A Reconfiguration Algorithm for Power-Aware Parallel Applications

DANIELE DE SENSI, Computer Science Department, University of Pisa
MASSIMO TORQUATI, Computer Science Department, University of Pisa
MARCO DANELUTTO, Computer Science Department, University of Pisa

In current computing systems, many applications require guarantees on their maximum power consumption to not exceed the available power budget. On the other hand, for some applications, it could be possible to decrease their performance, yet maintaining an acceptable level, in order to reduce their power consumption. To provide such guarantees, a possible solution consists in changing the number of cores assigned to the application, their clock frequency and the placement of application threads over the cores. However, power consumption and performance have different trends depending on the application considered and on its input. Finding a configuration of resources satisfying user requirements is in the general case a challenging task.

In this paper we propose NORNIR, an algorithm to automatically derive, without relying on historical data about previous executions, performance and power consumption models of an application in different configurations. By using these models, we are able to select a close to optimal configuration for the given user requirement, either performance or power consumption. The configuration of the application will be changed on-the-fly throughout the execution to adapt to workload fluctuations, external interferences and/or application's phase changes. We validate the algorithm by simulating it over the applications of the PARSEC benchmark suite. Then, we implement our algorithm and we analyse its accuracy and overhead over some of these applications on a real execution environment. Eventually, we compare the quality of our proposal with that of the optimal algorithm and of some state of the art solutions.

CCS Concepts: **•Theory of computation → Self-organization; Online learning algorithms; •Hardware → Power estimation and optimization; •Software and its engineering → Software performance;**

Additional Key Words and Phrases: Power-Aware Computing; Online Learning; Self-Adaptive Runtime; Power Capping; Dynamic Concurrency Throttling; DVFS; Multi-Core

ACM Reference Format:
Daniele De Sensi, Massimo Torquati, and Marco Danelutto. 2016. A Reconfiguration Algorithm for Power-Aware Parallel Applications. *ACM Trans. Architec. Code Optim.* V, N, Article 43 (December 2016), 25 pages.
DOI: http://doi.acm.org/10.1145/3004054

1. INTRODUCTION

The problem of writing power-aware applications, providing guarantees on a minimum performance level or a maximum power consumption, is a challenging hot topic.

Indeed, power consumption management has become a major concern for data centers due to economic cost, reliability problems and environmental reasons. Several *power capping* techniques have been proposed in order to avoid exceeding the available power or thermal budget [Cochran et al. 2011b], [Gandhi et al. 2009], [Hoffmann

This work has been partially supported by the EU FP7-ICT-2013-10 project REPARA (No. 609666), the EU H2020-ICT-2014-1 project REPHRASE (No. 644235) and the University of Pisa Project PRA_2016_64.

et al. 2011], [Bhattacharya et al.]. Such constraints could change dynamically through time, for example due to different renewable energy availabilities during the day [Chen et al. 2013] and to satisfy them, we need to guarantee that each single server does not exceed a certain power budget [Lefurgy et al. 2008].

On the other hand, performance constraints may be set on applications that do not need to run at their maximum speed [Mishra et al. 2015], [Cochran et al. 2011a], [Li and Martínez 2006], [Shafik et al. 2015]. For example, in a video processing application we need to produce the output frames at a minimum rate. Processing the frames faster than that rate could lead to a higher power consumption, without necessarily improving the user experience.

A possible approach to the problem is to select the right combination of resources to be allocated to the application. We call a specific combination of resources a *configuration*. Static solutions based on complex analytical models are not fully appropriate for dynamic scenarios. In fact, the performance and the power consumption of the application may change during its execution, due to the intrinsic nature of the application or due to external perturbations. For these reasons, it is of paramount importance to develop models and runtime systems able to find the optimal configuration to be used and to dynamically adapt it throughout the entire application execution.

In this work we propose NORNIR, a new algorithm for finding system configurations that satisfy a power consumption or a performance requirement as provided by the application user. Furthermore, in case of a performance constraint, among all configurations that satisfy that constraint, the algorithm selects the configuration that has the lowest power consumption (and vice versa in the case of a power consumption constraint).

To achieve this goal, the proposed algorithm collects performance and power consumption data for some different configurations while the application is running and uses these data to perform online training of *prediction models*. These models will be used to select the best configuration satisfying the user requirement and to change it throughout the application execution. The configuration is changed by modifying the number of cores assigned to the application (*Dynamic Concurrency Throttling* – DCT), the clock frequency of these cores (*Dynamic Voltage and Frequency Scaling* – DVFS) and the placement of the threads on the cores (*Processor Affinity*).

The prediction models will be dynamically re-trained if the application enters into a phase characterised by different performance or power consumption behaviour (e.g., when moving from a CPU bound application phase to an I/O bound application phase). An important property of the proposed algorithm is that it does not rely on any information coming from previous application runs. Indeed, many existing approaches leverage on previously collected data (both from the same application or from different applications) to predict performance and power consumption of an application. However, such approaches often fail if the application has a behaviour that has not been previously observed. For this reason, we will consider a *worst-case scenario* where no previous data is available or where previously collected data cannot be exploited to infer the models for new applications, as typical in cloud computing [Armbrust et al. 2010]. Despite being in such worst-case scenario, as we will show in Section 5, we are able to achieve comparable or in some cases even better results with respect to those state-of-the-art approaches that leverage on knowledge coming from past applications executions.

In this work we focus on CPU power consumption since, as shown in [Zomaya and Lee 2012], CPU is still the most power consuming component in current multicores architectures. Moreover, operating on memory and I/O power consumption do not lead in general to significant improvements [Gray et al. 2008], [Zomaya and Lee 2012], [David et al. 2011].

To keep our approach as more general as possible, we will isolate and describe the mechanisms, available in most runtime systems, that a generic runtime system should provide in order to interact with our algorithm. To evaluate the quality of our proposal, we first simulate it over the applications of the PARSEC benchmark suite. Then, we implement our algorithm and we analyse its accuracy and overhead over some of these applications on a real execution environment. The source code of the algorithm and the applications used for the validation are publicly available[1].

The main contributions of this work may be summarised as follows:

(1) A new algorithm for self-adaptative, power-aware parallel applications. The proposed strategy is able to find a close to optimal [CORES, FREQUENCY, THREADS PLACEMENT] configuration that satisfies bounds on performance and/or on power consumption without any historical knowledge about previous applications runs. Moreover, it re-evaluates the optimal configuration if a change in the application or in the external environment occurs. The algorithm is able to distinguish intrinsic application changes from fluctuations caused by changes in the input pressure, improving the stability with respect to other existing approaches.
(2) Two models for predicting the performance and the power consumption of parallel applications at different configurations.
(3) The implementation of the algorithm for a real execution environment and the validation over several real-world applications with different execution behaviours, allowing to spot strong and weak points of the proposed strategy.
(4) The comparison of the results with those obtained by the optimal algorithm and by some other well-known solutions.

The rest of the paper is organised as follows. In Sec. 2 we describe some similar existing techniques and their strong and weak points. The algorithm proposed in this work is described in Sec. 3, while Section 4 presents the implementation of our solution by using a real runtime system. In Sec. 5 we describe the applications we used to validate our approach and the achieved results, comparing them with those obtained by other existing techniques. Eventually, Sec 6 concludes the paper, outlining some possible future directions.

2. RELATED WORK

In this section we describe some existing approaches for finding applications' configurations satisfying given performance and/or power constraints.

The first set of solutions is characterised by the use of complex analytical models, obtained by an *offline* training phase [Curtis-Maury et al. 2008b], [Curtis-Maury et al. 2008a], [Cochran et al. 2011b], [Cochran et al. 2011a]. These approaches are based on the collection of profiling data for a wide set of different applications, that will be used to train machine learning algorithms. When a new application is executed, the model will be instantiated with the parameters monitored at runtime in order to predict the best available configuration.

In [Delimitrou and Kozyrakis 2014], [Delimitrou and Kozyrakis 2013] the authors propose a cluster management system to allocate the right amount of resources to different applications given a performance requirement. By using classification techniques, they are able to perform this decision by exploiting information on previously executed applications. This work is orthogonal to ours since they consider multiple applications each one with its own requirement but only performance requirements can be specified. On the other hand, our work focuses on a single application scenario but allows to specify both performance and power consumption constraints. In addition to

[1]http://danieledesensi.github.io/nornir/

this, [Delimitrou and Kozyrakis 2014] is targeted towards cluster scenarios, operating on the number of cores per node and on the number of nodes used by each application, while our work is targeted to a single node scenario and operates at a finer grain, by changing the number of used cores, their frequency and the threads placement.

Albeit offline approaches are usually characterised by a low runtime overhead, their accuracy deeply depends on the choice of the data used to train the model. For example, if a behaviour that has never been experienced in the training phase occurs, these methods could produce sub-optimal solutions [Domingos 2012], [Mishra et al. 2015], [Shafik et al. 2015].

To mitigate these limitations, some models integrate the offline collected data with additional data obtained while the application is running [Mishra et al. 2015], [Filieri et al. 2014]. However, such solutions still share some of the limitations of the full offline approaches.

A different approach is to not rely on any previously collected information, similarly to what our algorithm does, and to only make decisions on the base of the data collected while the application is running. Such kinds of solutions are usually based on heuristics [Li and Martínez 2006], [Porterfield et al. 2013], [Sridharan et al. 2013], [Wang et al. 2015] or on more complex techniques like control theory [Maggio et al. 2010], optimisation algorithms [Petrica et al. 2013] or online machine learning [Hsu and Feng 2005], [Shafik et al. 2015]. Albeit they avoid some typical problems of the offline approaches, the collection of runtime data and the computation of the model while the application is running could introduce a significant overhead. However, as we will show in Section 5, when lightweight prediction algorithms are used it is possible to reduce such overhead.

Other methods, orthogonal to our solution, try to constrain the power consumption or the performance of an application by acting on mechanisms different from those considered in this work [Totoni et al. 2015]. Another alternative approach to the presented problem is approximate computing [Vassiliadis et al. 2015] trying to reduce power consumption by reducing the quality of the results computed by the application.

Overall, we can observe that existing approaches have several limitations. Firstly, in several cases, the proposed solutions are either simulated or the analysis is done on post-mortem data [Petrica et al. 2013], [Li and Martínez 2006], [Suleman et al. 2008], [Ding et al. 2008]. Despite a simulation may provide a good idea about the precision of the model, it is difficult to accurately estimate the run-time overhead of these methods. As we will show in Section 5, different overheads are introduced when implementing such models, which in turn may limit the usability of these approaches. Moreover, some solutions do not explicitly model the power consumption, thus only providing the possibility to specify performance constraints [Delimitrou and Kozyrakis 2014], [Li and Martínez 2006]. In many cases, they can only find the most efficient [Sridharan et al. 2013] or the most performing configuration [Pusukuri et al. 2011], [Curtis-Maury et al. 2008a], [Suleman et al. 2008]. In addition to that, they usually do not distinguish between the different types of events that can lead to a reconfiguration [Mishra et al. 2015], [Marathe et al. 2015], characterising all these events as generic *phase changes*. However, we claim that when such distinction is done, better results can be achieved, as we will show in Section 5.4.

Among the existing power-aware runtime systems, the one proposed in [Gandhi et al. 2009] works by forcing the core to alternate between active and idle periods. However, no consideration on its use for parallel applications has been done.

Table I. Comparison between some existing techniques with the solution proposed in this paper ('NORNIR').

	Power constr.	Perf. constr.	Idle cores	DVFS	DCT	Threads placement	Online learning only	Multi apps
[Gandhi et al. 2009]	✓		✓					
[Cochran et al. 2011b]	✓			✓	✓			
[Delimitrou and Kozyrakis 2014]		✓			✓			✓
[Marathe et al. 2015]	✓			✓	✓			
[Alessi et al. 2015]	✓	✓		✓	✓			
[Mishra et al. 2015]	✓	✓		✓	✓	✓		
NORNIR (2016)	✓	✓		✓	✓	✓	✓	

More recently, in [Mishra et al. 2015] the authors implemented a solution to find the configuration with the minimum energy consumption under a performance constraint, by using a model trained offline and by improving it with online collected data[2].

The solution proposed in [Marathe et al. 2015] is able to select the most performing configuration under a power budget by analysing all possible configurations.

The authors of [Alessi et al. 2015], proposed an extension to OpenMP in order to specify power and performance constraints. The best configuration will be selected by using hill-climbing heuristics. However, in the results they only show the performance constrained scenario while the accuracy of the heuristic is not deeply analysed. Differently from our approach, they also provide the possibility to specify custom adaptation mechanisms in addition to DVFS and DCT.

Other solutions [Wang et al. 2015], [Hsu and Feng 2005], [Porterfield et al. 2013] only apply either DVFS or DCT. However, as shown in [Curtis-Maury et al. 2008b], [Danelutto et al. 2015], when these two mechanisms are used together, the range of possible available configurations is extended and is possible to further reduce the power consumption.

In Table I, we report a summary of the main characteristics of the existing algorithms for performance and/or power capping. The work most similar to ours is [Mishra et al. 2015]. We will make a comparison in Section 5.

Despite the advantages with respect to existing solutions, our algorithm is not flawless. For example, it mainly works for long-running applications (several tens of seconds, at least). Furthermore, it does not perform well on applications with an extremely dynamic behaviour (i.e. many very small different phases). However, as shown in [Sembrant et al. 2012] many real applications are characterised by a small number of phases (4-8), thus the number of different trainings to be performed is usually limited and does not significantly impact application performance, as shown in Section 5. Moreover, in this work we considered the common scenario, typical of many recent research works in this field, where only one application requires specific performance or power consumption constraints. Considering multiple applications would pose significant extra challenges to be solved (some of them outlined in Section 6), which are clearly outside the scope of this paper. It is worth to remark that, the proposed solution allows other applications to run on the system at the same time. The only limitation is that only one application at a time is allowed to have requirements on performance or power consumption.

[2]The approach is general enough and it also works for the power capping scenario, albeit not considered in [Mishra et al. 2015].

Table II. Basic metrics that characterise the application.

METRIC	DESCRIPTION
Service time T_S	Is the average time between the beginning of the executions on two consecutive elements to be processed.
Service rate μ	Is the inverse of the service time, i.e. the average number of elements that can be processed by the application per time unit $\mu = \frac{1}{T_S}$
Interarrival time T_A	Is the average time between the reception of two consecutive elements to be processed by the application.
Arrival rate λ	Is the inverse of the interarrival time, i.e. the average number of elements received per time unit $\lambda = \frac{1}{T_A}$
Utilisation ρ	Is the ratio between the arrival rate and the service rate $\rho = \frac{\lambda}{\mu}$.
Bandwidth B	Is the average number of elements produced by the application per time unit. When $\rho < 1$ (i.e. $\lambda < \mu$), the system can manage all the arriving requests. In such case, we have $B = \lambda$. If this is not the case, the system is saturated and we have $B = \mu$. In general, we have $B = min(\lambda, \mu)$.
Power consumption P	Is the power consumed by the application.

3. OUR ALGORITHM

In this Section, we describe the theoretical foundations we exploit and the algorithm we propose to dynamically reconfigure an application to satisfy a given bound either on the performance or on the power consumption. First, we provide a brief background introducing the used terminology and the base assumptions, then we describe the algorithm starting from the simplest scenario, i.e. there are no internal or external interferences that alter the application behaviour. Finally, we describe how the algorithm can adapt the configuration of the application when internal and external conditions change during the application execution.

3.1. Background

Without loss of generality, we assume that the considered parallel application, and in particular its runtime system, can be modelled as a queueing network [Thomasian and Bay 1986]. In a nutshell, the parallel runtime is seen as a graph whose nodes are threads (service centres) and edges among threads are communication channels implemented as queues.

We characterise such queueing system with few metrics described in Table II.

In general, both the power consumption and the service rate depend on the amount of resources used by the application. Since in this work we are considering the number of physical cores n used by the application, their frequency f and the threads placement p, we denote the service rate as $\mu(n, f, p)$ and the power consumption as $P(n, f, p)$. In Figure 1 is reported a simple schema of the possible behaviour of μ, λ and B with respect to the number of cores used for executing the application.

The target of this work is to design an algorithm capable of managing the following scenarios:

— Given a bandwidth requirement $\overline{B}$, the algorithm should find a triplet $< \widetilde{n}, \widetilde{f}, \widetilde{p} >$ such that $B(\widetilde{n}, \widetilde{f}, \widetilde{p}) \geq \overline{B}$ and such that does not exist any other triplet $< n, f, p >$ that satisfies $P(n, f, p) < P(\widetilde{n}, \widetilde{f}, \widetilde{p})$. That means, among all the configurations satisfying the bandwidth requirement, the algorithm should pick the one with the lowest power consumption.

— Given a power consumption requirement $\overline{P}$, the algorithm should find a triplet $< \widetilde{n}, \widetilde{f}, \widetilde{p} >$ such that $P(\widetilde{n}, \widetilde{f}, \widetilde{p}) \leq \overline{P}$ and such that doesn't exists any other triplet $< n, f, p >$ with $B(n, f, p) > B(\widetilde{n}, \widetilde{f}, \widetilde{p})$. That means, among all the configurations satisfying the power requirement, the algorithm should pick the most performing one.

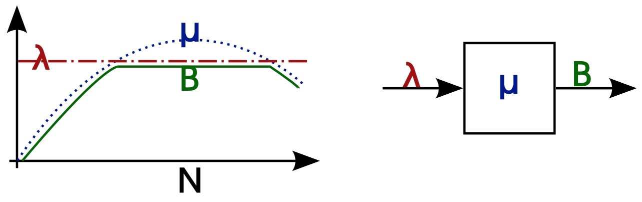

Fig. 1. Left: A possible behaviour of interarrival rate, service rate and bandwidth with respect to the amount of used number of cores n. Right: A simple runtime modelling with respect to the metrics of interest.

The algorithm is designed in order to satisfy such requirements even if the service rate or the interarrival rate change during application execution.

Indeed, service rate may change due to external interferences (e.g. other applications running on the system) or to intrinsic characteristics of the application. For example, consider a video surveillance application for counting people in a crowded places. Even if the interarrival rate is steady through time, the amount of time required to process each frame may be different in different part of the day. Accordingly, we could have a lower service rate when there are many people in a video frame and a higher service rate when the scene is not very crowded.

On the other hand, the interarrival rate mainly depends on external factors and it is not under the direct control of the application. It may change during application execution as well. If we consider for example a network monitoring application, the interarrival rate will be higher during working hours while it will be lower during the night. The proposed algorithm is able to detect these two different cases and to manage them differently.

If the user knows the number of elements s that have to be processed by the application, instead of setting a bandwidth requirement, he can express the performance requirement as a maximum execution time T. Since $T = B \cdot s$, the algorithm will automatically translate the execution time requirement into a bandwidth requirement. If the average bandwidth changes during application progress, the bandwidth requirement will change too, in order to satisfy the execution time constraint specified by the user.

3.2. Stationary behaviour

To simplify exposition, we first consider the case where both $\mu(n, f, p)$ and λ are constant throughout the execution. Then, we will extend the algorithm to consider a more generic case where they both can change.

We consider two distinct phases throughout the application execution: *calibration* phases and *steady* phases (see Figure 2).

When the application starts, the runtime system enters the calibration phase. In this phase, it executes the following steps:

(1) The configuration of the application is changed, by selecting a configuration not yet visited.
(2) The service rate and power consumption of the application are monitored for a short predefined period of time.
(3) The monitored data is used to refine the power consumption and service rate models.
(4) Values predicted by the model are then compared with those monitored in the current configuration. If the prediction error is lower than a specified threshold value, the calibration phase finishes. Otherwise, the process is iterated.

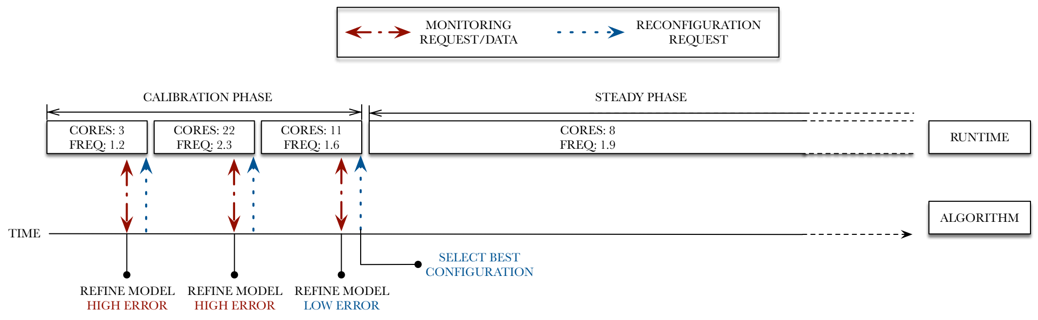

Fig. 2. Different phases of our algorithm.

Mainly, the runtime system is training a machine learning algorithm while the application is running. Step 4 is based on the assumption that the error is uniformly distributed over all possible configurations so that, by looking at the prediction error of the current configuration, we may have an estimation of the error in all other possible configurations. We observed that this property holds in practice for the prediction algorithms we propose. However, in case the error is not uniformly distributed, the effect is that the algorithm will potentially miss-predict some configurations and will select a sub-optimal configuration.

When a calibration phase finishes, we have accurate models for predicting service rate and power consumption of the application in all possible configurations. Starting from the service rate model, we can obtain a prediction of $B(n, f, p)$ by computing $B(n, f, p) = min(\mu(n, f, p), \lambda)$. To obtain λ, we can then use the service rate model to select the configuration with the highest service rate. Since $\rho(n, f, p) = \frac{\lambda}{\mu(n,f,p)}$, then $\lambda = \mu(n, f, p)\rho(n, f, p)$. According to queueing theory, this equation holds only if $\rho(n, f, p) < 1$. If this is not the case, we can consider $\lambda = \infty$. Indeed, independently from its real value, it will always be greater than μ for every (n, f, p) triplet. Please note that this is an important step. Indeed, if we just consider $B(n, f, p) = \mu(n, f, p)$ we could have a low utilisation of resources since the application is actually throttled by the low input rate, thus leading to a power inefficient execution.

Once we are able to predict $\mu(n, f, p)$ and $P(n, f, p)$ for all $< n, f, p >$ triplets[3], the algorithm can select the best configuration according to the requirements given by the user and moves the application to that configuration, thus entering in the steady phase. If, according to the models, there are no configurations that satisfy the requirement, the algorithm selects the closest one to the requirement.

It is worth noting that, during the calibration phase the application is running, and, when switching from calibration to steady phase, the application is neither stopped nor the previously computed results are discarded. However, during calibration phase we still do not have the service rate and power models, and thus we could run the application into configurations which might violate the requirements. For this reason, calibration phases should last as short as possible. Alternatively, it is possible to set a maximum duration for the calibration phase (concerning visited configurations or absolute time) and to trade lower prediction accuracy with shorter calibration time. In Section 5 we analyse the trade-off between prediction accuracy and number of configurations visited during calibration phase as well as the impact of the calibration phase.

[3]The range of possible frequencies is discretized according to the capabilities of the underlying hardware. In general, only some specific frequencies can be selected.

To shrink the duration of the calibration phase, the algorithm could explore multiple configurations in parallel. We do not consider this possibility in our experiments due to the limited frequency scaling capabilities of our target architecture (we can partition the cores at most in two groups running at two different frequency). Another way to reduce calibration time is to exploit information about previously executed applications [Mishra et al. 2015], [Delimitrou and Kozyrakis 2014]. However, as anticipated in Section 1, we are considering a worst-case scenario, where such information is not present or cannot be exploited, for example, because the application is too different from those already submitted to the system.

The following pseudocode shows the proposed algorithm when $T_S(n, f, p)$ and T_A never change throughout the execution.

ALGORITHM 1: Reconfiguration algorithm - Stationary case - Part 1

```
Function FindHighestMuConf()
    $\mu_{max} \leftarrow 0$;
    forall n, f, p do
        $\mu_{pred}, P_{pred} \leftarrow$ Predict(n, f, p);
        if $\mu_{pred} > \mu_{max}$ then
            $\mu_{max} \leftarrow \mu_{pred}$;
            best ← n, f, p;
        end
    end
    return best;

Function GetBandwidthIn()
    maxMuConf ← FindHighestMuConf();
    ChangeConfiguration(maxMuConf);
    Sleep(samplingPeriod);
    $\mu_{real}, P_{real}, \rho_{real} \leftarrow$ Monitor();
    if $\rho_{real} < 1$ then
        return $\rho_{real}\mu_{real}$
    else
        return $\infty$;
    end

Function FindBestConfiguration()
    $P_{min} \leftarrow \infty$;
    $\lambda \leftarrow$ GetBandwidthIn();
    forall n, f, p do
        $\mu_{pred}, P_{pred} \leftarrow$ Predict(n, f, p);
        $B_{pred} \leftarrow min(\lambda, \mu_{pred})$;
        if $B_{pred} \geq B_{req}$ and $P_{pred} < P_{min}$ then
            $P_{min} \leftarrow$ predictedPower;
            best ← n, f, p;
        end
    end
    return best
```

ALGORITHM 2: Reconfiguration algorithm - Stationary case - Part 2

```
Function Calibrate()
    $\mu_{real}, P_{real} \leftarrow$ Monitor();
    $\epsilon_\mu \leftarrow \infty$;
    $\epsilon_P \leftarrow \infty$;
    repeat
        Refine(c, $\mu_{real}, P_{real}$);
        c ← PickUnvisitedConfiguration();
        $\mu_{pred}, P_{pred} \leftarrow$ Predict(c);
        ChangeConfiguration(c);
        Sleep(samplingPeriod);
        $\mu_{real}, P_{real} \leftarrow$ Monitor();
        $\epsilon_\mu \leftarrow \mid \frac{\mu_{pred}-\mu_{real}}{\mu_{real}} \mid$;
        $\epsilon_P \leftarrow \mid \frac{P_{pred}-P_{real}}{P_{real}} \mid$;
    until $\epsilon_\mu < threshold$ and $\epsilon_P < threshold$;
    return FindBestConfiguration();

Function Main()
    c ← Calibrate();
    ChangeConfiguration(c);
```

3.3. The prediction algorithm

By analysing the algorithm, we can identify four primary functions: *Monitor*, *Refine*, *Predict* and *ChangeConfiguration*. We now describe the *Refine* and *Predict* routines, while *Monitor* and *ChangeConfiguration* are implementation dependent and will be discussed in Section 4.

We first describe how to predict performance and power consumption for different cores n and frequency f. Then, we extend the models to consider different threads placements p. To predict power consumption and service rate of the application we decided to start with some general analytical models and to derive their application and architecture specific parameters at runtime by using *linear regression analysis*. By using linear regression [Montgomery and Peck], we model the relationship between two or more independent variables (called *predictors*) and a dependent variable (called *response*) by fitting a linear equation to observed data. In our case, the predictors are the number of cores used by the application and their frequency, while the response is the service time or the power consumption. Suppose we made a set of n *observations* to obtain the values of the responses $y_1, y_2, \ldots, y_n$. Let $x_i = x_{i,1}, x_{i,2}, \ldots, x_{i,p}$ denote the p predictors for the observation i. Then we have:

$$y_i = a_0 + a_1 x_{i,1} + \cdots + a_p x_{i,p} + \epsilon_i \tag{1}$$

where a_i is a regression coefficient and ϵ_i is a term representing a random error due to measurement error or fluctuations in the results.

By fitting a regression model to observations, we determine the a_i coefficients, thus enabling the prediction of the responses for the unobserved predictors. To fit the model, we use the *least squares method*, minimising the sum of the squares of the residuals, where a residual is the difference between the real value of the dependent variable and the value predicted by the model.

In our algorithm, the *Refine* routine will add a new observation and the *Prediction* routine will use the observations we made to derive the a_i coefficients. Accordingly, we need to express service time and power consumption in a form similar to the one in Equation 1, where a_i coefficients are architecture and application specific values.

Concerning the service time, we model it by using the *Universal Scalability Law* [Gunther 2006]:

$$\begin{aligned} T_S(n) &= \frac{T_S(1)}{n} + \frac{T_S(1) \cdot \alpha \cdot (n-1)}{n} + T_S(1) \cdot \beta \cdot (n-1) = \\ &= a_1 \frac{1}{n} + a_2 \frac{n-1}{n} + a_3(n-1) \end{aligned} \tag{2}$$

where α and β are application dependent constants, representing two possible issues that limit the achievable scalability. α represents the weight of the contention on shared data, while β represents the cost needed to keep the writable data coherent.

Since the above equation does not depend on the frequency f, we will only make observations on configuration where f is the minimum frequency possible f_{min}. In this way, the model we derive will be able to only predict service times of the form $T_S(n, f_{min})$.

To obtain the service time for different frequencies we use the following algorithm. First we monitor the service time $T_S(1, f_{max})$ and we compute $\gamma = \frac{T_S(1,f_{min})}{T_S(1,f_{max})}$. Then we can compute the service times for any configuration where $f = f_{max}$ as $T_S(n, f_{max}) = \frac{T_S(n,f_{min})}{\gamma}$. To compute service times for frequencies different from f_{max} and f_{min}, we observe that the service time has a linear relationship with the frequency [Xie et al. 2005], [Lee et al. 2007]. Therefore, for a given n, we can express $T_S(n, f) = a + b \cdot f$. Since we already know $T_S(n, f_{min})$ and $T_S(n, f_{max})$, we can obtain the $T_S(n, f)$ by using the equation for a line passing through two points:

$$T_S(n, f) = \frac{T_S(n, f_{max}) - T_S(n, f_{min})}{f_{max} - f_{min}} \cdot f + \frac{T_S(n, f_{min}) \cdot f_{max} - T_S(n, f_{max}) \cdot f_{min}}{f_{max} - f_{min}} \tag{3}$$

Concerning the power consumption model, it is inspired to the one presented in [De Sensi 2016]. However, the model has been significantly improved to consider different threads placements and different frequencies for active and inactive CPUs. According to [Chandrakasan and Brodersen 1995], [Alonso et al. 2014], [Kim et al. 2003], the power consumption of an active CPU can be modelled as:

$$P(n, f, v) = vI_{leak} + Acv^2fn \tag{4}$$

where v is the voltage, I_{leak} is the *leakage current*, A is the activity factor and c is the capacitance. For our purposes, we can consider A, c and I_{leak} as constants.

When more than one CPU is present on the system, we can apply this formula separately for each of them. To further reduce power consumption, we run the unused CPUs at the minimum available frequency f_{min}, with a corresponding operating voltage v_{min}. Since for unused CPUs we have $n = 0$, their power consumption is $v_{min}I_{leak}$. If $\overline{k}$ and $\underline{k}$ are the number of active and inactive CPUs respectively, f is the frequency of active CPUs and v is the voltages of the active CPUs, then we have :

$$P(n, f, v, \overline{k}, \underline{k}) = \overline{k}vI_{leak} + \underline{k}v_{min}I_{leak} + Acv^2fn \tag{5}$$

To remove the dependency from v, we can observe that the voltage is strictly correlated to the frequency, since by increasing the frequency we raise the operating voltage and vice versa. Consequently, we can use a tabular function $V(f)$ to get the voltage value associated to a specific frequency level f. This function is architecture specific and may be obtained programmatically or by using the values provided by the CPU vendor. We can then rewrite equation 5 as:

$$P(n, f, \overline{k}, \underline{k}) = \overline{k}V(f)I_{leak} + \underline{k}v_{min}I_{leak} + AcV(f)^2fn \tag{6}$$

Let K be the number of CPUs available on the target architecture, then we have $\underline{k} = K - \overline{k}$.

$$\begin{aligned} P(n, f, \overline{k}) &= \overline{k}V(f)I_{leak} + (K - \overline{k})v_{min}I_{leak} + AcV(f)^2fn = \\ &= I_{leak}(\overline{k}(V(f) - v_{min}) + Kv_{min}) + AcV(f)^2fn \end{aligned} \tag{7}$$

Since the number of active CPUs depends on the specific thread placement, we will discuss how to remove this dependency in Section 3.3.1

3.3.1. Different threads placements. Some applications may be sensitive to thread placement (also known as thread-to-core affinity). For example, due to contention on shared caches, some memory intensive applications may benefit from a sparse placement, trying to minimise the number of caches shared by active threads. However, due to a higher amount of used resources, this placement will be in general more power consuming. We will consider in this work two different types of threads placements: *linear* and *interleaved*. *Linear* placement minimises the number of used CPUs, by placing a thread on a new CPU only if the all the cores on the used CPUs have been already allocated to other threads. On the other hand, *interleaved* placement minimises the amount of resources shared by threads, by allocating one thread per CPU in a round-robin way. For example, suppose to have a machine with 2 CPUs, each one with 4 cores on top and that the algorithm must place 6 threads over them. In the *linear* case it will place 4 threads on the first CPU (one per core) and 2 threads on the second one while with an *interleaved* placement, the algorithm will allocate 3 threads on the first CPU and 3 threads on the second one.

To evaluate the performance of these two different alternatives, the algorithm will simultaneously derive one performance model for the *linear* case and one for the *interleaved* case. During the calibration phase, configurations characterised by a *linear* placement will be used to refine the model for the *linear* case while configuration characterised by an *interleaved* threads placement will be used to improve the other performance model. When the optimal configuration must be selected, both models will be evaluated.

A similar process is performed for the power consumption model. However, in this case the number of used/unused CPUs changes according to the specific thread placement. Let $\widetilde{k}$ be the number of cores available for each CPU. For the linear placement we have:

$$\overline{k} = \left\lceil \frac{n}{\widetilde{k}} \right\rceil \tag{8}$$

While for the interleaved placement we have:

$$\overline{k} = min(n, K) \tag{9}$$

Having at most one thread per physical core, these two values clearly correspond to minimising and maximising the number of used CPUs for a given n.

Eventually, in case of *linear* placement, we may rewrite Equation 7 as:

$$\begin{aligned}P(n,f) &= I_{leak}\left[\left\lceil\frac{n}{\tilde{\tilde{k}}}\right\rceil\Big(V(f)-v_{min}\Big)+Kv_{min}\right]+AcV(f)^2fn=\\&=a_0\left[\left\lceil\frac{n}{\tilde{\tilde{k}}}\right\rceil\Big(V(f)-v_{min}\Big)+Kv_{min}\right]+a_1V(f)^2fn\end{aligned}\tag{10}$$

For *interleaved* placement, Equation 7 can be rewritten as:

$$\begin{aligned}P(n,f) &= I_{leak}\left[min(n,K)\Big(V(f)-v_{min}\Big)+Kv_{min}\right]+AcV(f)^2fn=\\&=a_0\left[min(n,K)\Big(V(f)-v_{min}\Big)+Kv_{min}\right]+a_1V(f)^2fn\end{aligned}\tag{11}$$

We would like to point out that this approach to application placement is general and can be extended to consider other threads placements as well.

3.4. Variations in service time and interarrival time

During the steady phase, the runtime system keeps collecting and analysing monitored data. Two different types of events can trigger a configuration change. The first is a change in the application phase (e.g. the application finishes a CPU intensive phase and enters into an I/O intensive phase). In this case, the already computed models would no longer be valid and a new calibration phase should be executed to obtain new service time and power consumption models. To detect such situation, several techniques have been recently proposed [Sembrant et al. 2011], [Nagpurkar et al. 2006]. Such techniques can be applied online while the application is running, thus are particularly suitable for our purposes. Therefore, we introduce a new function, called *PhaseChanged*, that tell us if an application phase change has been detected or not. If a phase with such characteristics was already detected before, the already computed models can be recalled and the new calibration can be avoided. The other event that can trigger a reconfiguration is a change in the arrival rate λ. Indeed, since we express the performance experienced by the user as $B(n,f)=min(\lambda,\mu(n,f))$, if λ changes, we may need to change configuration in order to satisfy user requirement. However, in this case there is no need to obtain new models and we can just recompute the best configuration. Distinguishing between these two types of events is a crucial point of our algorithm and it is one of the main characteristics that distinguish it from existing online approaches. We will evaluate the importance of this point in Section 5.4.

The new general control loop of our algorithm is shown in the following pseudocode. The *Calibrate()* and *FindBestConfiguration()* functions are the same one presented in the stationary case scenario.

ALGORITHM 3: Reconfiguration algorithm - General case

```
Function Main()
    c ← Calibrate();
    ChangeConfiguration(c);
    while true do
        Sleep(samplingPeriod);
        if PhaseChanged() then
            c ← Calibrate();
            ChangeConfiguration(c);
        else
            if InputBandwidthChanged() then
                c ← FindBestConfiguration();
                ChangeConfiguration(c);
            end
        end
    end
```

4. IMPLEMENTATION

In this Section, we discuss the main choices we made to implement the algorithm on a real execution environment.

To reach this goal, we must provide suitable *Monitor()* and *ChangeConfiguration()* routines. The *Monitor()* routine should be able to interact with the runtime to extract μ and ρ values. Such values can be computed by the runtime by performing internal monitoring or by providing appropriate instrumentation calls to the application. Since it is not currently possible to isolate the power consumption of individual applications, the power consumption P can, in general, be obtained by monitoring that of the whole system. Therefore, the algorithm limits the power consumption of the entire architecture rather than that of the runtime only. Thus, if other applications are running, we reconfigure the runtime by considering the power consumed by all running applications. Power consumption can be monitored by interacting with the operating system or with external power metering devices. Concerning the *ChangeConfiguration()* routine, it should be able to change the number of physical cores used by the runtime, their clock frequency and the threads placement. All these operations could be done by interacting with the operating system only. For example, the number of used cores and the threads placement can be changed by appropriately setting the thread affinities, while the clock frequency can be changed with appropriate tools (e.g. `cpufreq-set`). Alternatively, it is possible to ask the runtime system to change the number of cores it uses by changing the number of activated threads. For example, OPENMP provides a `omp_set_num_threads` call, and similar functions are present in other runtime systems. Such interactions are depicted in Figure 3.

For all the interactions with the operating system, we used the MAMMUT library[4]. MAMMUT (MAchine Micro Management UTilities) provides a set of functions for the management of local or remote machines. It abstracts, through an object oriented interface, a set of features normally provided by the OS (e.g. clock frequencies management, topology analysis, energy profiling). The runtime system we choose to test our algorithm is FASTFLOW [Danelutto and Torquati 2015], a C++ parallel programming framework. This choice has been driven by the fact that we have the expertise required to interact with its internal mechanisms in order to implement *Monitor()* and

[4]http://danieledesensi.github.io/mammut/

ChangeConfiguration() routines. However, any runtime system that provides similar mechanisms can be used.

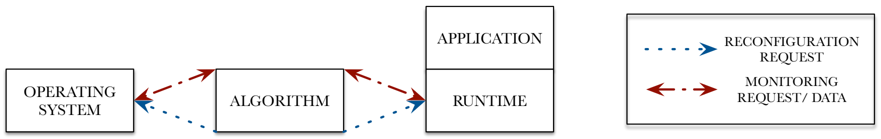

Fig. 3. Interaction between the algorithm, the runtime, and the operating system.

In our implementation, the algorithm is executed in a separate external thread, which interacts with the runtime via shared memory. The sampling interval is set by default to 0.1 seconds for the calibration phase and to 1 second for the steady phase. The sampling is more aggressive during calibration in order to reduce its duration. Furthermore, to avoid reconfigurations due to temporary spikes in the observed data samples, we apply exponential smoothing to monitored data. Accordingly, temporary fluctuations are filtered out, and the algorithm only reacts to steady and persistent changes in the runtime behaviour.

Our implementation of the algorithm and the applications we used to perform our experiments have been released as open source[5].

5. RESULTS

In this Section, we first simulate the proposed algorithm over all the applications of the PARSEC benchmark suite, comparing the results obtained by our algorithm against some well-known state of the art solutions. Then, we report the results obtained by a real parallel implementation of some of the PARSEC applications obtained by using the FASTFLOW parallel framework. We will analyse the results showing the overhead introduced by the algorithm with respect to the optimal solution, both in stationary and non-stationary situations.

All experiments were conducted on an Intel workstation with 2 Xeon E5-2695 @2.40GHz CPUs, each with 12 2-way hyperthreaded cores, running with Linux x86_64. This machine has 13 possible frequency levels: from 1.2GHz to 2.4GHz with 0.1GHz steps. We did not used hyperthreading, thus we have 624 possible configurations (13 frequency steps $\times$ 24 physical cores $\times$ 2 possible placements). The MAMMUT library has been used to compute the voltage table. On the considered platform, MAMMUT uses RAPL counters [Hähnel et al. 2012] to measure the power consumption.

In all the presented results, we only considered the time spent in the parallel sections of the applications, without considering initialisation and cleanup phases. This is a common assumption (as in [Cochran et al. 2011a] and [Pusukuri et al. 2011]) motivated by the fact that this work mainly focuses on optimising parallel phases. Initialisation and cleanup of the algorithm are still included, we just do not include the serial part of the application (e.g. loading the data-set from the disk).

5.1. Prediction simulation

To perform the simulation of all PARSEC applications, we first run each application in each possible system configuration. Then, re-ran all applications using the NORNIR runtime by simply overriding the call to the *Monitor()* function in the control loop in order to use as monitoring data those obtained by the previous runs. By applying this

[5]http://danieledesensi.github.io/nornir/

process, we can simulate only the stationary case (i.e. no changes in service rate and arrival rate). The non-stationary case will be analysed considering a real implementation of some parallel applications. We set the thresholds for the calibration phase termination to 10%. Higher values would lead to faster convergence and lower accuracy while lower values would result in slower convergence and higher accuracy. The tradeoff between performance model accuracy and number of configurations visited during the calibration phase is analysed in Figure 4. We achieved similar results for the power consumption model but they are not shown due to space constraints.

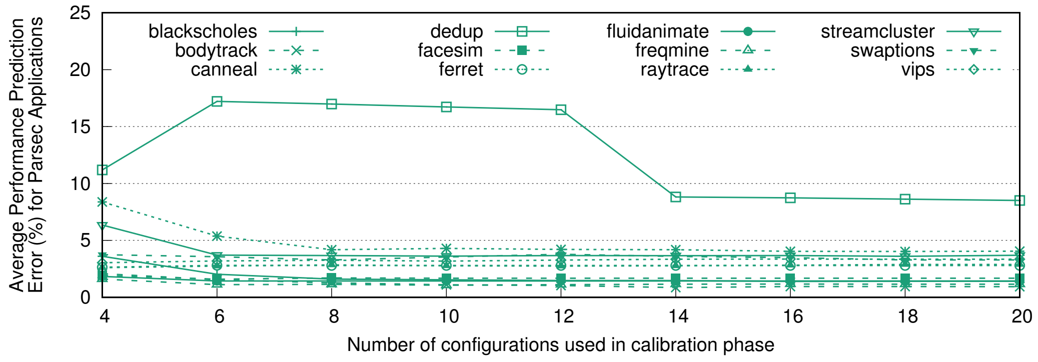

Fig. 4. Tradeoff between performance prediction accuracy and number of configurations visited during calibration phase.

In the figure, we have on the x-axis the number of configurations explored during the calibration phase, while on the y-axis we have the average error over all the PARSEC applications. For each PARSEC application, we considered as error the Mean Absolute Percentage Error (MAPE), defined as:

$$\varepsilon = \frac{1}{c}\sum_{i=1}^{c}\left|\frac{A_i - E_i}{E_i}\right| \tag{12}$$

where c is the total number of configurations (both those visited as well as those non-visited in the calibration phase), A_i is the real performance and E_i is the predicted performance. As shown in the figure, the proposed performance prediction model is able to quickly reach a low error with few calibration points.

To evaluate the quality of our proposal, we need to test if our algorithm is able to find a solution that satisfies the user requirement and, if so, how much such solution differs from the optimal one. Indeed, due to inaccuracy in the models and to overheads introduced by the algorithm, the configuration found by NORNIR could, in general, have worse performance or power consumption than the optimal one. For example, let's suppose to have a power budget of 40 Watts and that the optimal configuration consumes 39 Watts and has a service rate of 100 elements per seconds. Due to prediction inaccuracy, we could estimate that this optimal configuration consumes 41 Watts and then discarding it, or instead, estimate as the optimal solution a configuration that has the requested power consumption but a lower service rate (e.g. 80 elements per second), rather than the true optimal configuration. We say that the configuration found by NORNIR has a *loss* of $\frac{100-80}{100} \times 100.0 = 20\%$ with respect to the optimal solution. Similar situations may occur for performance constraints. Moreover, in some cases such errors may lead to a violation of the user requirements.

To avoid biases in the results due to specific requirements choices, we need to test NORNIR over a wide range of different requirements. To this purpose, we adopted a technique used in similar works [De Sensi 2016], [Mishra et al. 2015]. For example, if we need to specify a power constraint for an application that has a minimum power consumption of 20 Watts and a maximum of 220 Watts, then we will use bounds equal to 40, 60, ... and 200. Basically, we slice the range in 10 equal intervals in order to cover the entire spectrum of possible power consumptions, avoiding biases due to specific constraints choices. The same technique has also been adopted for the performance bounds. For each application, we show the average values obtained over all these constraints.

We compared our reconfiguration algorithm with the one presented in [Mishra et al. 2015]. Such solution uses an offline approach based on previous profiling of the applications. This profiling data is integrated with information collected online while the application is running. To perform the experiments, we used the publicly available source code provided by the authors. We call this approach MIXED. For the experiments with a performance constraint, we have also compared our approach with a well-known heuristic presented in [Li and Martínez 2006]. This heuristic uses a combination of hill climbing and binary search algorithms to find the lowest power consuming solution under a performance constraint. However, it is not able to find the most performing solution under a given power budget. We call this algorithm HEURISTIC.

The first result we obtained is that both the NORNIR algorithm and the HEURISTIC have been able to satisfy the specified user requirement in all the presented test cases. On the contrary, the MIXED solution missed the requirement in 51.4% of the cases when a power consumption constraint is specified and in 2.8% of the cases where a minimum performance level is required. Concerning the number of configurations visited before finding the optimal one, both NORNIR and HEURISTIC require 8 steps in average for both the performance constraint and the power constraint cases. However, as shown in Table III, the solutions found by the HEURISTIC algorithm are generally worst than the one found by NORNIR. On the other hand, MIXED performs a fixed number of steps (set to 20 by default in the source code provided by the authors). However, albeit the number of steps is higher than those required by NORNIR, the accuracy is not better. Moreover, we believe that the reason why this approach misses the target in 51.4% of the power consumption requirements is due to the choice of using a fixed amount of configurations to derive the model. Indeed, if it were possible to build the model incrementally, the accuracy would be probably improved by adding some more point.

In Table III we show the loss of the solutions found by the algorithms with respect to the optimal configurations, for all the applications of the PARSEC benchmark suite[6]. For the MIXED solution the results for some benchmark are marked with N.D.. Indeed, MIXED requires all the applications to have the same number of configurations. However, some of these applications can only run with some specific number of threads. Accordingly, it was not possible to predict values for such applications. Other results are marked as MISS, to indicate that the algorithm was not able to satisfy the user requirement in any of the tests for that application.

5.2. Accuracy and overhead analysis on a real environment

In this Section, we analyse the overhead introduced by the algorithm execution and its accuracy on real parallel executions in a stationary situation. To obtain these results, we implemented two PARSEC applications (*blackscholes* and *canneal*) by using

[6]x264 benchmark is missing because we were not able to run it over our target architecture.

Table III. Loss percentage of the found configurations with respect to the optimal ones.

Benchmark	Performance constraint			Power constraint	
	Nornir	Mixed	Heuristic	Nornir	Mixed
Blackscholes	2.264	7.527	4.935	0.881	0.808
Bodytrack	0.571	5.766	2.026	1.471	2.612
Canneal	1.749	10.726	6.106	2.831	15.313
Dedup	4.815	n.d.	13.268	6.092	n.d.
Facesim	0.122	n.d.	26.623	0.000	n.d.
Ferret	0.000	n.d.	7.679	0.549	n.d.
Fluidanimate	0.357	n.d.	20.892	1.164	n.d.
Freqmine	1.268	2.831	2.836	1.838	Miss
Raytrace	2.239	4.144	6.385	0.296	Miss
Streamcluster	3.143	15.762	9.327	5.763	9.304
Swaptions	1.719	1.635	2.936	1.928	0.471
Vips	0.769	1.016	5.397	0.428	0
Average	**1.584**	6.176	9.034	**1.936**	4.751

Table IV. Percentage of tests for which a solution satisfying the user requirement was found.

	Nornir	Nornir+Pdp16	Mixed	Heuristic	Rapl
% Success	**100%**	100%	84,03 %	93%	80,5%

the FastFlow parallel framework[7]. In addition to that, we used some applications already implemented in the FastFlow framework: a data compressor (*pbzip2*[8]) and a video denoiser (*denoiser*). For the two Parsec applications we used the *native* input provided together with the applications. For *pbzip2* we used a 6,3GB file containing a dump of all the abstracts present on the English Wikipedia on 01/12/2015. For the denoiser application we used a 1 hour length, 480x270 resolution video.

We compare the results with those obtained by replacing the prediction models of Nornir with the models presented in [De Sensi 2016] (denoted as Nornir+Pdp16). For the power capping experiments, we also compared our algorithm with a hardware-enforced solution (Rapl) available on newer Intel's processors [Rountree et al. 2012], which can be used to limit the maximum power consumption over a time window. We set a time window of 1 second, equal to the one used by Nornir. For each constraint, we repeated the test 10 times and for each application we show the average value of the obtained results over all possible constraints, as well as their pooled standard deviation.

User requirements satisfaction. In Table IV we report the percentage of tests for which a solution satisfying the user requirement was found.

Loss with respect to optimal solution. However, even when a configuration is found, such configuration could be far from the optimal, due to overhead and prediction inaccuracy. To understand the possible overhead introduced by the algorithm execution, consider the case where the user set a power constraint of 50 Watts and let's assume that the most performing configuration under this budget consumes exactly 50 Watts. Since the execution of the algorithm is also contributing to the power consumption, the real power budget available for the application execution will be less than 50 Watts and the optimal configuration under these conditions will most likely process a lower bandwidth with respect to the true optimal one. Accordingly, the results presented in Figure 5 about loss percentage, include both the overhead introduced by the algorithm and the inherent inaccuracy of the prediction models.

[7]http://mc-fastflow.sourceforge.net
[8]http://compression.ca/pbzip2/

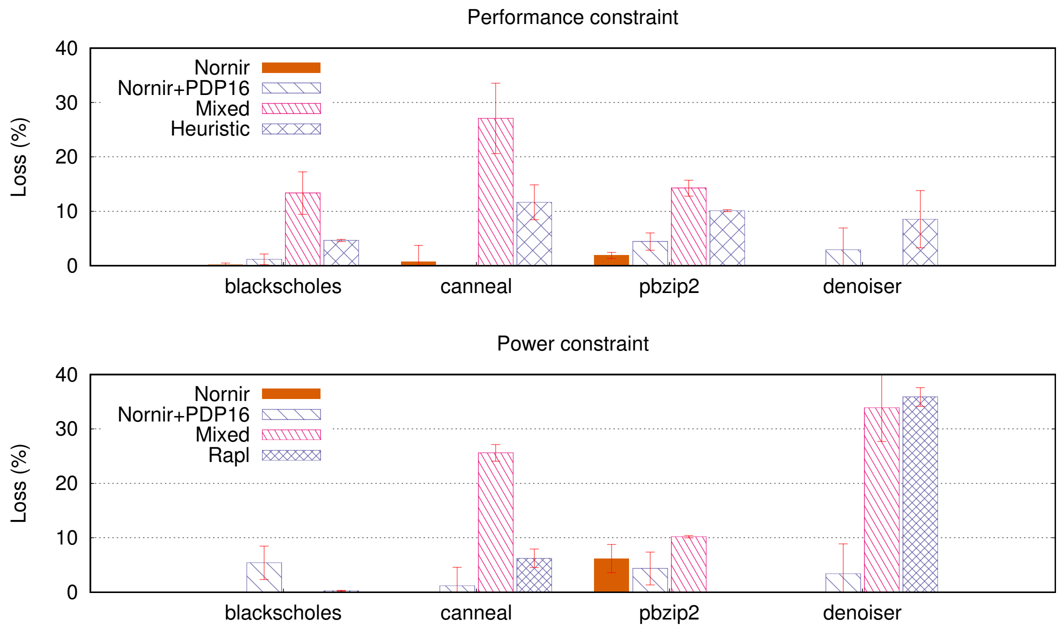

Fig. 5. Loss percentage with respect to the optimal configuration when a constraint on the performance or power consumption is required. When the bar is not shown, the algorithm found the optimal solution.

As we can see from the figure, the configurations found by our algorithm are at most 5% worst than the true optimal configurations. When the bar is not shown, there is no loss with respect to the optimal configuration. In all the experiments NORNIR found a better solution than the other algorithms. On about 40% of the experiments, the RAPL solution violates the power constraint. We believe that this happens because such solution mainly operates on hardware mechanisms while NORNIR can also modify the structure of the application. However, it is worth noting that the RAPL solution has the advantage of not requiring any explicit interaction with the runtime system of the applications. For some constraints, NORNIR is able to reduce the power consumption by 10.37% and to increase the performance by 29.64% with respect to NORNIR+PDP16.

Calibration time. Another important analysis is about the time spent in the calibration phase. Albeit during the steady phase our model always satisfies the user requirements, during the calibration phase we have no guarantees, since the algorithm is still deriving prediction models and is moving from one configuration to another without knowledge of their performance and power consumption. In the simulated experiments we gave results about the average number of steps required. However, we would like to analyse the percentage of the total execution time spent during the calibration phase for real executions.

Figure 6 reports these data for each application. The results show that the runtime system spends around 10% of the total execution time for calibration. Note that, while the algorithm is calibrating, the application is producing useful results. It is important to remark that, albeit our approach mainly targets long running applications, 20% of our test cases have an execution time below 20 seconds. For applications with longer execution time, the calibration time has a much lower impact. Alternatively, the user can specify a maximum calibration duration, at accuracy's expense. The HEURISTIC approach is always characterised by a lower calibration time, due to the simplicity

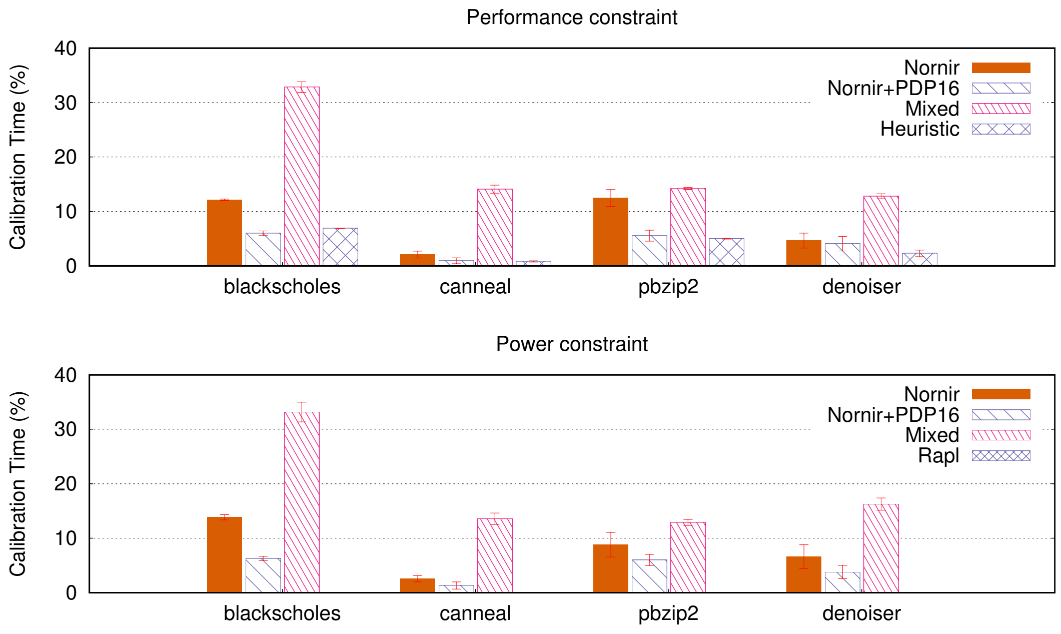

Fig. 6. Percentage of total execution time spent during the calibration phase.

Table V. Average calibration time (seconds) of NORNIR for each application.

APPLICATION	BLACKSCHOLES	CANNEAL	PBZIP2	DENOISER
CALIBRATION TIME	2.123	0.488	15.793	4.401

of the algorithm. However, the solution found is usually worst than the one found by NORNIR. On the other hand, MIXED solution always exhibits a higher calibration time, both caused by the fact that the number of configurations to be explored is fixed and also by the higher latency required to compute the model once all the data has been gathered. NORNIR+PDP16 algorithm is characterised by a lower calibration time with respect to NORNIR because the number of visited configurations is lower since different threads placements are not considered. Calibration time for the RAPL solution is not reported since it is always very close to 0.

In Table V we reported the absolute average calibration time in seconds spent by NORNIR for each application. The reason why different applications have different calibration times is due to some implementation choices. When NORNIR sends a monitoring request to the application runtime, the runtime will forward the request to each of its threads. However, the threads can only reply to the request if they are not currently processing an input element. Accordingly, the longer is the average time to process an input element, the longer will be the time the algorithm has to wait for the monitored data, and consequently the longer the calibration will last.

5.3. Phase change

After analysing the quality of the algorithm in a stationary situation, we analyse how it reacts when a phase change is detected. For this experiments, we detect a phase change when the variation in the service rate is greater than a given threshold. In general, the greater the threshold value, the more the algorithm will be stable and less reactive (and vice versa). We experimentally found that a threshold of 40% is a good balance between stability and reactiveness on the given platform.

We consider the *denoiser* application with variable input resolutions [9]. Since the application exhibits different service rate and power consumption trends for different resolutions, when the input resolution changes, a phase change is detected and new models are computed. For this experiment, we used a 960x540 resolution input that, between 0 and 42 seconds and between 100 and 141 seconds, drops to 480x270. The bandwidth required (B) is 200 frames per second.

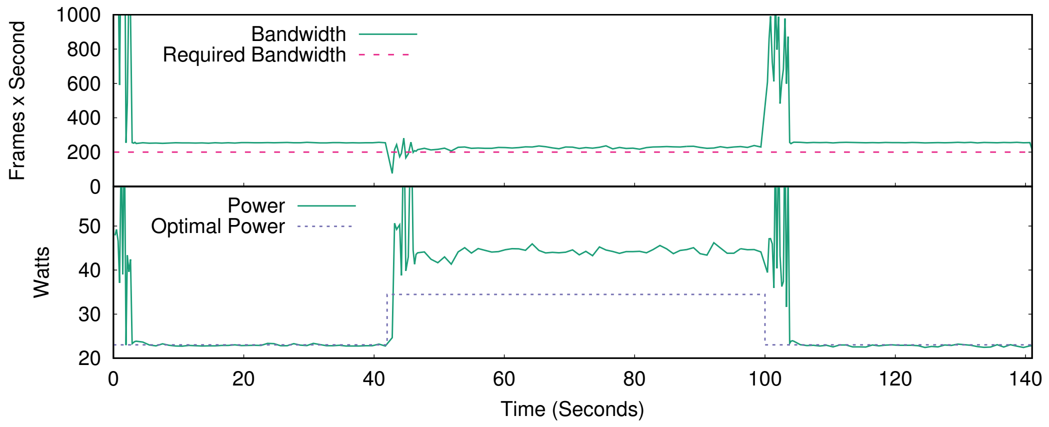

Fig. 7. Reaction of the algorithm to phase changes.

Figure 7 shows how the NORNIR algorithm changes the configuration of the application in order to satisfy the requirement. Moreover, as we can see from the bottom part of the graph, the configuration found is very close to the optimal one. Around 42 and 100 seconds, the algorithm detects that the application entered in a different phase, thus it starts a new calibration. As described in Section 3, while calibrating, the system cannot guarantee that the requirement is satisfied nor that it selects configurations close to the optimal one. Moreover, although the system succeeds in guaranteeing the user requirement, it may still select a configuration with sub-optimal power consumption (e.g., between 42 and 100 seconds). A similar behaviour to the one reported in Figure 7 occurs even when interferences are caused by external applications running on the same system.

5.4. Input bandwidth change

To analyse how the algorithm reacts to fluctuations in the arrival rate, we considered a network monitoring application [Danelutto et al. 2015]. This application analyses all the traffic travelling over a network, applying *Deep Packet Inspection* techniques to find possible security threats. For our experiments we used synthetic traffic data, while the arrival rates are those of a real backbone network[10]. For this experiment we used a dynamic requirement, i.e. the algorithm must be able to always keep $\mu \geq \frac{\lambda}{0.9}$. By doing so, we ensure that all the data received are processed and, at the same time, the system is not under-utilised since the utilisation is $\rho = \frac{\lambda}{\mu} \leq 0.9$. The application is executed

[9]Current video streaming protocols dynamically adjust the video resolution according to the available network bandwidth.
[10]http://www.caida.org/data/realtime/passive/?monitor=equinix-chicago-dirA, 24 hours of traffic between 03/01/2016 and 04/01/2016.

for 24 hours, and the results are shown in Figure 8. In the upper part of the figure, we

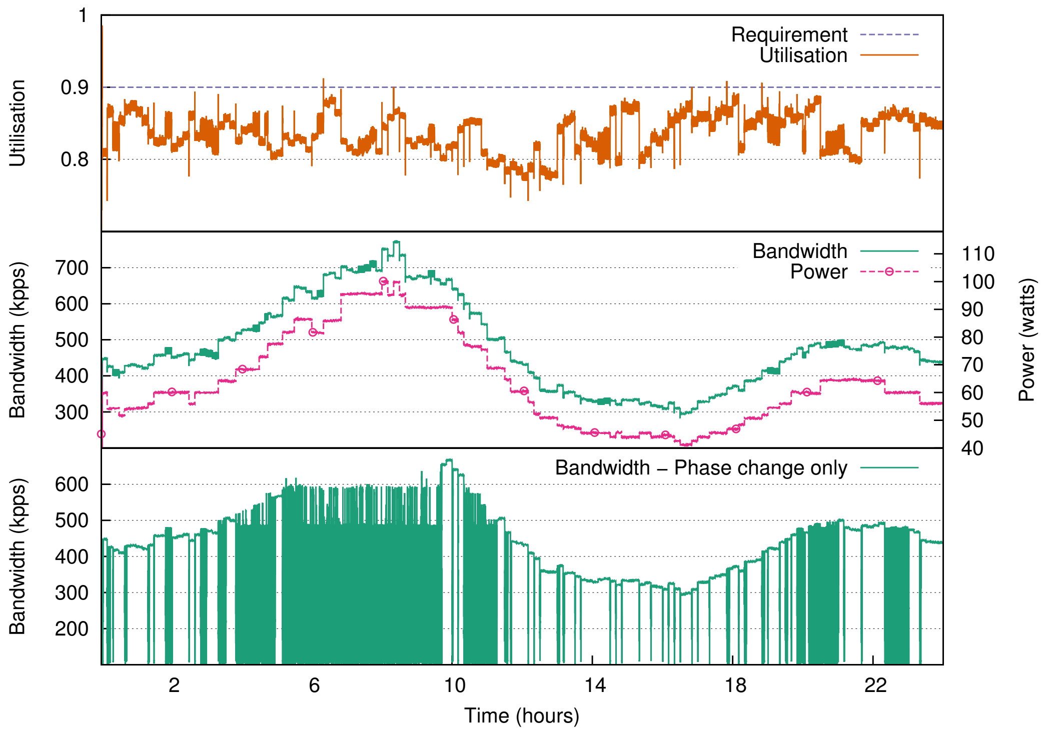

Fig. 8. Time behaviour of an application in presence of arrival rate fluctuations. Bandwidth is expressed in thousands of packets per second.

show that the algorithm is able to always satisfy the user requirement, independently from the arrival rate. In the central part we show the output bandwidth B (arrival rate λ is not shown since we were always able to keep $B = \lambda$) and the power consumption P, proving that the algorithm can dynamically adapt the amount of resources used by the runtime to changes in the arrival rate, optimising its power consumption.

Eventually, we evaluate one of the main advantages of our algorithm with respect to other existing online algorithms. In the bottom part of the figure, we show the behaviour of NORNIR when the distinction between the two types of dynamicity (i.e. phase change and arrival rate change) is not made, as common in many existing approaches [Mishra et al. 2015], [Shafik et al. 2015], [Marathe et al. 2015]. To test this case, we artificially disable the arrival rate change detection in the algorithm. In such case, each change in the arrival rate would appear as a phase change, thus triggering the computation of new prediction models. As we can see from the figure, this causes the system to be very unstable 5394 reconfigurations against the 75 performed by NORNIR, with a significant impact on the performance (-25.46%).

6. CONCLUSIONS AND FUTURE WORK

In this work we proposed NORNIR, a new algorithm for self-adaptation of parallel applications able to find a close to optimal [CORES, FREQUENCY, THREADS PLACEMENT] configuration that satisfies a user bound on performance and/or on power consumption. The algorithm does not leverage on any knowledge about previous applications executions, instead, it tries to derive on-the-fly prediction models for the power and the performance of the application. These models are used to select the best configuration satisfying user's requirements. The algorithm keeps monitoring the runtime to react either to changes in the input data interarrival rate and to intrinsic changes in the application behaviour. We also described a possible implementation of the algorithm over the FASTFLOW runtime system. We validated the quality of our proposal both with the help of simulations and with real executions over different applications. Moreover, we compared the results we obtained with those achieved by state of the art solutions, assessing the quality of our algorithm.

As a future work, we planned to extend the proposed algorithm to consider multiple applications, each one with its own requirement. This requires several challenges to be solved. Indeed, the algorithm currently assumes to have a complete control of the underlying machine (e.g. for frequency scaling and threads placement). Coordinating frequency management between different applications is an open problem and only few solutions exist [Ribic and Liu 2016]. In addition to that, calibration phases of different applications may interfere with each other, i.e. the calibration of an external application must be distinguished from phase changes and from workload pressure fluctuations. Moreover, multiple applications may have different requirements, incompatible with each other (e.g. one application may want to maximise performance while the other may want to minimise power consumption). Eventually, consider a system where only an application A is running. After its calibration, it derives a performance model M_A and selects an optimal configuration K_A. When an application B come into the system, it starts its calibration and obtain a performance model M_B. However, the model M_A was computed when A was the only application running. Accordingly, now that B is running, A needs to compute a new performance model M'_A. After that this model has been computed, A selects a new optimal configuration K'_A. However, application B computed its model M_B when the application A was in configuration K_A. Accordingly, B needs to compute a new model M'_B. This process may not converge or may need several iterations, especially when multiple applications with different requirements are present on the system. A subset of these challenges have been smartly solved in some existing works (e.g. application interference in [Delimitrou and Kozyrakis 2014], [Delimitrou and Kozyrakis 2013]). On the other hand, other important problems related to power constraints and shared frequency management have not been yet considered (to the best of our knowledge) in the context of multiple running applications.

REFERENCES

Ferdinando Alessi, Peter Thoman, Giorgis Georgakoudis, Thomas Fahringer, and Dimitrios S. Nikolopoulos. 2015. *OpenMP: Heterogenous Execution and Data Movements 11th Intl. Workshop on OpenMP, IWOMP 2015, Proc.* Springer, Chapter Application-Level Energy Awareness for OpenMP, 219–232.

Pedro Alonso, Manuel F. Dolz, Rafael Mayo, and Enrique S. Quintana-Ort. 2014. Modeling power and energy of the task-parallel Cholesky factorization on multicore processors. *Computer Science - Research and Development* 29, 2 (2014), 105–112.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58.

Arka A. Bhattacharya, David Culler, Aman Kansal, Sriram Govindan, and Sriram Sankar. The Need for Speed and Stability in Data Center Power Capping. In *Proc. of IGCC 2012*. IEEE Computer Society, 1–10.

A.P. Chandrakasan and R.W. Brodersen. 1995. Minimizing power consumption in digital CMOS circuits. *Proc. of the IEEE* 83, 4 (Apr 1995), 498–523.

Hao Chen, Can Hankendi, Michael C. Caramanis, and Ayse K. Coskun. 2013. Dynamic Server Power Capping for Enabling Data Center Participation in Power Markets. In *Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD '13)*. IEEE Press, Piscataway, NJ, USA, 122–129.

Ryan Cochran, Can Hankendi, Ayse Coskun, and Sherief Reda. 2011a. Identifying the optimal energy-efficient operating points of parallel workloads. (Nov. 2011), 608–615.

Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. 2011b. Pack & Cap: adaptive DVFS and thread packing under power caps. In *Proc. of the 44th Annual IEEE/ACM Intl. Symposium on Microarchitecture - MICRO-44 '11*. ACM Press, New York, New York, USA, 175.

M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. 2008a. Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems* 19, 10 (Oct 2008), 1396–1410.

Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. 2008b. Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores. In *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 250–259.

M. Danelutto, D. De Sensi, and M. Torquati. 2015. Energy Driven Adaptivity in Stream Parallel Computations. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Intl. Conf. on*. 103–110.

Marco Danelutto and Massimo Torquati. 2015. Structured Parallel Programming with "core" FastFlow. In *Central European Functional Programming School*. LNCS, Vol. 8606. Springer, 29–75.

Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. 2011. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *Proc. of the 8th ACM Intl. Conf. on Autonomic Computing (ICAC '11)*. ACM, New York, NY, USA, 31–40.

Daniele De Sensi. 2016. Predicting performance and power consumption of parallel applications. In *24th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing, PDP 2016*.

Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 77–88.

Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014), 127–144.

Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. 2008. A helper thread based EDP reduction scheme for adapting application execution in CMPs. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE Intl. Symposium on*. 1–14.

Pedro Domingos. 2012. A Few Useful Things to Know About Machine Learning. *Commun. ACM* 55, 10 (Oct. 2012), 78–87.

Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2014. Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees. In *Proc. of the 36th Intl. Conf. on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 299–310.

Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, Jeffrey Kephart, and Charles Lefurgy. 2009. Power Capping Via Forced Idleness. In *Proc. of Workshop on Energy-Efficient Design (WEED 09) Austin, Texas*.

Larry D. Gray, Anil Kumar, and Harry H. Li. 2008. *Workload Characterization of the SPECpower_ssj2008 Benchmark*. Springer Berlin Heidelberg, Berlin, Heidelberg, 262–282.

Neil J. Gunther. 2006. *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Perform. Eval. Rev.* 40, 3 (Jan. 2012), 13–17.

Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic Knobs for Responsive Power-aware Computing. *SIGPLAN Not.* 46, 3 (2011), 199–212.

Chung-hsing Hsu and Wu-chun Feng. 2005. A Power-Aware Run-Time System for High-Performance Computing. In *Proc. of the ACM/IEEE SC 2005 Conf.* 1–1.

N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. 2003. Leakage current: Moore's law meets static power. *Computer* 36, 12 (Dec 2003), 68–75.

Sang-Jeong Lee, Hae-Kag Lee, and Pen-Chung Yew. 2007. Runtime Performance Projection Model for Dynamic Power Management. In *Proc. of the 12th Asia-Pacific Conf. on Advances in Computer Systems Architecture (ACSAC'07)*. Springer-Verlag, Berlin, Heidelberg, 186–197.

Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. 2008. Power Capping: A Prelude to Power Shifting. *Cluster Computing* 11, 2 (June 2008), 183–195.

Jian Li and José F. Martínez. 2006. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. *Proc. of Intl. Symposium on High-Performance Computer Architecture* (2006), 77–87.

M. Maggio, H. Hoffmann, M.D. Santambrogio, A. Agarwal, and A. Leva. 2010. Controlling software applications via resource allocation within the heartbeats framework. In *Decision and Control (CDC), 2010 49th IEEE Conf. on*. 3736–3741.

Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. 2015. A Run-Time System for Power-Constrained HPC Applications. In *Proc. of High Performance Computing - 30th Intl. Conf., ISC High Performance 2015*. 394–408.

Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. 2015. A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints. *ACM SIGARCH Computer Architecture News* 43, 1 (March 2015), 267–281.

Douglas C. Montgomery and Elizabeth Peck. *Introduction to linear regression analysis*. John Wiley & sons.

Priya Nagpurkar, Chandra Krintz, Michael Hind, Peter F. Sweeney, and V. T. Rajan. 2006. Online Phase Detection Algorithms. In *Proc. of the Intl. Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, Washington, DC, USA, 111–123.

Paula Petrica, Adam M. Izraelevitz, David H. Albonesi, and Christine A. Shoemaker. 2013. Flicker: a dynamically adaptive architecture for power limited multicore systems. *ACM SIGARCH Computer Architecture News* 41, 3 (July 2013), 13.

Allan K. Porterfield, Stephen L. Olivier, Sridutt Bhalachandra, and Jan F. Prins. 2013. Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs. In *Proc. of IPDPSW 2013*. IEEE, 884–891.

Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. 2011. Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring. In *Proc. of the 2011 IEEE Intl. Symposium on Workload Characterization (IISWC '11)*. IEEE Computer Society, Washington, DC, USA, 116–125.

Haris Ribic and Yu David Liu. 2016. AEQUITAS: Coordinated Energy Management Across Parallel Applications. In *Proc. of ICS 2016 (ICS '16)*. ACM, New York, NY, USA, Article 4, 12 pages.

Barry Rountree, Dong H. Ahn, Bronis R. de Supinski, David K. Lowenthal, and Martin Schulz. 2012. 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, 947–953.

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

A. Sembrant, D. Eklov, and E. Hagersten. 2011. Efficient software-based online phase classification. In *Workload Characterization (IISWC), 2011 IEEE Intl. Symposium on*. 104–115.

Rishad A. Shafik, Anup Das, Sheng Yang, Geoff Merrett, and Bashir M. Al-Hashimi. 2015. Adaptive Energy Minimization of OpenMP Parallel Applications on Many-Core Systems. In *Proc. of the 6th PARMA-DITAM workshop '15*. ACM Press, New York, New York, USA, 19–24.

Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2013. Holistic run-time parallelism management for time and energy efficiency. In *Proc. of the 27th Intl. ACM Conf. on Intl. Conf. on supercomputing - ICS '13*. ACM Press, New York, New York, USA, 337.

M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. 2008. Feedback-driven threading. In *Proc. of the 13th Intl. Conf. on Architectural support for programming languages and operating systems - ASPLOS XIII*, Vol. 42. ACM Press, New York, New York, USA, 277.

Alexander Thomasian and Paul F Bay. 1986. Analytic queueing network models for parallel processing of task systems. *Computers, IEEE Transactions on* 100, 12 (1986), 1045–1054.

Ehsan Totoni, Nikhil Jain, and Laxmikant V. Kalé. 2015. Power Management of Extreme-Scale Networks with On/Off Links in Runtime Systems. *TOPC* 1, 2 (2015), 16.

Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos Chalios, Christos D Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2015. A Programming Model and Runtime System for Significance-aware Energy-efficient Computing. *SIGPLAN Not.* 50, 8 (2015), 275–276.

Wei Wang, A. Porterfield, J. Cavazos, and S. Bhalachandra. 2015. Using Per-Loop CPU Clock Modulation for Energy Efficiency in OpenMP Applications. In *Proc. of ICPP 2015*. 629–638.

Fen Xie, Margaret Martonosi, and Sharad Malik. 2005. Efficient Behavior-driven Runtime Dynamic Voltage Scaling Policies. In *Proc. of the 3rd IEEE/ACM/IFIP Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS '05)*. ACM, New York, NY, USA, 105–110.

Albert Y. Zomaya and Young Choon Lee. 2012. *Energy Efficient Distributed Computing Systems* (1st ed.). Wiley-IEEE Computer Society Pr.

Received June 2016; revised August 2016; accepted September 2016