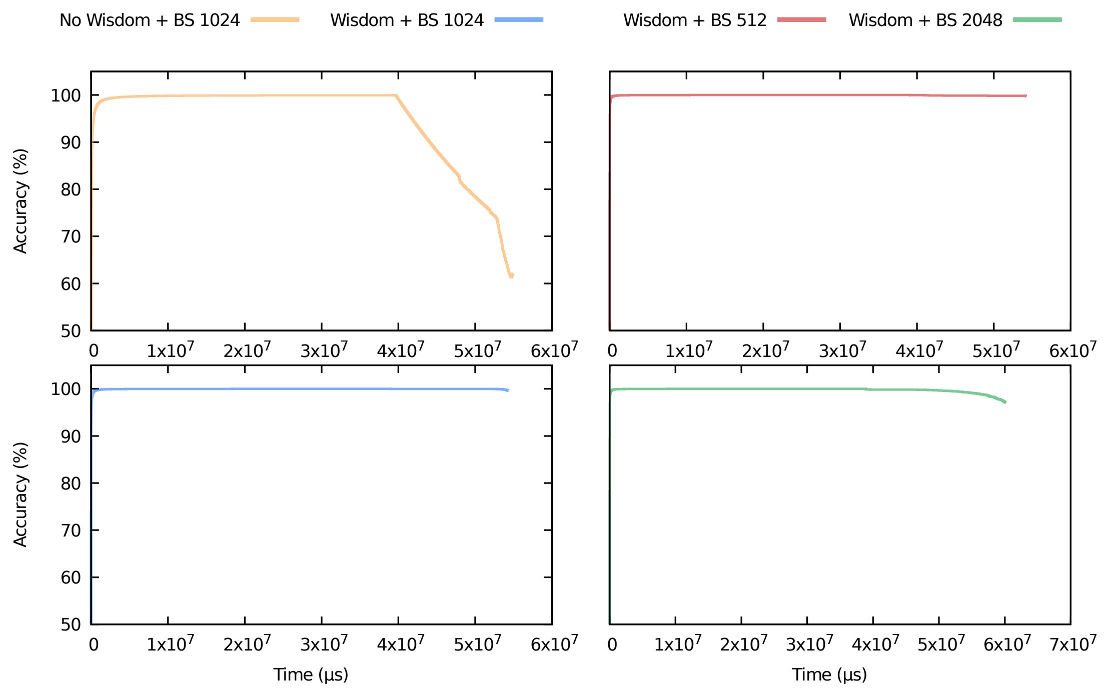


Fig. 10.12: Testing different inputs with the same wisdom information for Cholesky in SSF

Our third execution downgraded the block size from 1024 to **512 by 512** elements. To fully test normalized costs, we also added a fourth execution with a larger block size than the default. The fourth execution had a block size of **2048 by 2048**.

	No Wisdom + BS 1024	Wisdom + BS 1024	Wisdom + BS 512	Wisdom + BS 2048
Average Accuracy	62.11%	99.60%	99.75%	97.08%

Table 10.8: Average accuracy of CPU usage predictions for figure 10.12

The accuracy at each timestep for all these executions is shown in figure 10.12. As shown, normalized costs are effective for all the tested parameters. In the first series we see that after the initialization and matrix transformation operations, there is a drastic change in the accuracy of performance, due to not having predictions for those tasks. However, when using the wisdom file created by the first execution, different block sizes seem to yield almost identical accuracy. We tested both a bigger and smaller block size to make sure it was independent of the growth or decrease in the number of tasks and the granularity of these. Table 10.8 shows the average accuracy of the previous four executions.

10.7 Displaying PQoS Events through Extrae

Since we could access any metric in real-time, we decided to test a proof-of-concept by inserting Extrae custom event calls in the runtime. This is to be able to externalize PQoS events to traces. To do this, we executed Cholesky in SSF using only 8 of the 56 available cores. We decided to limit the number of cores for visualization purposes.

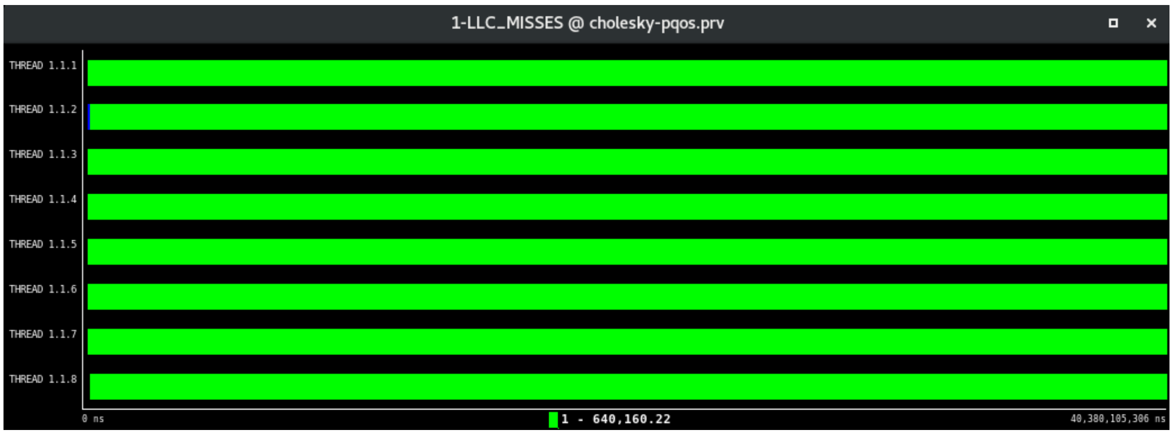


Fig. 10.13: LLC Misses as reported by PQoS through Extrae

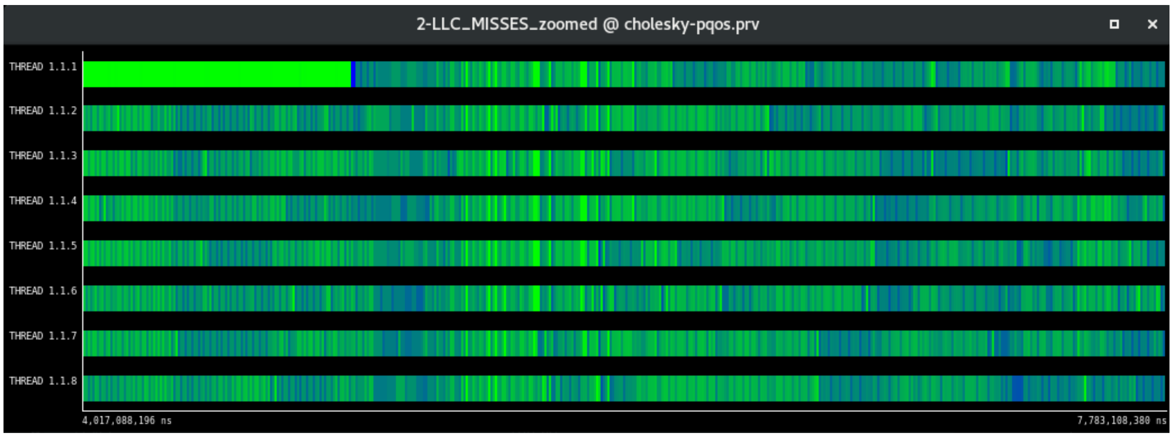


Fig. 10.14: LLC Misses as reported by PQoS through Extrae – Zoomed-in view

Figure 10.13 shows a trace of the execution in a view that displays the number of LLC misses during the whole execution. To visualize how these are output from the runtime, we include figure 10.14, which is a zoomed-in fragment of the first figure (see time scale). As shown, every time a task is created, paused, resumed, or completed, LLC misses (and other events) are polled from the thread and output into the trace. This creates different intervals with different values for the same event.

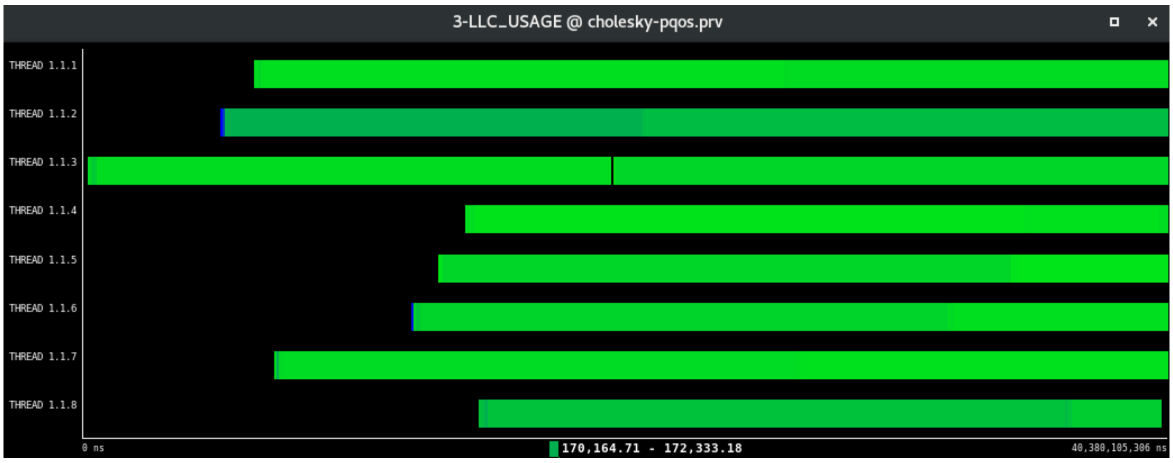


Fig. 10.15: LLC usage as reported by PQoS through Extrae

In figure 10.15 we display the LLC utilization. As shown, threads are prone to allocate a bigger amount of LLC the first time they have to. Memory bandwidth, shown in figure 10.16, is displayed differently. Since it is anothter type of event, only when threads are using memory bandwidth it shows up in traces. When they are not, nothing shows up (dark part).

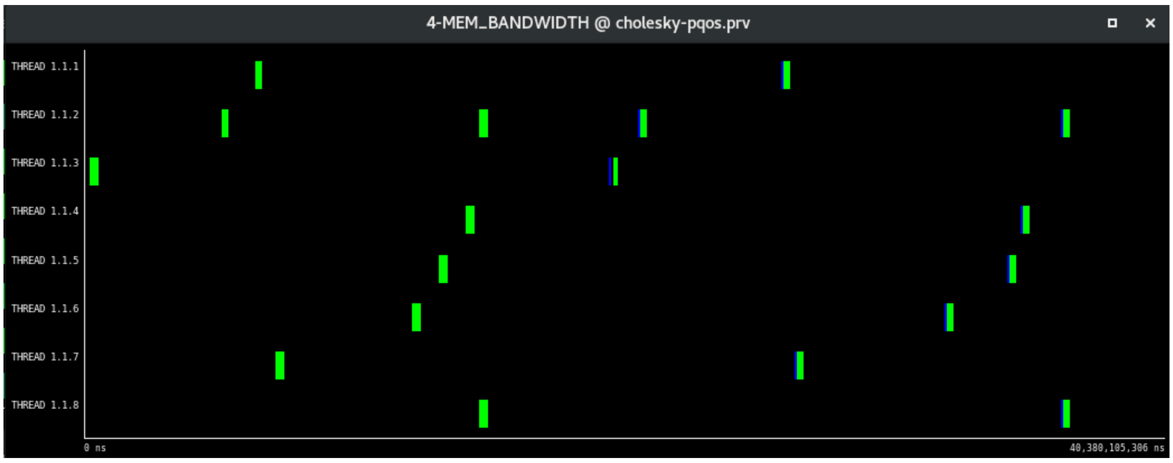


Fig. 10.16: Memory Bandwidth used by threads as reported by PQoS through Extrae

Finally, in figure 10.17 and 10.18 we show the retired instructions and the unhalted cycles, respectively. Not much can be seen in these. However, Extrae allows for views to be combined. By combining these two previous views, we obtain the instructions-per-cycle view, shown below in figure 10.19.

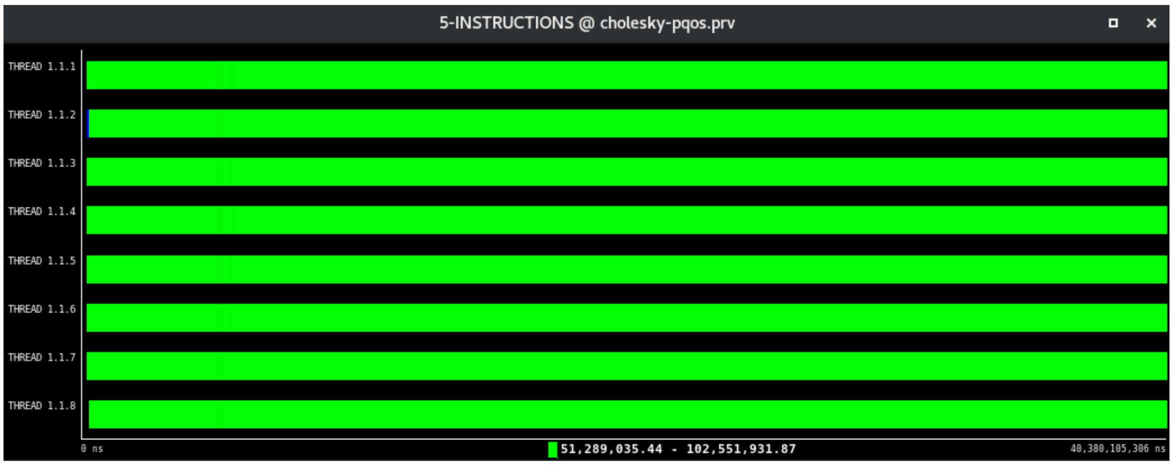


Fig. 10.17: Number of retired instructions over time as reported by PQoS through Extrae

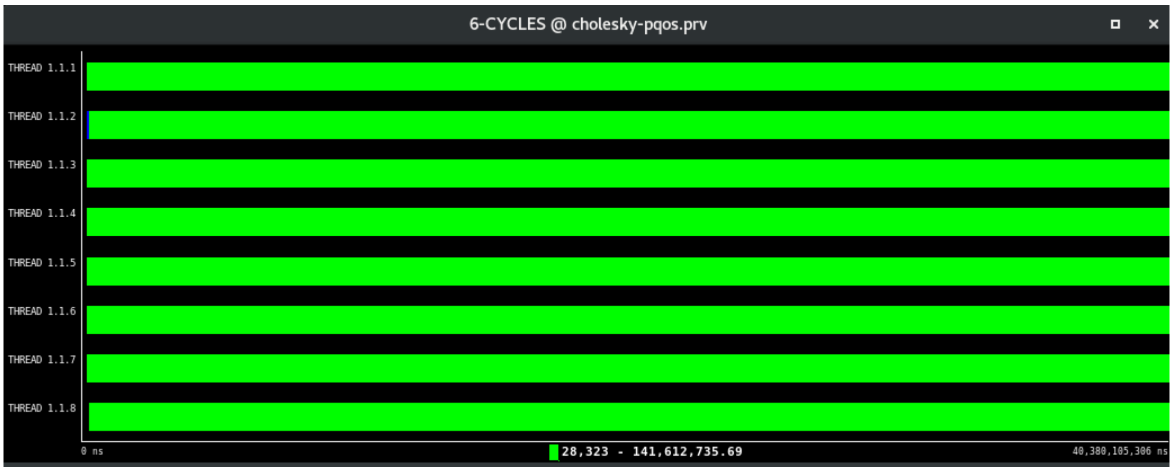
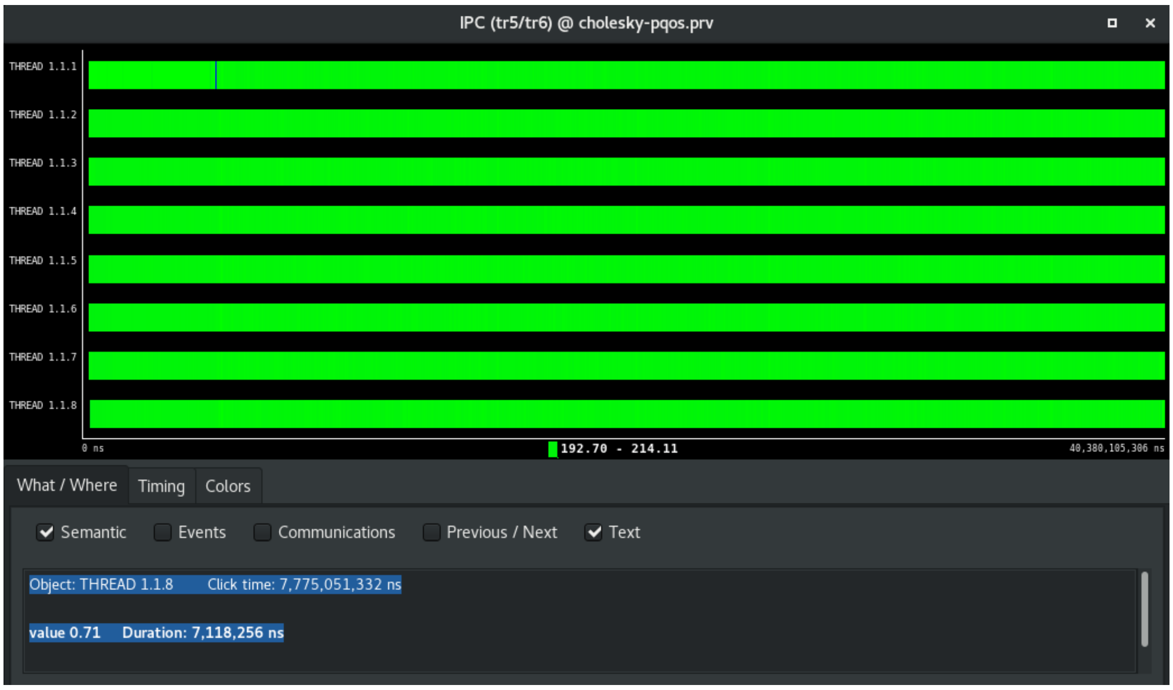


Fig. 10.18: Number of unhalted clock cycles over time as reported by PQoS through Extrae



Object: THREAD 1.1.8 Click time: 7,775,051,332 ns

value 0.71 Duration: 7,118,256 ns

Fig. 10.19: Combination of the views shown in figures 10.18 and 10.17 in order to create the IPC view

As shown, by simply double-clicking at any location in the view, we can obtain the IPC at that exact moment (below the figure). This works seamlessly for all of the events shown previously.

11 | Conclusions

In this thesis, we have studied different techniques through which we can obtain the metrics of a runtime library in a lightweight manner. Using the information gathered from these metrics, we have created tools to monitor all the different elements that form a task-based programming model. Our study can be replicated to any, but we have exemplified our contributions using the OmpSs-2 programming model and the Nanos6 runtime library.

To integrate these tools in the Nanos6 runtime, we created a monitoring infrastructure that controls all the aforementioned tools as separate submodules. For completeness, and to explore other techniques, we integrated another module with the monitoring module. This module measures different hardware events for elements such as cores, threads, or tasks.

With all these metrics, events, and the information gathered through them, we created an infrastructure of predictions. Through hints provided by users to the runtime, all these aforementioned metrics can be normalized to predict future values for tasks, threads, CPUs, and runtime-wise metrics.

We have also demonstrated how both runtimes and underlying systems can benefit from these metrics and predictions. This is through the creation of enhanced scheduling techniques that poll precise predictions to better schedule tasks or to better utilize resources.

To fully understand its potential and accuracy, we have first evaluated timing predictions on tasks. Upon examining preliminary results, we opted to further optimize our approach by introducing a 'wisdom' mechanism that takes advantage of normalized metrics by saving these for future executions. To assess the usability of the predictions, we also put to the test the accuracy of CPU usage predictions with different architectures and various applications. Last, we demonstrated the effectiveness of normalizing metrics through the `cost` clause and the better utilization of resources through new scheduling techniques.

12 | Future Work

As shown in previous sections, our monitoring infrastructure provides runtimes with much information. We have utilized all of them to produce predictions, however hardware-based events can be very useful for other studies.

We plan to categorize tasks by their hardware events using the PQoS module. Also, Intel's® CMT-CAT library brings many possibilities to the table. Two clear examples would be the limited allocation of last level cache occupation and memory bandwidth usage.

Apart from this, as mentioned before, the expression to compute the error of predictions does not take into account things such as the runtime not responding instantly to predictions. Even though there might be a lack of workload in the runtime, CPU utilization does not instantly drop. This causes part of the error in accuracy not to be real.

Finally, through the usage of predictions and normalized cost, we plan to create a scheduler that takes into account the relative computational weight of tasks when scheduling them.

Bibliography

[1] Khang T Nguyen. Cache monitoring technology example data: Application profiling, figure 4, December 2014. URL https://software.intel.com/en-us/blogs/2014/12/11/intels-cache-monitoring-technology-use-models-and-data. vii, 25, 28, 29, 30

[2] Henry Gabb, Richard M. Jackson, and Michael J.E. Sternberg. Modelling protein docking using shape complementarity, electrostatics and biochemical information. *Journal of molecular biology*, 272:106–20, 10 1997. doi: 10.1006/jmbi.1997.1203. 1

[3] William C. Skamarock, Joseph B. Klemp, Jimy Dudhia, David O. Gill, Dale M. Barker, Wei Wang, and Jordan G. Powers. A description of the advanced research wrf version 3. ncar technical note -475+str, 2008. 1

[4] Claude Brezinski. La méthode de cholesky. *Revue d'histoire des mathématiques*, 11 (2):205–238, 2005. URL http://http://www.numdam.org/item/RHM_2005__11_2_205_0. 1

[5] 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*, pages 124–131, Sep. 2009. doi: 10.1109/ICPP.2009.64. 1

[6] Dixie Hisley, Gagan Agrawal, Punyam Satya-narayana, and Lori L. Pollock. Porting and performance evaluation of irregular codes using openmp. *Concurrency - Practice and Experience*, 12:1241–1259, 10 2000. doi: 10.1002/1096-9128(200010)12:12<1241::AID-CPE523>3.0.CO;2-D.

[7] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the plasma numerical library to the openmp standard. *International Journal of Parallel Programming*, 45(3):612–633, Jun 2017. ISSN 1573-7640. doi: 10.1007/s10766-016-0441-6. URL https://doi.org/10.1007/s10766-016-0441-6.

[8] Dimitri Komatitsch, David Michéa, and Gordon Erlebacher. Porting a high-order finite-element earthquake modeling application to nvidia graphics cards using cuda. *Journal of Parallel and Distributed Computing*, 69(5):451 – 460, 2009.

ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2009.01.006. URL http://www.sciencedirect.com/science/article/pii/S0743731509000069. 1

[9] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Müller, and Michael M. Resch. Towards efficient execution of mpi applications on the grid: Porting and optimization issues. *Journal of Grid Computing*, 1(2):133–149, Jun 2003. ISSN 1572-9184. doi: 10.1023/B:GRID.0000024071.12177.91. URL https://doi.org/10.1023/B:GRID.0000024071.12177.91. 1

[10] N. P. Karunadasa and D. N. Ranasinghe. Accelerating high performance applications with cuda and mpi. In *2009 International Conference on Industrial and Information Systems (ICIIS)*, pages 331–336, Dec 2009. doi: 10.1109/ICIINFS.2009.5429842. 1

[11] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2:19 – 25, 2015. ISSN 2352-7110. doi: https://doi.org/10.1016/j.softx.2015.06.001. URL http://www.sciencedirect.com/science/article/pii/S2352711015000059. 1

[12] Message Passing Interface Forum. Mpi: A message-passing interface standard – version 3.1, June 2015. URL https://www.mpi-forum.org/docs/. Accessed: 29-12-2018. 1

[13] OpenMP Architecture Review Board. Openmp application programming interface – version 5.0, November 2018. URL https://www.openmp.org/specifications/. Accessed: 29-12-2018. 1, 5

[14] Barcelona Supercomputing Center. The ompss programming model specification. URL https://pm.bsc.es/ftp/ompss/doc/spec/. Accessed: 29-12-2018. 1

[15] Ron Bekkerman, Mikhail Bilenko, and John Langford. Scaling up machine learning: Parallel and distributed approaches. 08 2011. doi: 10.1145/2107736.2107740. 1

[16] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence, UAI 2010*, 06 2010. 1

[17] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 164–177, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40047-6. 5

[18] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *2008 37th*

International Conference on Parallel Processing, pages 536–545, Sep. 2008. doi: 10.1109/ICPP.2008.88. 5

[19] Eduard Ayguadé, James Beyer, Alejandro Duran, Roger Ferrer, Grant Haab, Kelvin Li, and Federico Massaioli. An extension to improve openmp tasking control. In Mitsuhisa Sato, Toshihiro Hanawa, Matthias S. Müller, Barbara M. Chapman, and Bronis R. de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, pages 56–69, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13217-9. 5

[20] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL http://dl.acm.org/citation.cfm?id=1413370.1413407. 5, 6

[21] E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991. ISSN 1045-9219. doi: 10.1109/71.86103. 5

[22] Antoni Navarro, Sergi Mateo, Josep Maria Perez, Vicenç Beltran, and Eduard Ayguadé. Adaptive and architecture-independent task granularity for recursive applications. In Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller, editors, *Scaling OpenMP for Exascale Performance and Portability*, pages 169–182, Cham, 2017. Springer International Publishing. ISBN 978-3-319-65578-9. 5, 6, 37, 66

[23] Gad Aharoni, Dror G. Feitelson, and Amnon Barak. A run-time algorithm for managing the granularity of parallel functional programs. *Journal of Functional Programming*, 2(4):387–405, 1992. doi: 10.1017/S0956796800000484. 6

[24] Eduard Ayguadé, Bob Blainey, Alejandro Duran, Jesús Labarta, Francisco Martínez, Xavier Martorell, and Raúl Silvera. Is the schedule clause really necessary in openmp? In Michael J. Voss, editor, *OpenMP Shared Memory Parallel Programming*, pages 147–159, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45009-2. 6

[25] Fahimeh Farahnakian, Pasi Liljeberg, and Juha Plosila. Lircup: Linear regression based cpu usage prediction algorithm for live migration of virtual machines in data centers. pages 358–364, 09 2013. doi: 10.1109/SEAA.2013.23. 6

[26] Marina Kudinova, Anna Melekhova, and Alexander Verinov. Cpu utilization prediction methods overview. In *Proceedings of the 11th Central &#38; Eastern European Software Engineering Conference in Russia*, CEE-SECR '15, pages 7:1–7:10, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4130-1. doi: 10.1145/2855667.2855675. URL http://doi.acm.org/10.1145/2855667.2855675. 6

[27] S. M. Sadjadi, Shu Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. J. Collazo-Mojica. A modeling approach for estimating execution time of long-running scientific applications. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, April 2008. doi: 10.1109/IPDPS.2008.4536214. 6

[28] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155 – 162, 2012. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2011.05.027. URL http://www.sciencedirect.com/science/article/pii/S0167739X11001129. 6

[29] A. Qawasmeh, A. M. Malik, and B. M. Chapman. Adaptive openmp task scheduling using runtime apis and machine learning. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 889–895, Dec 2015. doi: 10.1109/ICMLA.2015.111. 6

[30] Barcelona Supercomputing Center. Ompss-2 specification, 2018. URL https://pm.bsc.es/ftp/ompss-2/doc/spec/. Accessed: 29-12-2018. 7, 8, 13, 16, 17, 20

[31] Barcelona Supercomputing Center. Barcelona supercomputing center's website, 2018. URL https://www.bsc.es/. Accessed: 29-12-2018. 10, 20

[32] Barcelona Supercomputing Center Programming Models Group. The nanos6 runtime repository, 2018. URL https://github.com/bsc-pm/nanos6. Accessed: 29-12-2018. 20

[33] Intel®. Intel® resource director technology, 2018. URL https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html. Accessed: 29-12-2018. 24, 27

[34] Intel®. Intel cmt cat repository, 2018. URL https://github.com/intel/intel-cmt-cat/. Accessed: 29-12-2018. 26

[35] Barcelona Supercomputing Center Programming Models Group. The mercurium compiler, 2018. URL https://pm.bsc.es/mcxx. Accessed: 29-12-2018. 32

[36] Barcelona Supercomputing Center Performance Tools. Paraver, 2018. URL https://tools.bsc.es/paraver. Accessed: 29-12-2018. 32

[37] Kent Beck, James Grenning, Robert C. Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Arie van Bennekum, Andrew Hunt, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, Jon Kern, Dave Thomas, Martin Fowler, and Brian Marick. Manifesto for agile software development, 2001. 35

[38] FFTW Authors. Fftw repository, 2018. URL https://github.com/FFTW/fftw3. Accessed: 29-12-2018. 38

[39] N Gunther. Unix load average part 1: How it works, 2010. URL http://www.perfdynamics.com/Papers/la1.pdf. 45

[40] Mike Loukides, Tim O'Reilly, Jerry Peek, and Shelley Powers. *UNIX Power Tools*. O'Reilly Media, February 2009. 45

[41] Adrian Cockcroft and Richard Pettit. *Sun Performance and Tuning (2Nd Ed.): Java and the Internet*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. ISBN 0-13-095249-4. 45

[42] Eric Niebler. Accumulators in the boost library, 2018. URL https://www.boost.org/doc/libs/1_65_0/doc/html/accumulators.html. Accessed: 29-12-2018. 58

[43] Wikipedia contributors. Mean percentage error, 2018. URL https://en.wikipedia.org/wiki/Mean_percentage_error. Accessed: 29-12-2018. 63

[44] Wikipedia contributors. Symmetric mean absolute percentage error, 2018. URL https://en.wikipedia.org/wiki/Symmetric_mean_absolute_percentage_error. Accessed: 29-12-2018. 64

[45] Wikipedia contributors. Relative change and difference, 2018. URL https://en.wikipedia.org/wiki/Relative_change_and_difference. Accessed: 29-12-2018. 64

[46] Leo Törnqvist, Pentti Vartia, and Yrjö O. Vartia. How should relative changes be measured? *The American Statistician*, 39(1):43–46, 1985. doi: 10.1080/00031305.1985.10479385. URL https://doi.org/10.1080/00031305.1985.10479385. 64

[47] Slurm. Online documentation of slurm, v. 18.08. 2018. URL https://slurm.schedmd.com/. Accessed: 29-12-2018. 67

A | The Cholesky Factorization

In this chapter we display the complete code of the blocked Cholesky factorization we used for our evaluations. We believe this is one of the most important benchmarks we used. Users have control over decisions such as:

- Whether the matrix is created from scratch or loaded from a file.
- Whether the matrix initialized is saved into a file.
- Whether there is a check of the factorization at the end of the execution.

The only current constraint is that the input size (matrix leading dimension) must be correctly divided by the block size specified (`input_size % block_size == 0`).

```c
// "util.h" defines other less important functions such as
// initialization tasks and layout-transforming tasks
#include "util.h"

void oss_potrf(int nblocks, size_t bsize,
               int i, int j, double A[nblocks][nblocks][bsize][bsize])
{
  #pragma oss task label(potrf) cost((1.0/3.0) * bsize*bsize*bsize)
  inout(A[i][j])
  {
    lapack_int error = LAPACKE_dpotrf(
      LAPACK_COL_MAJOR,
      'L', bsize,
      &A[i][j][0][0], bsize
    );
    assert(error == 0);
  }
}

void oss_trsm(int nblocks, size_t bsize,
              int ai, int aj, double A[nblocks][nblocks][bsize][bsize],
              int bi, int bj, double B[nblocks][nblocks][bsize][bsize])
{
  #pragma oss task label(trsm) cost(bsize*bsize)
  in(A[ai][aj])
  inout(B[bi][bj])
  cblas_dtrsm(
    CblasColMajor,
    CblasRight,
    CblasLower,
    CblasTrans,
    CblasNonUnit,
    bsize, bsize, 1.0,
    &A[ai][aj][0][0], bsize,
    &B[bi][bj][0][0], bsize
  );
}

void oss_syrk(int nblocks, size_t bsize,
              int ai, int aj, double A[nblocks][nblocks][bsize][bsize],
              int bi, int bj, double B[nblocks][nblocks][bsize][bsize])
{
  #pragma oss task label(syrk) cost(bsize*bsize*bsize)
  in(A[ai][aj])
  inout(B[bi][bj])
  cblas_dsyrk(
    CblasColMajor,
    CblasLower,
    CblasNoTrans,
    bsize, bsize, -1.0,
    &A[ai][aj][0][0], bsize,
    1.0,
    &B[bi][bj][0][0], bsize
  );
}

void oss_gemm(int nblocks, size_t bsize,
              int ai, int aj, double A[nblocks][nblocks][bsize][bsize],
              int bi, int bj, double B[nblocks][nblocks][bsize][bsize],
              int ci, int cj, double C[nblocks][nblocks][bsize][bsize])
{
  #pragma oss task label(gemm) cost(bsize*bsize*bsize)
  in(A[ai][aj])
  in(B[bi][bj])
  inout(C[ci][cj])
  cblas_dgemm(
    CblasColMajor,
    CblasNoTrans,
    CblasTrans,
    bsize, bsize, bsize, -1.0,
    &A[ai][aj][0][0], bsize,
    &B[bi][bj][0][0], bsize,
    1.0,
    &C[ci][cj][0][0], bsize
  );
}


void cholesky(int nblocks, size_t bsize, double A[nblocks][nblocks][bsize][
    bsize])
{
  int i, j, k;
  for (i = 0; i < nblocks; ++i) {
    // Diagonal Block Factorization
    oss_potrf(nblocks, bsize, i, i, A);

    // Triangular Systems
    for (j = i + 1; j < nblocks; ++j) {
      oss_trsm(nblocks, bsize, i, i, A, i, j, A);
    }

    // Update Trailing Matrix
    for (j = i + 1; j < nblocks; ++j) {
      for (k = i + 1; k < j; ++k) {
        oss_gemm(nblocks, bsize, i, j, A, i, k, A, k, j, A);
      }
      oss_syrk(nblocks, bsize, i, j, A, j, j, A);
    }
  }
}


int main(int argc, char* argv[])
{
  setbuf(stdout, NULL); // Do not buffer prints
  if (argc < 4 || argc > 5) {
    printf("\nUsage: ./cholesky <size> <block_size> <check> [out]\n");
    printf(" size : Matrix' order (size x size)\n");
    printf(" block_size : Blocking factor for the matrix.\n");
    printf(" check : Whether to check the factorization's result (1/0).\n");
    printf(" out : [optional] The file name where the matrix used will be
        written for future usage.\n");
    printf("size % block_size = 0\n");
    printf("Different block sizes may generate different matrixes depending
        on the test, use the 'out' option with caution.\n\n");
    exit(1);
  }

  const size_t ld = atoi(argv[1]); // Matrix size
```

```c
const size_t bsize = atoi(argv[2]); // Block size
const int nblocks = ld / bsize; // Number of blocks
const size_t len = ld*ld*sizeof(double);
const int check = atoi(argv[3]); // Factorization check

int out = 0;
char * filename;
if (argc == 5) {
  out = 1;
  filename = argv[4];
}

assert(ld % bsize == 0);
assert(check == 1 || check == 0);

// Allocate matrixes
double (*matrix)[nblocks][bsize][bsize] = malloc(len);
assert(matrix != NULL);
double (*original_matrix)[nblocks][bsize][bsize] = malloc(len);
assert(original_matrix != NULL);

// Init matrix
initialize_matrix(nblocks, bsize, (double *) original_matrix, out,
    filename);

// Transform from flat (original_matrix) to tile (matrix)
flat2tile(nblocks, bsize, (double *) original_matrix, matrix);

#pragma oss taskwait

struct timeval start, stop;
gettimeofday(&start, NULL);

// Compute cholesky factorization
printf("Executing the factorization...\n");
#pragma oss task label(cholesky) cost((1.0/3.0)*ld*ld*ld)
inout(matrix[0;nblocks][0;nblocks][0;bsize][0;bsize])
cholesky(nblocks, bsize, matrix);

#pragma oss taskwait

gettimeofday(&stop, NULL);
float elapsed = 1000000.0 * (stop.tv_sec - start.tv_sec);
elapsed += stop.tv_usec - start.tv_usec;
elapsed /= 1000000.0;

float gflops = ((1.0/3.0) * ld*ld*ld) / (elapsed * 1.0e+9);

```

```
  if (check) {
    // Allocate new matrix to transform it from tiled to flat
    double (*factorized_matrix)[nblocks][bsize][bsize] = malloc(len);
    assert(factorized_matrix != NULL);

    // Transform from tile (matrix) to flat (factorized_matrix)
    tile2flat(nblocks, bsize, matrix, (double *) factorized_matrix);

    #pragma oss taskwait

    // Check factorization
    const double EPS = BLAS_dfpinfo( blas_eps );
    check_factorization(ld, (double *) original_matrix, (double *)
        factorized_matrix, ld, EPS);
    free(factorized_matrix);
  }

  // Print results
  printf("\n");
  printf("%s: %s\n" , "BENCHMARK" , "cholesky");
  printf("%s: %llu\n", "SIZE" , ld);
  printf("%s: %f\n" , "PERFORMANCE(GFlops)", gflops);
  printf("%s: %d\n" , "BLOCK_SIZE" , bsize);
  printf("%s: %f\n" , "TIME(s)" , elapsed);
  if (out) {
    printf("%s: %s\n", "MATRIX_FILE_NAME" , filename);
  }

  free(matrix);
  free(original_matrix);
}
```

Code A.1: Blocked Cholesky Factorization code using OmpSs-2 directives.