Simplifying Self-Adaptive and Power-Aware Computing with Nornir

Daniele De Sensi, Tiziano De Matteis, Marco Danelutto

Department of Computer Science, University of Pisa, Italy
Largo B. Pontecorvo 3, I-56127 Pisa, Italy

Abstract

Self-adaptation is an emerging requirement in parallel computing. It enables the dynamic selection of resources to allocate to the application in order to meet performance and power consumption requirements. This is particularly relevant in *Fog Applications*, where data is generated by a number of devices at a varying rate, according to users' activity. By dynamically selecting the appropriate number of resources it is possible, for example, to use at each time step the minimum amount of resources needed to process the incoming data.

Implementing such kind of algorithms may be a complex task, due to low-level interactions with the underlying hardware and to non-intrusive and low-overhead monitoring of the applications. For these reasons, in this paper we propose Nornir, a C++-based framework, which can be used to enforce performance and power consumption constraints on parallel applications running on shared memory multicores. The framework can be easily customized by algorithm designers to implement new self-adaptive policies. By instrumenting the applications in the PARSEC benchmark, we provide to strategy designers a wide set of applications already interfaced to Nornir. In addition to this, to prove its flexibility, we implemented and compared several state-of-the-art existing policies, showing that Nornir can also be used to easily analyze different algorithms and to provide useful insights on them.

Keywords: Self-Adaptive, Power-Aware, Quality of Service, Data Stream Processing, Fog Computing, Parallel Computing

1. Introduction

Nowadays, the amount of data produced by *internet-connected* devices is constantly growing. *Data Steam Processing* (DaSP) applications deal with the processing of this continuous and, often, infinite flows of information. Usually, these applications have to face different challenges concerning *Quality of Service* (QoS) expectations from end users and developers. For this reason, they exploit parallel and distributed hardware to cope with the high volume of incoming data in a quasi-real time fashion. In addition to this, DaSP applications are affected by highly variable arrival rates and changes in their workload characteristics. Due to economic costs, resource availability and resource sharing issues, using all the available resources is often not the right choice. Moreover, power consumption management has become a major concern for data centers and using computing facilities that could be otherwise allocated to other jobs is a waste of resources. Therefore, *Self-Adaptivity* (sometimes referred as *autonomicity* or *elasticity*) is a fundamental feature: applications must be able to autonomously adjust their resources usage (i.e. their *configuration*) to accommodate dynamic requirements and workload variations by maintaining the desired QoS in terms of performance and/or power consumption.

These issues are particularly relevant in the area of *Fog Computing*, where data is generated at varying rates according to users' activity. Being able to explicitly control power consumption and performance of an application is fundamental to decide if some parts of the computation need to be offloaded to the cloud [1]. Furthermore, sensors and edge devices often have *energy harvesting* capabilities [2], i.e. they can harvest power from renewable energy sources, like from embedded solar panels. In such cases, it may be useful to maximize performance while not consuming more energy than that harvested from the environment [3].

Controlling performance and power consumption is not only relevant for streaming applications but also for classical *batch* applications. Indeed, by finding proper tradeoffs between performance and power consumption it would be possible, for example, to explicitly control the battery life of mobile devices (e.g. smartphones).

Programming tools for facing these issues are missing in existing *Stream Processing Systems* (SPSs) and *Parallel Programming Frameworks*. Currently, users and applications programmers have to manually decide when to change the operating conditions of the deployed applications. Moreover, despite many self-adaptive strategies have been recently proposed (e.g. [4, 5, 6, 7, 8]), implementing such strategies is a cumbersome and error-prone duty for developers. Indeed, they have to deal with many architectural low-level issues related to hard-

Email addresses: `desensi@di.unipi.it` (Daniele De Sensi), `dematteis@di.unipi.it` (Tiziano De Matteis), `marco.danelutto@unipi.it` (Marco Danelutto)

ware management mechanisms like voltage, frequency, cores topology, etc. Even interfacing with applications for monitoring purposes may not be an easy task. We believe that this is one of the reasons why these strategies are usually only simulated on post-mortem data instead of being validated on actual application executions. Despite a simulation can provide a first approximation of the accuracy of the algorithm, it is difficult to precisely estimate the run-time overhead and the effectiveness of these methods. Even when such algorithms are actually executed, they are often validated on a specific application or runtime [4]. Since the logic for implementing the self-adaptivity is embedded inside the application code, it would be difficult to use the same implementation of the algorithm for different applications. As a consequence, comparing new self-adaptive strategies with existing ones would be a difficult task.

For these reasons, in this paper we present Nornir, a customizable C++ framework for self-adaptive and power-aware computing[1]. On one side, Nornir can be used to enforce specific performance and power consumption requirements on parallel applications, by using some self-adaptive strategies already provided by the framework. On the other side, Nornir can be customized by adding new self-adaptive algorithms. This would allow a designer (i.e. the person in charge of creating new self-adaptive strategies) to just focus on the algorithm, by exploiting the infrastructure provided by the framework to interact with the application and with the underlying computing system. We believe that this is a fundamental step for rapidly prototyping new self-adaptive techniques and for their wide adoption. In addition, it would be possible for the designers to easily compare their new algorithms with those already provided by Nornir.

As additional contributions, we provide a wide set of applications already interfaced to Nornir, by instrumenting the applications in the PARSEC [9] benchmark. By doing so, we allow the designers of new self-adaptive strategies to easily test and validate their algorithms. The instrumented applications have been integrated in the existing P^{3}ARSEC benchmark suite [10] and released as open source.

Eventually, to prove the flexibility of our approach, we used Nornir to implement some state-of-the-art self-adaptive algorithms and to test them over the instrumented PARSEC benchmarks. This allowed us to compare these algorithms (some of which were originally only tested on post-mortem data) and to show advantages and disadvantages of each of them.

The paper is organized as follows. In Section 2 we provide the background and we describe some state of the art approaches. In Section 3 we depict the design of Nornir and in Section 4 we show how application users can use Nornir to enforce performance and power consumption requirements on their applications. In Section 5 we discuss the possibilities an application programmer has for connecting an application to Nornir and in Section 6 we outline how Nornir can be customized by adding new self-adaptive algorithms. After that, in Section 7 we show how to extend Nornir to support, for example, other runtimes for parallel applications. We briefly describe the applications we already interfaced to Nornir in Section 8, and we will use them in Section 9 to compare some existing self-adaptive algorithms and to evaluate how they behave when they are actually executed instead of being just simulated. Finally, in Section 10 we draw conclusions.

2. Background and Related Work

Self-adaptive systems are able to alter their behavior according to QoS requirements and to the surrounding conditions in order to achieve some goal, without any human intervention. Altering the behavior usually implies changing the *configuration* of the application, e.g. the amount of used resources.

Self-adaptive solutions are usually time-driven and, at each time step, act by following a generic *Monitor-Analyze-Plan-Execute* (MAPE) loop [11]. In the *Monitor* phase, various measurements are collected from the application (e.g. performance and power consumption). In the *Analyze* phase, monitored data collected at the current and previous time steps, is compared against the user's requirements. If requirements are violated, in the *Plan* phase new optimal resources allocation are computed. The planned decisions are applied to the application during the *Execute* phase, by acting on proper *actuators*.

Different self-adaptive strategies have been proposed to satisfy user's requirements in terms of performance ([6, 4, 8, 12]), power consumption ([7]) or both of them ([5, 13]). Such requirements are usually enforced even in presence of workload fluctuations or external interferences. However, in many cases, these techniques are only simulated or implemented for specific applications.

In the literature, some proposed framework target a problem similar to the one we are addressing in this work [14, 15, 16]. However, they provide very limited customization opportunities, are quite outdated and the source code is not publicly available.

Adam [17] allows the customization of the *plan* phase but the *execute* and *monitor* phases are fixed, allowing the interaction with only two actuators (DVFS and Thread Packing).

SEEC [18], *Bard* [19] and *Poet* [20] allow the customization of the *monitor* and *execute* phases of the MAPE loop, but provide their own *plan* algorithm. Albeit being a flexible strategy, it is not possible to replace it with a different one. This is an important limitation, since in some cases the designer may exploit some knowledge about the application or the architecture to design more efficient *plan* algorithms with respect to those already provided by the framework. Moreover, *Bard* and *Poet* require the user to know a priori the performance and power consumption of the application in each possible configuration. This is a cumbersome task for the user, since it needs to execute the application in all its possible configurations. Furthermore, such frameworks cannot be used if the applications to be executed are not known a beforehand.

IBM StreamS [21] is a commercial and proprietary framework for the development and deployment of (distributed)

[1]The framework is released under open source license and publicly available at `http://danieledesensi.github.io/nornir/`

stream processing applications. Complex applications are expressed by means of computational graphs of *Processing Elements* (PEs) that cooperate by exchanging data. Each PE can be internally parallelized exploiting a certain number of threads. StreamS may change at runtime the number of used threads to maximize throughput (using its own strategy [22]), but it does not allow users to enforce a specific throughput or power consumption for the application or to define custom decision strategies.

With respect to the mentioned solutions, Nornir is a completely customizable and publicly available framework. It allows runtime developer to modify the *monitoring* and *execute* phases. In addition to this, it allows the designer of the *plan* strategy to just focus on the algorithm, exploiting the infrastructure provided by the framework to collect the data and to apply the decisions. This is a fundamental step to quickly prototype and validate customized planning strategies and to easily compare them with existing ones.

3. Nornir Design

Nornir is a customizable C++ framework for self-adaptive and power-aware parallel applications on shared memory multicore machines. It can be used to reconfigure parallel applications to enforce specific performance and power consumption requirements and, at the same time, can be customized by adding new self-adaptive strategies. Its general architecture is depicted in Figure 1.

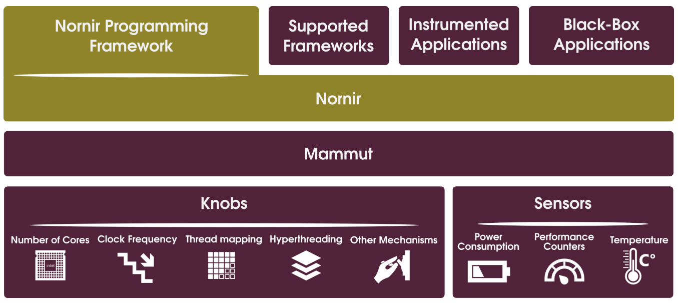


Figure 1: General Architecture of Nornir Framework.

Nornir can control different types of applications (upper layer of Figure 1). To reach this goal, Nornir couples an application with a Manager, which is in charge of driving application execution to enforce the requirements expressed by the user. At each sampling interval (also known as *control step*), the Manager acts by executing an iteration of the so-called *Monitor-Analyze-Plan-Execute* (MAPE) loop [11].

In the *monitor* phase, data is collected from “sensors” on the computing node and on the application (e.g. power consumption of the system, performance of the application, etc...). Such data is aggregated, for example by averaging the monitored samples over a time interval, and then passed to the *analyze* phase. If the monitored data is not compliant with the user's requirement, the *plan* phase is triggered to decide the actions to be executed to achieve the goals specified by the user. Such actions can be very simple (e.g. scale down the clock frequency of the CPU) or could also be expressed as complex operations. Eventually, the *execute* phase applies the reconfiguration plan by using appropriate “actuators” (also known as “knobs”). Such actuators may be present on both the computing node and the application. To implement the *monitor* and *execute phases*, Nornir interacts with the system knobs and sensors (e.g. power consumption one) by using the Mammut [23] library[2].

A common underlying assumption in these approaches is the *iterative* nature of the managed application [24, 18, 25]. With *iterative* we refer to an application that performs roughly the same (or very similar) computation in each iteration or over a set of contiguous iterations, thus exhibiting a certain degree of repetitive behavior. This is a crucial property for the execution of the MAPE loop, since the manager can assume that the decisions taken at the current control step will still be valid at the next control step, since in the meanwhile the characteristics of the computation have not significantly changed. Many real applications fall into this category [25, 26, 24, 27, 10]. It is worth noting that the application may still be characterized by different *phases*, unless there are extreme behaviors like one different phase at each control step. However, this is not the case in many real applications [28], which are in general characterized by an iterative behavior and by a small number of different phases.

Nornir features can be summarized as follows. It is:

- ready-to-use: the application *users* can express QoS requirements on a controlled application by choosing a set of available adaptation strategies. A Manager will control the application. How this interaction take place is responsibility of the application *programmer* that can exploit different possibilities (Section 5);
- customizable: *strategy designers* can focus on the implementation of their new self-adaptive strategy by using the provided set of resource management mechanisms and the application monitoring infrastructure (Section 6);
- extendable: support to additional parallel programming framework or monitoring tools can be easily added (Section 7).

In the next sections we will details how these different actors can interact with Nornir.

4. The User Perspective

Given an application connected to Nornir, the *user* can express specific requirements on that application, regarding performance or power consumption. If the application is modular, like for example in *data stream processing*, where the application may be composed by multiple co-running parallel operators, we would need to coordinate such operators. For example,

[2]Mammut is an object-oriented C++ framework allowing a transparent and portable monitoring of system sensors as well as management of several system knobs. It is publicly available at: `http://danieledesensi.github.io/mammut/`

an operator may decide to increase the clock frequency while the other decides to decrease it. In such a case, a coordination is required to pick the best action for both operators. However, coordination of multiple operators (or multiple parallel applications) will be part of our future work. It is worth noting that we can still manage the cases where the operators are not co-running (e.g. one parallel operator starts only when the previous one terminated) as well as applications composed by multiple phases.

We classify applications into two types:

Streaming Applications These applications receive a “stream” of data, i.e. a continuous flow of elements. The elements to be processed are not already available but will be received at a possibly variable rate. The same function is applied over each received element, or on a window (i.e. a subset) of recently received elements. The number of elements to be received may be *finite* or *infinite* (i.e. the application could possibly run “forever”).

Batch Applications This class includes all the applications that need to process data which are already available and can be accessed by the application at any time

It is possible to specify different types of requirements, on the metrics we report in Table 1.

The most appropriate requirement depends on the application and on the user preferences. For example, in streaming applications the user may want to use a number of resources proportional to the current workload. To reach this goal, it is sufficient to require a *utilization factor* less than 1, while minimizing the power consumption. This ensures that all the stream elements will be timely processed with the minimum power consumption possible. In other scenarios, different requirements may be more appropriate. For example, consider an application performing a nightly backup of all the data produced during the day in a big company. In such case, we may want to set a maximum execution time for this process (i.e. to terminate before the arrival of employees in the morning), while consuming as less energy as possible.

The user details the constraints that should be enforced on the application by specifying them in an `XML` file. Listing 1 shows an example of requirements which can be used to ask Nornir to find a configuration characterized by a power consumption lower than 50 Watts and a latency lower than 30 ms. Since more than one configuration could have such characteristics, the user wants Nornir to select the one characterized by the highest throughput.

By using the configuration file, the user can also specify other optional parameters such as the length of the control step, the self-adaptive strategy to be used, which control knobs can be used by the algorithm and other parameters specific to the used reconfiguration algorithm.

If the application is executed on a public cloud, we need to consider that the user of the application and the cloud provider may have different goals, and that the user may not explicitly monitor and control the power consumption. Moreover, the user may not have access to some knobs (e.g. frequency scaling). In such a case, Nornir can still be used by the application user, by only acting on the knobs he has access to (e.g. number of threads), to tune and control the performance of the application. Alternatively, Nornir could also be used by the cloud operator, to control the power consumption of the applications ran by the users and to ensure that the available power budget is not exceeded.

Lastly, datacenter providers are considering the possibility to provide economical incentives to the user if they are willing to reduce their power consumption [29, 30]. In such a case, after a negotiation with the users, datacenter operators could use Nornir to constraint the power consumption of users applications.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nornirParameters>
    <requirements>
        <throughput>MAX</throughput>
        <latency>30</latency>
        <powerConsumption>50</powerConsumption>
    <requirements>
</nornirParameters>
```

Listing 1: Example of user requirements.

5. The Application Programmer

As anticipated, to control the application the Manager may need to interact with the application during the *monitor* and *execute* stages of the MAPE loop. To perform the interaction, the application *programmer* needs to attach a Manager to the *user*’s application. The Manager runs in a separate thread/process and interacts with the application to gather monitoring data and to apply reconfiguration decisions (e.g. changing the number of threads), enforcing the *user*’s requirements. Nornir offers different possibilities to application *programmers* for realizing this interaction, allowing to chose the desired trade-off between configuration optimality and required programming effort. In general, more intrusive approaches collect more precise metrics and leads to better solutions, while requiring a higher effort to the *programmer*. Furthermore, such solutions allow Nornir to access some actuators which may be specific to the application or to the runtime system, extending the range of possible configurations and allowing the self-adaptive algorithms to take better decisions. On the other hand, some approaches allow users to directly interface applications to Nornir, without any *programmer* intervention, despite this may lead to suboptimal decisions during the *plan* phase.

In Figure 2 we depict a flowchart showing the different possibilities available to the programmer and the sections where they are described. In the following, we will discuss the different opportunities, starting from the less optimal and less intrusive ones.

Metric	S	Description (Streaming)	Description(Batch)
Throughput	R	Number of stream elements processed per second.	Number of iterations executed per second.
Latency	L	Time required to process a single stream element.	Time required to perform a single iteration.
Completion Time	T	Time required to process all the elements on the stream. The user needs to specify the expected length of the stream. By doing so, NORNIR can estimate the completion time as the ratio between the number of the number of elements still to be processed and the current throughput. For this reason, this requirement can only be specified when the length of the stream is known a priori.	Time required to perform all the iterations. Similarly to the *streaming* case, the number of iterations need to be known a priori.
Utilization Factor	ρ	Represents the utilization of the application, i.e. the fraction of time spent processing stream elements (between 0 and 1). $1-\rho$ is the fraction of time wasted by the application waiting for new data to arrive from the stream. A low ρ means that the resources allocated to the application are not fully utilized. Note that this definition is slightly different from the classical queueing theory one, where ρ could also be greater than 1.	Not applicable on *batch* applications since ρ is always 1 (because the data is always available to the application and it never needs to wait for new data to arrive).
Power Consumption	P	Instantaneous power consumption. Since current operating systems do not provide mechanisms to monitor the individual power consumption of each application, this may correspond to the system power consumption.	The same as streaming.
Energy	E	Power integrated over time. It can be both specified as energy required to process a single stream element (or iteration in batch applications) or to process all the elements. In the latter case, the number of elements to be received (or the number of iterations to be executed) must be known a priori, since energy will be estimated as $E = P \times T$.	The same as streaming.

Table 1: Performance requirements that can be specified by the *user*.

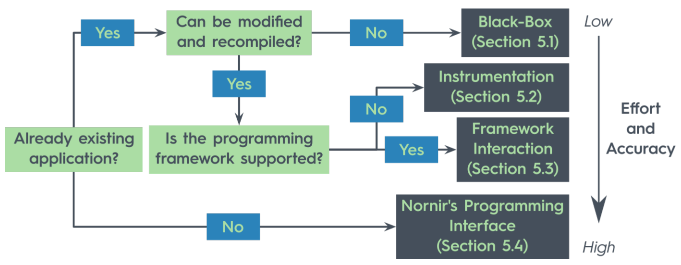

Already existing application?
Yes
Can be modified and recompiled?
No
Black-Box (Section 5.1)
Yes
Is the programming framework supported?
No
Instrumentation (Section 5.2)
Yes
Framework Interaction (Section 5.3)
No
Nornir's Programming Interface (Section 5.4)
Low
Effort and Accuracy
High

Figure 2: Flowchart describing the possible choices for the application programmer to interface NORNIR manager to an application.

5.1. *Black-Box Interaction*

The simplest solution is to use NORNIR on an existing application without any modification to the code and without any programmer intervention. In some cases this may be the only feasible solution since the *programmer* may not have the possibility to modify and/or recompile the application. A *user* can specify constraints on the `foobar` application by using the NORNIR applications launcher, as shown in Listing 2.

```
manager-blackbox --parameters parameters.xml
                 --application ./foobar
```

Listing 2: Example of attachment of a NORNIR MANAGER to an already existing application.

The NORNIR MANAGER will run in a separate process and will not interact directly with the application. In this case NORNIR can monitor the application only by relying on performance counters, for example by monitoring the number of assembler instructions executed per time unit (i.e. instructions per second, IPS). Since it is not possible for NORNIR to monitor detailed metrics like latency and throughput, the *user* can only express performance requirements for the application in terms of IPS. Correlating the IPS to the actual application throughput is not an easy task and not very intuitive from the *user* perspective. Moreover, as shown in [18, 31, 32] performance counters may not be a good performance proxy since they are not always strictly correlated to the actual application-level performance. For these reasons, this approach should be used only if none of the other solutions (Sections 5.2, 5.3 and 5.4) can be adopted.

5.2. *Instrumentation*

If the source code of the application can be modified, the *programmer* can explicitly interface it to a NORNIR MANAGER running in a separate process. As outlined in Section 1, self-adaptive algorithms mainly work on iterative applications. For streaming applications, an iteration would correspond to the processing of an element received from the stream. To be homogeneous in the following, even when considering streaming applications, we will usually refer generically to the term *iteration*. The idea is to insert few instrumentation calls in the existing application. These calls will be invoked at each iteration of the application, to collect all the performance data needed by the manager to take reconfiguration decisions (e.g. latency and throughput). Instead of sending data to the manager at each iteration, data is stored locally, and it is sent to the manager only when the manager explicitly requests such data. All these operations are implicitly done when executing the instrumentation calls, as shown in Listing 3.

On the left, we have the original streaming application in which at each iteration of the `while` loop a stream element is received and processed. On the right, we have the same application after it has been instrumented. Here, in line 2 NORNIR opens a connection towards the MANAGER and sends to it the XML configuration file which we described in Section 4 (containing, among the others, *user*'s requirements). The processing of each stream element is wrapped between 2 calls (lines 5 and 7). In line 9, the connection with the NORNIR MANAGER is closed. Timestamps collected during the `r.begin()` and

```
StreamElement* s;
while(s = receive())
    process(s);
```

```
using nornir::Instrumenter;
Instrumenter r("parameters.xml");
StreamElement* s;
while(s = receive()){
    r.begin();
    process(s);
    r.end();
}
r.terminate();
```

Listing 3: Example of streaming application instrumentation.

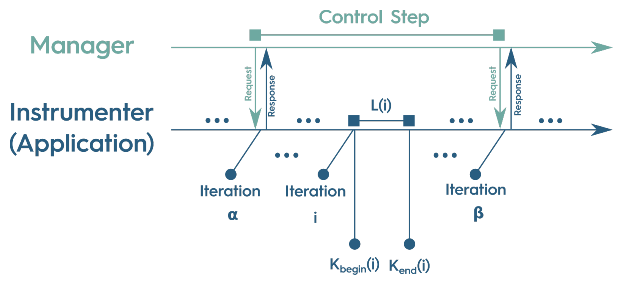


Figure 3: Relationship between α, β and arrival of stream elements.

`r.end()` calls are stored and used by the `Instrumenter` to derive performance metrics.

When the Manager needs to collect monitoring data about the application (i.e. once per control step), it will send a request to the `Instrumenter`. The `r.begin()` call verifies if there is any pending request and, if this is the case, the monitored data is sent to the Manager.

To understand how throughput, latency and utilization factor are derived from the timestamps, let us focus on a single control step. We define with $K_{begin}(i)$ and $K_{end}(i)$ the timestamps associated to `r.begin()` and `r.end()` calls performed for the $i-th$ iteration. Moreover, let's suppose that at iteration α the `Instrumenter` finds a pending monitoring request and that no other monitoring requests are received until iteration β. The distance between α and β depends on the length of the control step used by the Manager. We define with $s = \beta - \alpha$ the number of elements processed by the application during a control step. The relationship between these quantities is clarified in Figure 3.

Nornir derives the metrics in Table 1 in the following way:

Throughput is computed as

$$R = \frac{\beta - \alpha}{K_{begin}(\beta) - K_{begin}(\alpha)}$$

i.e. the number of iterations performed by the application between two successive monitoring requests, divided by the time elapsed between such requests.

Latency The latency monitored for the $i-th$ element is

$$L(i) = K_{end}(i) - K_{begin}(i)$$

To get the average latency per element, we need to sum all the latencies computed between two requests and to divide it by the number of received elements, i.e.

$$L = \frac{\sum_{i=\alpha}^{i\leq\beta} K_{end}(i) - K_{begin}(i)}{\beta - \alpha}$$

Utilization Factor The time between the start of processing of two successive elements is $K_{begin}(i+1) - K_{begin}(i)$. This includes both the time to process element i (i.e. the latency) and the time spent waiting for element $i+1$ to appear on the input stream[3]. The ratio between the latency and the time elapsed between the start of two successive elements is

$$\rho(i) = \frac{K_{end}(i) - K_{begin}(i)}{K_{begin}(i+1) - K_{begin}(i)}$$

and corresponds to the time spent doing useful work when processing element i. To get the average utilization, it is sufficient to average these quantities over control step, i.e.

$$\rho = \frac{\sum_{i=\alpha}^{i<\beta} \rho(i)}{\beta - \alpha}$$

Using this formulation, we have that $0 \leq \rho \leq 1$ for streaming applications while $\rho = 1$ in batch application (since we have $K_{begin}(i+1) - K_{end}(i) = 0$, i.e. we never wait for arrival of new data).

The advantage of this approach is that despite requiring the insertion of only 4 instrumentation calls in the already existing application, it provides Nornir all the application-level performance metrics. Moreover, differently from the *Black-Box* case, it is now possible for the user to express requirements on all the metrics presented in Table 1.

To use `begin()` and `end()` in multiple threads at the same time, it is sufficient to specify the identifier of the calling thread. In Listing 4 we show how they can be used, for example, to instrument a parallel OpenMP application.

```
Instrumenter r("parameters.xml");
StreamElement* s;
while(s = receive()){
    #pragma omp task {
        int threadId = omp_get_thread_num();
        r.begin(threadId);
        process(s);
        r.end(threadId);
    }
}
r.terminate();
```

Listing 4: Example of OpenMP application instrumentation.

It is worth to highlight that instrumentation can be performed on any C or C++ based applications (e.g. implemented by using OpenMP, TBB or C++ Threads).

[3]Actually it also includes the time required to perform the actual reading of the element once it appears on the stream. To be more accurate, we should remove this quantity from the computation of the utilization factor. However, this would require application-specific modifications, leading to a more intrusive approach. We prefer to have a slightly approximated factor to keep this solution as less intrusive as possible.

Furthermore, besides predefined performance metrics, it is also possible to store custom values (e.g. application specific metrics). Such values will be eventually provided to the self-adaptive algorithm, which can use them together with standard performance metrics to perform its decisions (Section 6).

The framework has been designed to be as lightweight as possible. In particular, the `begin()` and `end()` calls only store the timestamps, while the actual communication of the data to the manager is done by a separate *support* thread. Locks are not used, neither between the threads calling `begin()` and `end()` calls nor by the *support* thread. The only synchronizations used, involve an atomic flag shared between each application thread and the support thread. Despite this, for applications characterized by a low latency, the insertion of these two instrumentation calls may have an impact on application performance. Since the overhead is directly related to how often the `begin()` and `end()` functions are called, by calling them less often (every m iterations instead of every iteration), the overhead will be reduced. Accordingly, it is always possible to find the right sampling interval m which will lead to an overhead below 1%. However, choosing the right value of m could be critical for application programmers. For this reason, Nornir uses an *adaptive sampling* mechanism, which automatically selects the appropriate sampling length m to have a controlled overhead below 1%.

Lastly, let us consider the case where the application is not receiving any data to process (i.e. it is stuck on line 3 of Listing 4, waiting for new data to arrive). In this case (which can only occur for streaming applications), the manager could be stuck waiting for an answer from the application. To avoid this issue, it is possible to specify a maximum timeout value, after which the application is considered to have (temporarily) a throughput of zero iterations per second.

Despite being an integrating part of Nornir, the instrumentation part has been implemented as a separate library, which we called Riff[4]. We made this choice since Riff could also be useful as a standalone library to be used for performance monitoring of applications. We did not rely on existing monitoring tools [24, 33] since they do not provide the possibility to monitor utilization factor for streaming applications, to communicate custom monitoring values or to select the appropriate sampling rate automatically.

5.3. Interaction with Supported Runtimes

As described in Section 5.2, any iterative parallel application can be interfaced to Nornir by using instrumentation. However, in some cases it may be useful to have a more intrusive interaction, to access some additional reconfiguration mechanisms which may be provided by the framework. For example, some frameworks provide the possibility to dynamically change the number of threads used by the runtime (i.e. *concurrency throttling* [13]). To access these knobs, we need to perform an explicit interaction with the runtime of the application. At the time being, this is possible only for applications implemented with the FastFlow framework [34]. FastFlow is a C++ pattern based parallel programming framework, targeting multicore applications. To interface an existing FastFlow application with the Nornir Manager, is sufficient to provide the Manager (which will run in a separate thread) an handler to the FastFlow parallel application. To provide this support for other frameworks, custom monitoring and execution phases needs to be implemented, as we will describe in Section 7.1 and Section 7.2 respectively.

In general, a direct interaction with the runtime is not strictly necessary. Indeed, all the results we will show in Section 9 have been collected by performing experiments on legacy applications, implemented with Pthreads or OpenMP, by just instrumenting them with the technique we showed in Section 5.2.

5.4. Applications Implemented from Scratch

The *programmer* can write a parallel application by using the parallel programming interface provided by Nornir. This interface allows the *programmer* to write both structured (i.e. parallel patterns based) and unstructured applications expressed as a graph of concurrent activities. By doing so, Nornir can access many internal features of the runtime, thus extending its monitoring capabilities and being able to operate on additional executors. We provide both a *threads-based* interface (similar to the FastFlow one) and a *tasks-based* programming interface (details can be found in [35]).

6. The Strategy Designer

Self-adaptive strategy designers can add their new algorithms to the framework by exploiting the monitoring infrastructure and actuators already provided by Nornir.

Listing 5 shows a simplified version of the main parts of the manager implementation (the actual implementation consists of approximately 18000 lines of code). The meaning of this code snippet will become clearer with the next section. For the moment being, we can focus on the MAPE loop implemented in lines 10-24 of Listing 5.

To define a custom self-adaptive strategy, the *designer* must define a subclass of the Selector class (Listing 5, lines 35-38) and implement the `getNextKnobsValues` function. In its own Selector the *designer* can access different information provided by the superclass, like parameters specified by the *user*, the current configuration of the application and statistics about the previous monitored samples. The type of monitored data available depends on how the manager has been attached to the application. For example, if *black-box* interaction was used, the only available information is the throughput (expressed as IPS). Monitored data is kept consistent and updated by Nornir and should be exploited by the algorithm *designer* to select the values to be applied to each knob (i.e. the configuration) at the successive control step. Table 2 reports the knobs currently implemented.

The output of the function represents the value that each knob must assume at the end of the control step. Once the decision is made, the next values of each knob must be stored into a KnobsValues object, an array of values (one for each knob) which

[4]`https://github.com/DanieleDeSensi/riff`

```
1  typedef enum{
2      KNOB_VIRTUAL_CORES = 0,
3      ...
4      KNOB_NUM
5  }KnobType;
6
7  class Manager{
8      ...
9      void run(){
10         while(isRunning()){
11             sleep(samplingInterval);
12
13             // Monitor
14             ApplicationSample s = getSample();
15             storeSample(s);
16
17             // Analyze & Plan
18             KnobsValues k =
                ↪   _selector->getNextKnobsValues();
19
20             // Execute
21             for(uint i = 0; i < KNOB_NUM; i++){
22                 _knobs[i]->changeValue(k[i]);
23             }
24         }
25     }
26     virtual ApplicationSample getSample()=0;
27 };
28
29 class Knob{
30     ...
31     std::vector<double> _knobValues;
32     virtual void changeValue(double v)=0;
33 };
34
35 class Selector{
36     ...
37     virtual KnobsValues getNextKnobsValues()=0;
38 }
```

Listing 5: Simplified version of the main parts of Nornir implementation.

can be accessed by using the enumeration values identifying the type of the knob (Listing 5, lines 1-5). The returned object will then be used to set the appropriate values on the available knobs (Listing 5, lines 9-11).

For example, Listing 6 shows how to implement a simple selector that, when the monitored latency is lower than 100 ms, will force the application to run on the 25% of the available cores, setting them to work at 50% of their maximum clock frequency. When the latency is higher (or equal) than 100 ms, it will run the application on the 80% of the available cores and will set them to work at 100% of their maximum frequency.

Nornir will then automatically translate the percentage values for number of cores and frequencies in real values, according to the availability of resources on the target architecture. This greatly simplify the process for the designers, since they do not have to deal with many different tools for each considered knob. Moreover, there is no need to change the algorithm code when running the self-adaptive strategy on a different computing system with a different number of cores or a different number of frequency steps. This is possible since Nornir relies on the Mammut library to perform the actual interaction with the

Knob	Description
Number of Cores	Turns off (or on) some cores. If possible (e.g. for FastFlow applications), it will also change the number of threads used by the application (without stopping or restarting it), to have one thread on each active core. Otherwise, threads which were running on the shutdown cores will be moved to the active cores, thus leading to a situation where more threads will contend for the same core. Threads will be allocated to cores through the *Threads Mapping* knob, while this knob only enforces the specified number of cores to be active.
Hyperthreading Level	Number of hardware threads contexts to use on each physical core
Threads Mapping	Once the number of cores to use has been decided, this knob can be used to apply a given placement. For example, to place them on a set of cores sharing some resources (e.g. last level caches) for minimizing power consumption, or to place them on a set of cores with the minimum amount of shared resources, wasting more power but improving performance.
Clock Frequency	Operates on the clock frequency (and voltage) of the cores, allowing to trade a decreased performance for a lower power consumption.
Concurrency Control	For applications using FastFlow as runtime support, this knob operates on the algorithm to be used when two threads access their shared message queue [36].

Table 2: Knobs currently implemented in Nornir

```
1  class SelectorDummy: public Selector{
2    ...
3    KnobsValues getNextKnobsValues(){
4      KnobsValues k(KNOB_VALUE_RELATIVE);
5        if(_samples->average().latency < 100){
6          k[KNOB_VIRTUAL_CORES] = 25;
7          k[KNOB_FREQUENCY] = 50;
8        }else{
9          k[KNOB_VIRTUAL_CORES] = 80;
10         k[KNOB_FREQUENCY] = 100;
11       }
12       return k;
13   }
14 };
```

Listing 6: Example of Selector implementation.

hardware components, thus exploiting its portability.

Alternatively, it is possible to directly express absolute values for the knobs. By replacing `KNOB_VALUE_RELATIVE` with `KNOB_VALUE_REAL` in line 4 of Listing 6, Nornir will interpret lines 6 and 7 as "*Run the application on 25 cores and set their frequency to 50Hz*". The `_samples` variable contains the monitored data and the designer can use this object to retrieve the average value of each metric, the last monitored value, the minimum and the maximum, etc...

6.1. Full Example

To appreciate the programmability of Nornir, we show in Listing 7 the code required to implement the self-adaptive algorithm described in [37], which we denoted as *Heuristic* in this paper. This algorithm can be used to enforce a throughput higher than a given threshold on an application. Since more configurations may have such throughput, the algorithm will select the configuration with the lowest power consumption among them.

The algorithm predicts the throughput of any configuration starting from the throughput of the current configuration, supposing that it scales linearly with both the frequency and the number of cores. Accordingly, if $R(n, f)$ is the throughput of a configuration using n cores with a clock frequency f, we have:

$$R(n, f) = R(\bar{n}, \bar{f}) \cdot \frac{n \cdot f}{\bar{n} \cdot \bar{f}}$$

where $\bar{n}$ is the number of cores currently used and $\bar{f}$ is the current clock frequency. In lines 4-5 we retrieve the n and f for the configuration `pConf` we want to predict, while in lines 6-7 we get the current number of cores, the current clock frequency and the current throughput. Eventually, in lines 9-11 we perform the prediction.

Heuristic predicts the power consumption $P(n, f)$, as:

$$P(n, f) = n \cdot f \cdot v^2$$

with v being the voltage associated to a specific number of cores and clock frequency. The voltage depends only on these two factors and it is precomputed when installing Nornir and stored to a configuration file, which can be accessed at runtime through the `getVoltage` function. Lines 18-21 predicts the power consumption of a specific configuration `pConf` by using this model.

Then, as described in Section 6 we need to extend the `Selector` class and define its `getNextKnobsValues` function (lines 44-51). In lines 45-48 we check if the monitored throughput is higher or equal to the one required. The monitoring data can be accessed through the `_samples` variable provided by the `Selector` class[5]. If this is the case, the configuration we are using satisfies the requirements expressed by the user and we simply return it. Otherwise, the algorithm searches a better configuration (line 49), by calling the `getBestConfiguration` function. This function scans all the configurations (line 34) and, for each of them, checks if the throughput is higher than that required by the user, and if the power consumption is lower then the lowest found up to that moment. If this is the case, both the best configuration and its power consumption are updated (lines 35-39). Eventually, the best configuration found is returned (line 40). As discussed in Section 6, this configuration will eventually be enforced by Nornir by using the appropriate actuators.

Although being a simple example, it clearly shows the advantages using Nornir. Indeed, we only focused on the algorithm, without dealing with complex issues related to monitoring the application and interfacing with the underlying hardware, which is automatically managed by Nornir.

[5]For brevity's sake, we did not show the constructors of the different classes. Constructor of `SelectorAnalytical` class simply creates the two predictors objects, by providing them the `_configuration` and `_samples` variables, which it gets from the `Selector` base class.

```
1  class PredictorHeuristicThroughput {
2  public:
3    double predict(const KnobsValues& pConf){
4       double cores = pConf[KNOB_VIRTUAL_CORES];
5       double freq = pConf[KNOB_FREQUENCY];
6       double currCores =
          ↪  _configuration.getRealValue(KNOB_VIRTUAL_CORES);
7       double currFrequency =
          ↪  _configuration.getRealValue(KNOB_FREQUENCY);
8       double currThr = _samples->average().throughput;
9       double scalingFactor = (cores * freq) /
10                             (currCores * currFrequency);
11      return currThr * scalingFactor;
12   }
13 };
14
15 class PredictorHeuristicPower {
16 public:
17   double predict(const KnobsValues& pConf){
18     double cores = pConf[KNOB_VIRTUAL_CORES];
19     double freq = pConf[KNOB_FREQUENCY];
20     double voltage = getVoltage(cores, freq);
21     return (cores*voltage*voltage*freq);
22   }
23 };
24
25 class SelectorAnalytical: public Selector {
26 private:
27   PredictorHeuristicThroughput* _pThr;
28   PredictorHeuristicPower* _pPow;
29 public:
30   KnobsValues getBestConfiguration(){
31     KnobsValues bestConf =
         ↪  _configuration.getRealValues();
32     double bestConfPower =
         ↪  numeric_limits<double>::max();
33
34     for(const KnobsValues& pConf :
         ↪  _configuration.getAllRealCombinations()){
35       if(_pThr->predict(pConf) >
           ↪  _p.requirements.throughput &&
36          _pPow->predict(pConf) < lowestPower){
37         bestConf = pConf;
38         bestConfPower = _pPow->predict(pConf);
39       }
40     }
41     return bestConf;
42   }
43
44   KnobsValues getNextKnobsValues(){
45     if(_samples->average().throughput >=
46        _p.requirements.throughput){
47       return _configuration.getRealValues();
48     } else {
49       return _configuration.getBestConfiguration();
50     }
51   }
52 };
```

Listing 7: Implementation of the algorithm presented in [37] (denoted as *Heuristic* in this paper.

7. Extending Nornir

The strategy designer or runtime support developer may decide to extend Nornir to support other parallel programming frameworks or to use other control knobs beside those already provided.

7.1. Monitor

Extending the monitoring infrastructure may be useful for two different reasons. On one side, the monitoring part can be extended to gather additional metrics (for example in the *black-box* case to retrieve metrics different than IPS). On the other side, it can be used to interface Nornir with other parallel programming frameworks.

In both cases, a new manager class must be defined, by subclassing the Manager class and implementing the `getSample` function (Listing 5, line 26). For example, consider the case where we want to interface Nornir with the Intel TBB runtime. In this case, the `getSample` function should implement the logic to interact with the runtime and to collect performance metrics (e.g. number of tasks executed per second). If needed, the user can store additional framework specific values (e.g. number of tasks stolen by each thread) inside some member variables of the `ApplicationSample` class.

At each control step, Nornir calls this function (Listing 5, line 14) and the sample will be stored (Listing 5, line 15) to be accessible from the *Analyze* and *Plan* phases, as we saw in Section 6. By doing so, in the strategy the *designer* could, for example, take decisions according to framework-specific metrics.

7.2. Execute

To implement a new executor, the *designer* must define a subclass of the Knob class (Listing 5, lines 29-35). In the constructor, the `_knobValues` vector must be populated with the set of values that the knob can assume. When the planning phase terminates, the function `changeValue` will be called by the manager on all the available knobs (Listing 5, lines 21-23). The parameter `k[i]` corresponds to the value that the specific knob must assume according to the planning algorithm. By implementing the function `changeValue`, the *designer* specifies the actual code to be executed when the *Plan* phase decides to change the value of that knob. The new Knob object must then be created and added to the `_knobs` array (used in Listing 5, line 22). Moreover, a new enumeration value must be assigned to this knob (Listing 5, lines 1-5). As anticipated, many available knobs have been implemented by using Mammut, which could also be used to implement new actuators. New knobs can be added to support framework-specific actuators. For example, let us suppose the designer wants to add the possibility to dynamically change the number of threads in OpenMP. In this case, a new Knob should be added, and the designer should add in the `changeValue` function the code to change the number of threads used by OpenMP (for example by setting the `OMP_NUM_THREADS` environment variable).

8. Building the Testbed

As discussed in Section 5.2, Nornir can be used for controlling already existing applications by instrumenting them. In the following we will briefly discuss how it can be used to instrument the applications of the *Princeton Application Repository for Shared-Memory Computers* (PARSEC) benchmark [9]. Apart from being a useful example of usage of the framework, the contribution of this part is the release of a "Nornir-ready" version of the benchmark suite. The modified PARSEC applications have been added to the P^{3}ARSEC (Parallel Patterns PARSEC) benchmark suite [10][6] and released as open source.

8.1. The PARSEC benchmark

The PARSEC benchmark is a well-known benchmark suite composed of 13 parallel applications, diverse in terms of application domain, programming model (pipeline, data-parallel and unstructured), granularity, working set size, data sharing and data exchange patterns [9]. Thanks to this heterogeneity, this suite is often used in the HPC community with different purposes. Therefore, providing a Nornir-ready version of the PARSEC benchmark is interesting to validate self-adaptive algorithms on a wide range of real world scenarios.

The PARSEC suite comprises different parallelization of the applications. For all the benchmarks we considered the Pthreads version, except for `freqmine` which only provides the OpenMP version. Being integrated in the original PARSEC tool chain, this Nornir-based version can be seamlessly executed by using the standard tools already provided by PARSEC. For example, to run the `blackscholes` application with specific power consumption and performance requirements, the user should create, in the PARSEC root directory, the XML configuration file containing the requirements and the self-adaptive reconfiguration algorithm to be used. Then, the application can be executed by using the standard `parsecmgmt` tool as follows:

```
./parsecmgmt -a run -p blackscholes -i native -n 24
                 -c gcc-pthreads-nornir
```

where we specified the benchmark name, the size of the input set, the number of threads to be used and the version of the benchmark to be run (`gcc-pthreads-nornir`)[7]. Basically, by doing so we can set any performance and power consumption requirement on all the applications in the PARSEC benchmark.

8.2. Instrumenting the Applications

In principle, we can control the applications without instrumenting them, by relying on hardware performance counters (e.g. *Instructions Per Second* (IPS)). In this case, performance requirements should be expressed by the user in IPS. However, as discussed in Section 5.1, correlating IPS with the actual application performance is not an easy task. We believe that is much more intuitive for the application user to express requirements in terms of *frames per second* or *queries per second* than expressing them in terms of IPS. This is the reason why we decided to instrument the different applications.

[6]The repository is publicly available at `https://github.com/ParaGroup/p3arsec`

[7]In most cases, when additional implementations for the benchmark are available (e.g. implementations with OpenMP or Intel TBB), we provide support for those versions as well.

As we anticipated in Section 3, self-adaptive reconfiguration algorithms only work on iterative computations and all PARSEC applications exhibit some kind of iterative behaviour. To allow the framework to monitor the actual performance of the applications, we modified them by inserting the required instrumentation calls (see Section 5.2). They will be invoked at each loop iteration in order to send monitored performance (e.g. number of iterations performed per time unit) to the Nornir Manager. In Table 3 we show what an *iteration* is, according to the specific instrumentation we made. Note that this reflects directly on the way in which the user expresses performance requirements, since throughput requirements will be expressed in terms of *iterations per second*. Consequently, it would be possible for example for `blackscholes` to express requirements in terms of stock options processed per second.

Benchmark	Iteration
`Blackscholes`	1 Stock Option
`Bodytrack`	1 Frame
`Canneal`	1 Move
`Dedup`	1 Chunk
`Facesim`	1 Frame
`Ferret`	1 Query
`Fluidanimate`	1 Frame
`Freqmine`	1 Call of the `FP_growth` function
`Raytrace`	1 Frame
`Streamcluster`	1 Evaluation for opening a new center
`Swaptions`	1 Simulation
`Vips`	1 Image Tile
`X264`	1 Frame

Table 3: *Iteration* meaning in PARSEC benchmarks, according to the instrumentation we performed.

Instrumenting the applications only required identifying the outermost loop and the insertion of the two instrumentation calls at the beginning and end of the loop. This thanks to the fact that all the interaction with Nornir is managed by these two calls.

9. The Designer Use-Case

One of the key point of our approach is that the strategy designer can easily exploit the Nornir framework for comparing and evaluating different reconfiguration strategies on the same testbed. To prove the effectiveness of our approach, we implemented some existing strategies by using Nornir. Then, we exploited the testbed we described in Section 8 to compare these algorithms over a wide set of real applications. Implementing these algorithms required relatively low programming effort since we were able to only focus on the actual code of the algorithm, abstracting all the interactions with the underlying hardware and with the application. Moreover, while some of these algorithms were only simulated in the original works, by using Nornir we were able to actually execute them at runtime for reconfiguring parallel applications.

In the following, we will describe the different implemented strategies and then we will discuss some results obtained by considering the different solutions.

9.1. The strategies

We implemented different reconfiguration strategies: the first two were designed by ourselves, while the others are approaches proposed in the literature. We will not enter into the details of the implemented strategies but we will provide references for all of them. They are usually characterized by a first stage where they search for a proper configuration, which will then be used to run the application for the remaining part of the execution. In general, if the application enters a different phase (e.g. it moves from an I/O intensive phase to a CPU intensive phase), a new optimal configuration needs to be found. The implemented algorithms are:

Heuristic: being n the number of cores and f the CPU frequency, when the user requirements are violated, the algorithm will search for another $< n, f >$ configuration, with acceptable performance according to the requirements. For doing that, the algorithm first predicts the throughput of the application for all the possible pairs, by assuming it to be proportional to the number of cores used and the CPU frequency. Then, if required by the user, among all the configurations with acceptable performance, the algorithm could pick the one with the lowest estimated power consumption. Instead of predicting the actual power consumption, the algorithm will just estimate whether the power consumption of a configuration is higher or lower than the power consumption delivered by another configuration. For this reason, this algorithm can only be used to enforce performance requirements. More details about this algorithm can be found in [37].

Online Learning: this algorithm adopts machine learning techniques and allows the user to express also explicit power consumption requirements. It considers two distinct stages throughout the application execution: *training* phase and *steady* phase. When the application starts, the algorithm begins the training phase, by applying the following steps: *i)* a not yet visited $< n, f >$ configuration of the application is applied; *ii)* the throughput and power consumption of the application are monitored for a short predefined period; *iii)* monitored data is used to refine power consumption and throughput prediction models; *iv)* throughput and power consumption of the current configuration are predicted by using such models and are compared with real monitored values. If the prediction error is lower than a specified threshold, the training phase finishes, otherwise, the process is iterated. Once the training phase ends, the computed models are used to select a proper configuration according to the requirements expressed by the user. More details about this algorithm can be found in [13].

LiMartinez: Is a well-known heuristic presented in [12], which uses a combination of hill climbing and binary search algorithms to find the lowest power consuming solution (in terms of number of cores n and frequency f) under a performance constraint. However, since it does not explicitly model power consumption, it is not possible to specify any power consumption requirement.

Leo: Is a recent reconfiguration algorithm presented in [8]. It uses an offline learning approach based on previous profiling of the applications. These profiling data is integrated with information collected online while the application is running. This algorithm explores a fixed number of configuration during the online phase, set to 20 by its designers. Differently from the other algorithms we presented (which only operate on n and f), this algorithm is more general and it could in principle be used on any knobs. Due to technical limitation of the algorithm, it was not possible to run it on `Facesim` and `Fluidanimate`.

Rapl: Is an hardware-enforced solution available on newer Intel's processors [38], which automatically scales the clock frequency f and which can be used to limit the maximum power consumption of the CPU over a time window. We considered a time window of 1 second, equal to the one we considered for the other algorithms. This algorithm cannot be used to enforce performance requirements.

In Table 4 we outline the main characteristics of each algorithm. All these policies are provided by Nornir and are ready to be used without any additional effort.

Strategy	**Description**	**Supported Requirements**
Heuristic	Simple heuristic assuming the performance to be proportional to the number of cores and to their clock frequency.	Performance only.
Online Learning	Online learning algorithm accurately estimating performance and power consumption of the application, refining prediction models at runtime.	Performance and Power Consumption.
LiMartinez	Heuristic using hill climbing and binary search.	Performance only.
Leo	Offline learning algorithm, refining the model with data collected online.	Performance and Power Consumption.
Rapl	Hardware-enforced solution operating on the clock frequency of the cores.	Power Consumption.

Table 4: Reconfiguration Strategies

9.2. Evaluation

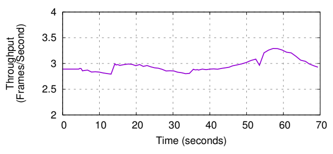
We conducted all the experiments on an Intel workstation with 2 Xeon E5-2695 @2.40GHz CPUs, each with 12 2-way hyperthreaded cores, running Linux x86_64. This machine has 13 possible frequency levels: from 1.2GHz to 2.4GHz with steps of 0.1GHz. In all the experiments we used the `native` input set provided by PARSEC.

To analyze the overhead introduced by Nornir, for each application we compared the best execution time with the execution time when the application is interfaced with Nornir. For all the applications, we measured an overhead lower than 1%.

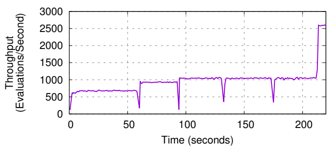
To better analyze the behavior of the different strategies, we divide the application into the following two sets, according to the regularity of their run-time behavior:

Stable This set includes `Blackscholes`, `Bodytrack`, `Canneal`, `Facesim`, `Raytrace`, `Streamcluster`, `Swaptions`, `Vips` applications. These applications are characterized by a throughput more or less stable during their execution. Fluctuations in the throughput occur but their amplitude is limited (as shown in Figure 4(a) for the `Raytrace` application) or not too frequent (as shown in Figure 4(b) for the last part of `Streamcluster` execution).

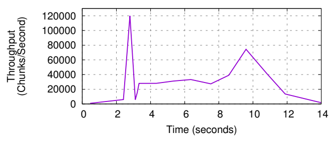
Unstable This set includes `Dedup`, `Ferret`, `Fluidanimate`, `Freqmine`, `X264` applications. These applications are characterized by large and frequent fluctuations, as shown in Figure 4(c) for the `Dedup` application.



(a) Throughput of the `Raytrace` application.



(b) Throughput of the `Streamcluster` application.



(c) Throughput of the `Dedup` application.

Figure 4: Stable and Unstable Applications

We need to perform this distinction since, as we will show later, more complex algorithms may have difficulties in managing *unstable* applications, while they will work better than other solutions when applied on *stable* applications.

In our evaluation we consider *throughput* and *power consumption* requirements. In the former case, the self-adaptive algorithm should find the configuration with the lowest power consumption among the configurations with a throughput higher than the specified one. In the latter, the self-adaptive algorithm should find the configuration with the highest throughput among the configurations with a power consumption lower than the specified one.

To avoid biases in the results due to specific requirements choices, we test the algorithms over a wide range of different requirements. For example, if we need to specify a power consumption requirement for an application that has a minimum power consumption of 20 Watts and a maximum of 220 Watts, we will slice the range in 10 equal interval, i.e. we will test the algorithms by setting the power consumption requirement to 40, 60, ... and 200 Watts. The algorithm will be executed 5 times for each requirement and for each application. We will show the average and the *pooled standard deviation* obtained over all these requirements. The pooled standard deviation is a weighted average of each group's standard deviation. We considered pooled standard deviation since it can estimate variance of several different groups when the mean of each groups may be different, but the variance of each groups is very similar. By doing so, we were able to aggregate the standard deviation among the different experiments done for all the benchmarks. For all the algorithms we set a control step of 1 second. To speed up the search of a proper configuration, we set the control step during the training phase (or search phase for the heuristics) equal to 100 milliseconds. On our online learning algorithm, the training phase will terminate when both the performance and power consumption prediction errors are lower than 10%.

When evaluating the reconfiguration algorithms, we will consider the metrics reported in Table 5.

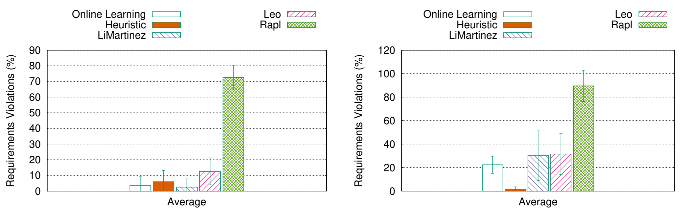
Metric	Description
Requirements Violations	The percentage of test cases for which the algorithm failed in finding a configuration with the required throughput or power consumption
Training Steps	The number of configurations visited by the algorithm during the training phase (or during the search phase for heuristics). If multiple training stages are executed (due to phase changes), this metric will consider the sum of all the training stages.
Training Time	The time spent in the training phase (or during the search phase for heuristics). Having a short training time is a desirable property since during the training no guarantees on performance and power consumption are provided. It is worth noting that the *training time* is not simply the number of training steps multiplied by the length of the control step. Indeed, when collecting monitoring data from the application, we need to collect data for at least one iteration. Accordingly, the *actual* length of the control step depends on the length of one application iteration.
Suboptimality	Even when a proper configuration is found, due to prediction inaccuracies it may be not the optimal one. With *suboptimality* we consider the percentage difference between the true optimal configuration and the configuration selected by the algorithm. As optimal configuration, we consider the one found by an ideal oracle, which knows *exactly* the power consumption and performance of all the possible configurations.

Table 5: Metrics used for evaluating the strategies

Intuitively, for all these metrics the lower is the value, the better is the algorithm.

9.2.1. *Requirement violations*

We want to evaluate the percentage of tests for which the different algorithms failed in finding a configuration satisfying the user requirements regarding performance or power consumption. Figure 5 shows these results averaged over all the PARSEC applications. We show the average between power consumption and performance requirements since there was no significant difference between the two. For algorithms supporting only one requirement (e.g. *Rapl* which supports power consumption requirements only), we show only the result for the supported requirement. A similar approach will be adopted for the next metrics.



(a) Stable Applications (b) Unstable Applications

Figure 5: Requirements violations of the different algorithms.

The first thing we notice is the high number of requirements that Rapl is not able to enforce. Indeed, since it only operates on the clock frequency, it is not able to decrease too much the power consumption of an application, thus violating low power consumption requirements. This is not an issue for the other techniques, since they also operate on the number of cores, thus being able to explore a broader range of possibilities.

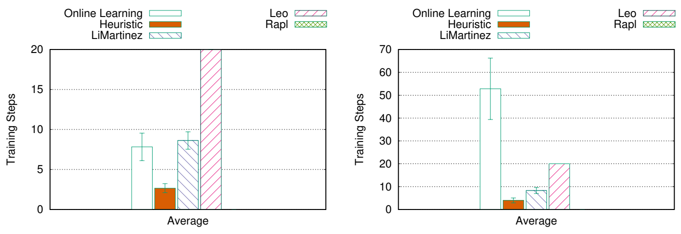
For the *stable* case (Figure 5(a)), Online Learning, Heuristic and LiMartinez algorithms violate the requirements in less than 10% of the cases. This percentage is slightly higher for Leo due to mispredictions.

Concerning the *unstable* applications (Figure 5(b)), algorithms that exploit some form of training significantly suffer from the instability of the application. For example, in the Online Learning algorithm, the configuration found by the algorithm could not be correct anymore, and could now violate the requirements. Despite a new training phase is performed, in some cases the algorithm could continuously chase the optimal configuration but without actually finding it since it keeps changing. This is not the case for Heuristic algorithm which, due to a simpler search phase, reacts better to phase changes, thus being able to satisfy user requirements even in presence of workload fluctuations.

9.2.2. *Training steps and training time*

Considering the number of cores and CPU frequency level, in the used workstation we have 312 different operating configurations. Figure 6 shows the average number of configurations visited by each algorithm before they find a proper configuration, averaged over all the requirements. For Leo, we only consider the number of configurations explored in the online training part, that, as suggested by the designer in [8], is set to 20. For Rapl algorithm we always consider both training steps and training time to be zero since, when it can enforce the specified power cap, it will usually do so in few milliseconds.

In *stable* applications, the Online Learning algorithm can usually find a proper configuration after visiting less than 10 configurations, while the Heuristic is faster and can converge after visiting less than 5 configurations. The LiMartinez algorithm usually needs few more steps than the Online Learning



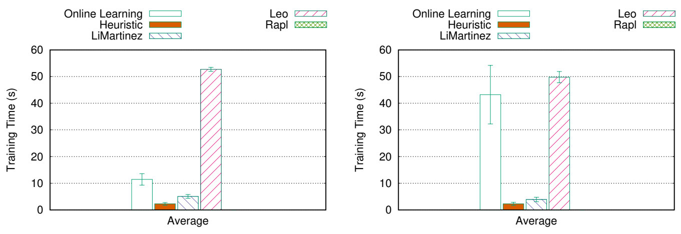
(a) Stable Applications (b) Unstable Applications

Figure 6: Training steps of the different algorithms.

algorithm before finding a suitable configuration.

For *unstable* applications, the situation is quite different. The Online Learning algorithm does not perform well, due to re-trainings, which increase the number of configurations explored. On the other hand, both the Heuristic and the LiMartinez algorithms can usually converge in less than 10 steps, since they do not require sophisticated training to be executed for every different phase.

Figure 7 reports the *training time*, i.e. the absolute time spent by the different algorithms in the training phase (see Table 5).



(a) Stable Applications (b) Unstable Applications

Figure 7: Training time (seconds) of the different algorithms.

For *stable* applications the Online Learning algorithm can quickly find a proper configuration, usually faster than the Leo algorithm but not faster than the Heuristic and LiMartinez heuristic algorithms. For *unstable* applications, Online Learning spends too much time performing re-trainings, while LiMartinez and Heuristic algorithms are not affected by fluctuations in the workload, thanks to the simplicity of their heuristic prediction algorithms. Despite Leo has a lower number of tested configurations with respect to the Online Learning algorithm, it spends more time in the training phase with respect to the other solutions. This happens since Leo explores many *slow* configurations, increasing the time spent in the training stage.

For completeness, in Table 6 we report the training time for each benchmark, expressed as a percentage over the execution time of the application. Training time for Rapl is not reported since it is always close to zero. In the first half of the table we report the data for *stable* applications, while in the second part we report results for *unstable* benchmarks. The results are coherent with those we showed in Figure 7. It is worth to remark that in these experiments we considered a worst case scenario, where the execution time of the application is typically lower than 1 minute. In such cases, even if the training only takes few

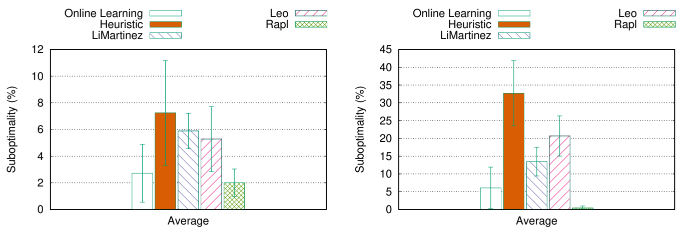
Benchmark	Online Learning	Heuristic	LiMartinez	Leo
`Blackscholes`	13.65	2.29	14.41	81.82
`Bodytrack`	15.23	6.44	10.16	61.76
`Canneal`	15.66	7.18	11.59	78.12
`Facesim`	17.82	6.71	13.12	N.A.
`Raytrace`	27.42	9.44	17.81	84.07
`Streamcluster`	11.02	2.33	5.63	40.62
`Swaptions`	7.46	1.22	6.81	63.37
`Vips`	20.91	3.59	25.58	90.43
`Dedup`	68.96	11.97	23.06	84.59
`Ferret`	47.82	4.99	9.14	58.68
`Fluidanimate`	51.29	6.82	3.99	N.A.
`Freqmine`	59.27	1.14	10.98	50.17
`X264`	71.28	10.58	17.92	80.00

Table 6: Average percentage of the execution time spent in training the algorithm. Rapl is not reported since it is almost 0 in all the cases. The top part of the table reports the *stable* benchmarks while the *unstable* benchmarks are reported in the bottom part of the table. Due to thecnical limitation of the algorithm, it was not possible to run Leo on `Facesim` and `Fluidanimate`.

seconds, in percentage, this looks like a significant impact. In practice, such kind of self-adaptive techniques are mostly suited for long running applications, for which the impact of the training would be much lower.

9.2.3. Suboptimality

Figure 8 shows the suboptimality of the configurations selected by the different algorithms, i.e., how far is the selected configuration from the optimal one.



(a) Stable Applications (b) Unstable Applications

Figure 8: Suboptimality of the different algorithms

For *stable* applications all the considered algorithms can find configurations characterized by a power consumption (for the performance requirement) or performance (for the power consumption requirement) within 5% from the optimal one. Differences become more evident in the case of *unstable* applications. The Rapl algorithm has the best performance here. However, as anticipated it can only provide guarantees on power consumption. The Online Learning algorithm is slightly worst than Rapl but can satisfy requirements on both performance and power consumption. Due to its simplicity, the Heuristic approach cannot find configurations as good as those found by Online Learning algorithm.

9.2.4. Summary

In Figure 9 we report a summary comparison through a radar chart, where we normalized the results we already showed with values between 0 and 1. On each radius, the highest the value, the better the algorithm is. Accordingly, if we consider for example the *Training Time* radius, a higher value does not mean that the algorithm has a *higher* training time but a *better* (i.e., shorter) training time. We added a *Supported Requirements* radius, to also consider the types of requirements supported by each algorithm. We assigned a value of 0.5 to algorithms supporting only power consumption or performance requirements and a value of 1.0 to algorithms supporting both performance and power consumption requirements.

Stable Applications. For stable applications, Online Learning, Heuristic and LiMartinez violate a comparable number of requirements. Rapl performs poorly since, by acting only on voltage and clock frequency, cannot decrease the power consumption too much. Regarding optimality, configurations found by Online Learning algorithm are very close to those found by an efficient and hardware-enforced solution such as Rapl. On the other hand, Online Learning and Leo supports both performance and power consumption requirements, while Heuristic and LiMartinez support only performance requirements and Rapl only supports power consumption requirements. Concerning the training, Online Learning can find a proper configuration in a time comparable to Heuristic, LiMartinez and Rapl.

Unstable Applications. For unstable applications the performance of the Online Learning and Leo algorithms decreases in terms of training time, training steps and requirements violations. However, our Heuristic algorithm still performs well and, concerning requirements violations, better than the other heuristic (LiMartinez) we considered in our comparison.

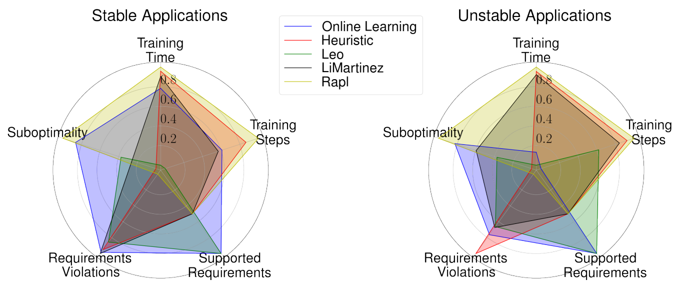


Figure 9: Summary comparison between the different algorithms.

To conclude, there are scenarios (e.g. *unstable applications*) where there is not a clear winner and the best algorithm to be used may depend on the user wishes. This highlights the importance of having a framework that allows an algorithm designer to easily create new algorithms and to compare them with existing strategies.

10. Conclusions and Future Work

Being able to explicitly control performance and power consumption of parallel application is a growing requirement in different scenarios. In this paper, we presented the design and implementation of Nornir, a framework which can be used to implement self-adaptive algorithms or to apply self-adaptive reconfiguration decisions on parallel applications.

By using Nornir, the algorithm designer can just focus on the algorithm itself, neglecting all the low-level issues related to interaction with the application and with the underlying computing node. We provide different solutions for interfacing Nornir to an existing application, according to the effort the programmer is willing to put in doing this task. The more is the programming effort, the more are the information collected by Nornir and the more are the reconfiguration mechanisms that the algorithms could be able to exploit. To provide a meaningful and varied testbed for the self-adaptive algorithms we instrumented the applications in the PARSEC benchmark, by creating a self-adaptive version of PARSEC, on which is possible to set performance and power consumption requirements for each application.

Eventually, we used our framework to implement different self-adaptive algorithms, some of which were originally only simulated. By doing so we achieved two goals. On one side, we proved that Nornir is flexible enough to implement a wide set of different algorithms and that is possible to do that by just implementing the algorithm logic, without having to explicitly interact with the application or with the underlying hardware. On the other side, we compared such algorithms, showing strength and weaknesses of each of them.

In the future, we plan to extend Nornir to support applications running on heterogeneous platforms, for example by dynamically adapting GPU resources. This would require adding new knobs to control resources on the GPU (e.g. block size, etc...), as well as new self-adaptive algorithms to select the appropriate value for each resource. New knobs can be added with relatively low effort, thanks to the modular design of Nornir, while the self-adaptation strategy can be implemented either by relying on state of the art solutions or by designing new strategies from scratch.

In addition, we would like to extend our work for applications running on distributed memory environment, either by providing support to MPI-based applications or by providing a more general solution, similarly to what we did with the instrumentation of general parallel applications.

Acknowledgements

This work has been partially supported by the EU H2020-ICT-2014-1 project RePhrase (No. 644235).

References

[1] Y. Xiao, M. Krunz, Qoe and power efficiency tradeoff for fog computing networks with fog node cooperation, in: IEEE INFOCOM 2017 - IEEE Conference on Computer Communications, 2017, pp. 1–9. `doi:10.1109/INFOCOM.2017.8057196`.

[2] S. Sudevalayam, P. Kulkarni, Energy harvesting sensor nodes: Survey and implications, IEEE Communications Surveys Tutorials 13 (3) (2011) 443–461. `doi:10.1109/SURV.2011.060710.00094`.

[3] S. Peng, T. Wang, C. Low, Energy neutral clustering for energy harvesting wireless sensors networks, Ad Hoc Networks 28 (Supplement C) (2015) 1 – 16. doi:https://doi.org/10.1016/j.adhoc.2015.01.004.
URL http://www.sciencedirect.com/science/article/pii/S1570870515000062

[4] B. Gedik, S. Schneider, M. Hirzel, K. L. Wu, Elastic scaling for data stream processing, IEEE Transactions on Parallel and Distributed Systems 25 (6) (2014) 1447–1463. doi:10.1109/TPDS.2013.295.

[5] T. De Matteis, G. Mencagli, Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing, in: Proc. of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2016, pp. 13:1–13:12.

[6] B. Lohrmann, P. Janacik, O. Kao, Elastic stream processing with latency guarantees, in: The 35th Intl. Conf. on Distributed Computing Systems, 2015.

[7] A. Gandhi, M. Harchol-Balter, R. Das, J. Kephart, C. Lefurgy, Power Capping Via Forced Idleness, in: Proc. of Workshop on Energy-Efficient Design, 2009.

[8] N. Mishra, H. Zhang, J. D. Lafferty, H. Hoffmann, A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints, ACM SIGARCH Computer Architecture News 43 (1) (2015) 267–281.

[9] C. Bienia, S. Kumar, J. P. Singh, K. Li, The parsec benchmark suite: Characterization and architectural implications, in: 17th Inter. Conf. on Parallel Architectures and Compilation Techniques, PACT '08, ACM, 2008, pp. 72–81. doi:10.1145/1454115.1454128.

[10] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, M. Danelutto, Bringing parallel patterns out of the corner: The p^3arsec benchmark suite, ACM Trans. Archit. Code Optim. 14 (4) (2017) 33:1–33:26. doi:10.1145/3132710.
URL http://doi.acm.org/10.1145/3132710

[11] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50. doi:10.1109/MC.2003.1160055.

[12] J. Li, J. F. Martínez, Dynamic power-performance adaptation of parallel computation on chip multiprocessors, Proceedings - International Symposium on High-Performance Computer Architecture 2006 (2006) 77–87.

[13] D. De Sensi, M. Torquati, M. Danelutto, A reconfiguration algorithm for power-aware parallel applications, ACM Trans. Archit. Code Optim. 13 (4) (2016) 43:1–43:25.

[14] R. Zhang, C. Lu, T. F. Abdelzaher, J. A. Stankovic, Controlware: a middleware architecture for feedback control of software performance, in: Proceedings 22nd International Conference on Distributed Computing Systems, 2002, pp. 301–310. doi:10.1109/ICDCS.2002.1022267.

[15] A. Goel, D. Steere, C. Pu, J. Walpole, Swift: A feedback control and dynamic reconfiguration toolkit, Tech. rep. (1998).

[16] B. Li, K. Nahrstedt, A control-based middleware framework for quality-of-service adaptations, IEEE Journal on Selected Areas in Communications 17 (9) 1632–1650. doi:10.1109/49.790486.

[17] J. Panerati, F. Sironi, M. Carminati, M. Maggio, G. Beltrame, P. J. Gmytrasiewicz, D. Sciuto, M. D. Santambrogio, On self-adaptive resource allocation through reinforcement learning, in: 2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013), 2013, pp. 23–30. doi:10.1109/AHS.2013.6604222.

[18] H. Hoffman, Seec: A framework for self-aware management of goals and constraints in computing systems, Ph.D. thesis, Cambridge, MA, USA (2013).

[19] C. Imes, H. Hoffmann, Bard: A unified framework for managing soft timing and power constraints, in: 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016, pp. 31–38. doi:10.1109/SAMOS.2016.7818328.

[20] C. Imes, D. H. K. Kim, M. Maggio, H. Hoffmann, Portable multicore resource management for applications with performance constraints, in: 2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC), 2016, pp. 305–312. doi:10.1109/MCSoC.2016.10.

[21] IBM, Ibm stream s, https://www.ibm.com/ms-en/marketplace/stream-computing (2018).

[22] B. Gedik, S. Schneider, M. Hirzel, K. L. Wu, Elastic scaling for data stream processing, IEEE Transactions on Parallel and Distributed Systems 25 (6) (2014) 1447–1463. doi:10.1109/TPDS.2013.295.

[23] D. De Sensi, M. Torquati, M. Danelutto, Mammut: High-level management of system knobs and sensors, SoftwareX 6 (2017) 150 – 154. doi:https://doi.org/10.1016/j.softx.2017.06.005.
URL http://www.sciencedirect.com/science/article/pii/S2352711017300225

[24] M. Maggio, H. Hoffmann, M. Santambrogio, A. Agarwal, A. Leva, Controlling software applications via resource allocation within the heartbeats framework, in: Decision and Control (CDC), 2010 49th IEEE Conference on, 2010, pp. 3736–3741. doi:10.1109/CDC.2010.5717893.

[25] D. Chasapis, M. Casas, M. Moretó, M. Schulz, E. Ayguadé, J. Labarta, M. Valero, Runtime-guided mitigation of manufacturing variability in power-constrained multi-socket numa nodes, in: Proceedings of the 2016 International Conference on Supercomputing, ICS '16, ACM, New York, NY, USA, 2016, pp. 5:1–5:12. doi:10.1145/2925426.2926279.
URL http://doi.acm.org/10.1145/2925426.2926279

[26] M. Casas, R. M. Badia, J. Labarta, Automatic phase detection and structure extraction of mpi applications, Int. J. High Perform. Comput. Appl. 24 (3) (2010) 335–360. doi:10.1177/1094342009360039.
URL http://dx.doi.org/10.1177/1094342009360039

[27] E. Totoni, J. Torrellas, L. V. Kale, Using an adaptive hpc runtime system to reconfigure the cache hierarchy, in: Proc. of SC 2014, IEEE Press, 2014, pp. 1047–1058.

[28] A. Sembrant, D. Black-Schaffer, E. Hagersten, Phase behavior in serial and parallel applications, in: Workload Characterization (IISWC), 2012 IEEE Intl. Symposium on, 2012, pp. 47–58.

[29] M. A. Islam, S. Ren, X. Wang, Greencolo: A novel incentive mechanism for minimizing carbon footprint in colocation data center, in: International Green Computing Conference, 2014, pp. 1–8. doi:10.1109/IGCC.2014.7039140.

[30] N. H. Tran, T. Z. Oo, S. Ren, Z. Han, E. N. Huh, C. S. Hong, Reward-to-reduce: An incentive mechanism for economic demand response of colocation datacenters, IEEE Journal on Selected Areas in Communications 34 (12) (2016) 3941–3953. doi:10.1109/JSAC.2016.2611958.

[31] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, M. Rinard, Dynamic Knobs for Responsive Power-aware Computing, SIGPLAN Not. 46 (3) (2011) 199–212.

[32] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, A. Agarwal, A generalized software framework for accurate and efficient management of performance goals, in: 2013 Proceedings of the International Conference on Embedded Software (EMSOFT), 2013, pp. 1–10. doi:10.1109/EMSOFT.2013.6658597.

[33] F. Sironi, D. B. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, M. D. Santambrogio, Metronome: Operating system level performance management via self-adaptive computing, in: Proceedings of the 49th Annual Design Automation Conference, DAC '12, ACM, New York, NY, USA, 2012, pp. 856–865. doi:10.1145/2228360.2228514.
URL http://doi.acm.org/10.1145/2228360.2228514

[34] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, M. Torquati, Accelerating code on multi-cores with fastflow, in: E. Jeannot, R. Namyst, J. Roman (Eds.), Proceedings of the 17th International Conference on Parallel Processing - Volume Part II, Vol. 6853 of Euro-Par'11, Springer-Verlag, 2011, pp. 170–181.
URL http://dl.acm.org/citation.cfm?id=2033408.2033428

[35] M. Danelutto, D. De Sensi, M. Torquati, A power-aware, self-adaptive macro data flow framework, Parallel Processing Letters 27 (01) (2017) 1740004. arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0129626417400047, doi:10.1142/S0129626417400047.
URL http://www.worldscientific.com/doi/abs/10.1142/S0129626417400047

[36] M. Aldinucci, M. Danelutto, D. De Sensi, G. Mencagli, M. Torquati, Towards power-aware data pipelining on multicores, in: Proc. of HLPP2017: Intl. Workshop on High-Level Parallel Programming, Valladolid, Spain, 2017.

[37] D. De Sensi, Predicting performance and power consumption of parallel applications, in: Proc. of 24th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing, 2016, pp. 200 – 207.

[38] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, M. Schulz, Beyond dvfs: A first look at performance under a hardware-enforced power bound, in: Proc. of the 2012 IEEE 26th Intl. Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 947–953.