Application-Aware Power Capping using Nornir[*]

Daniele De Sensi[1][0000−0002−7244−639X] and Marco Danelutto[1]

Computer Science Department, University of Pisa, Pisa, Italy
`http://pages.di.unipi.it/{desensi,danelutto}`
`{desensi, marcod}@di.unipi.it`

Abstract. Power consumption of IT infrastructure is a major concern for data centre operators. Since data centres power supply is usually dimensioned for an average-case scenario, uncorrelated and simultaneous power spikes in multiple servers could lead to catastrophic effects such as power outages. To avoid such situations, power capping solutions are usually put in place by data centre operators, to control power consumption of individual server and to avoid the datacenter exceeding safe operational limits. However, most power capping solutions rely on Dynamic Voltage and Frequency Scaling (DVFS), which is not always able to guarantee the power cap specified by the user, especially for low power budget values. In this work, we propose a power-capping algorithm that uses a combination of DVFS and Thread Packing. We implement this algorithm in the NORNIR framework and we validate it on some real applications by comparing it to the Intel RAPL power capping algorithm and another state of the art power capping algorithm.

Keywords: Power Capping · RAPL · Self-Aware Computing · Green Computing

1 Introduction

Power consumption management is becoming a critical factor in designing applications and computing systems. In data centres, the energy cost is quickly going to overcome the cost of the physical system itself [4]. Moreover, besides economic considerations, power consumption has a considerable impact on the environment, since during 2010 the CO_2 emissions of U.S. data centres were on par with those of an entire country like Argentina or Netherlands [19].

Traditionally, to avoid possible electric surges, data centre operators have over-provisioned data centre power, considering a worst-case power consumption [15]. Albeit this ensures reliability with high confidence, it is wasteful in terms of power infrastructure utilization. To improve efficiency, researchers are investigating the possibility to over-subscribe data centre power [15, 16, 20]. Namely, the data centre power demand could intentionally be allowed to exceed the power supply, under the assumption that correlated spikes in servers' power

[*] This work has been partially supported by Univ. of Pisa PRA_2018_66 DECLware: Declarative methodologies for designing and deploying applications.

consumption are infrequent. However, this exposes data centers to the risk of power outages, caused by unpredictable power spikes (e.g. due to an increase in the power consumption of more servers at the same time). Such an event would have catastrophic effects since it would lead to degradation in the final user experience or service outages. For these reasons, to achieve power safety and to avoid having under-utilized power provisioning, *power capping* techniques have been recently proposed [11, 22, 21, 18]. These techniques monitor the data centre power consumption and, when it gets close to the available capacity, request the servers to reduce their power consumption, usually by applying Dynamic Voltage and Frequency Scaling (DVFS) [6].

One of the most commonly used techniques is Intel RAPL power capping [23]. However, it can only operate in a predefined range according to the processor specifications, and any value outside this range will be ignored. However, by extending the range of values enforceable by a power capping mechanism it would be possible to better distribute the power budget on the available servers, for example by setting low budgets for servers running non-critical applications and by letting the computing nodes running important applications run without any power cap. In this work, we address this issue by proposing a power capping algorithm which combines DVFS and Thread Packing. Thread packing [8] is a technique which forces N threads to run on a number of cores C, with $C \leq N$, thus allowing the operating system to put some cores in sleep states. Moreover, we provide a working implementation of this algorithm by adding it to the Nornir framework, which would allow us to apply power capping to a specific application without any need to change the application code.

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

- We propose a power capping algorithm which, given a power cap, can find the most performing configuration in terms of clock frequency and the number of cores used.
- We implement this algorithm inside the Nornir framework.
- We validate the algorithm by comparing it against Intel RAPL power capping [11] and another state of the art algorithm [13], showing improvement in the performance of the selected configuration up to 2X.

The rest of the paper is organized as follows. Section 2 describes some related works, highlighting the strengths and weaknesses of each of them. Then, in Section 3 we provide some background by describing the Nornir framework. The design and implementation of the algorithm are described in Section 4 and it is then evaluated in Section 5. Eventually, Section 6 concludes and outlines some possible future directions for this work.

2 Related Work

Several works proposed different power capping algorithms and techniques. The most commonly used solution is Intel RAPL power capping [11], which is provided as a tool on Intel architectures. The tool dynamically scales the clock frequency and the voltage of the cores in order to enforce the power budget required by the user. However, as shown in the motivating example in Section 1, by only using DVFS it is not possible to decrease the power consumption below a certain threshold.

However, some works propose solutions using DVFS in conjunction with other techniques. For example, Conoci et al. [9] propose a power capping algorithm that uses DVFS and *concurrency throttling* (i.e. dynamically changing the number of threads at runtime). However, threads can be dynamically removed and added only for applications based on the thread pool model, thus limiting the applicability of the approach. On the contrary, our approach does not assume any particular application structure, since it relies on *thread packing*.

Other works also use *thread packing* [8]. However, differently from our approach, they require a training phase to be performed offline, before running the application. During the training phase, data about different applications will be collected to build a model to predict the performance and power consumption of the application in different configurations. Our algorithm relies on the opposite approach, by not requiring any training and by taking decisions only based on what is observed during application execution.

Some existing solutions do not require an offline training phase and use DVFS together with *thread packing* or *concurrency throttling* [13, 2], similarly to what is done in this paper. However, such solutions either require to modify the source code of the application or are tied to some specific programming model such as OpenMP. On the contrary, our approach does not make any assumption on the application and does not require any modification to existing applications.

Eventually, some works propose techniques to coordinate power capping at the datacenter level [22]. However, since they rely on Intel RAPL for power capping, such solutions are still affected by the problem outlined in Section 1.

3 The Nornir Framework

Nornir [12] is a customizable framework which can be used to add power-aware capabilities to applications. On one side, Nornir can be used to enforce power consumption and performance requirements to applications. For example, users could ask Nornir to dynamically change the number of resources used by a video processing application so that the application will consume no more than 60 Watts but, at the same time, it will process at least 20 frames per second. On the other side, it can be customized by researchers by adding new algorithms for selecting the proper amount of resources given the user constraints.

To monitor the application and to apply some decisions (e.g. dynamically remove threads from the application), Nornir needs to be interfaced with the application. This can be done in different ways:

1. The user could implement a parallel application from scratch, by using the programming API provided by Nornir
2. Nornir can natively interact with some parallel runtimes (OpenMP and FastFlow [1] are currently supported). If the application uses one of these

runtimes, Nornir can interact with the application with minimal modification to the application code.

3. Otherwise (e.g. for applications using *pthreads*), the user could insert a couple of instrumentation calls in the application. These calls will monitor the performance of the application and will send these data to Nornir, which will use it to decide how many resources to allocate to the application.
4. Eventually, if the user can not modify the application, Nornir can still monitor it by relying on hardware performance counters (e.g. number of instructions executed per time unit). Despite this solution have been previously described [12] (denoted as *black-box*), its efficacy has never been evaluated. It is worth mentioning that in this case, because Nornir doesn't explicitly interact with the application, performance requirements can only be expressed in terms of instructions executed per time unit.

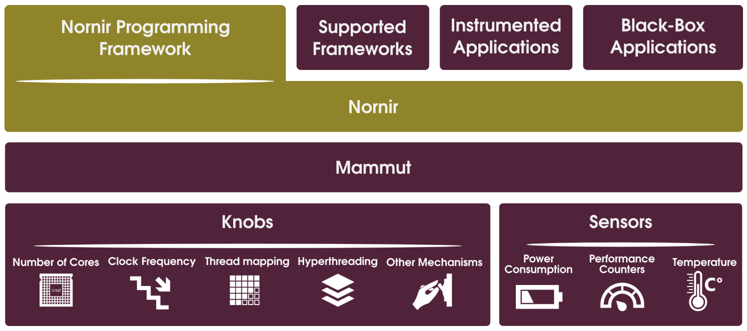


Fig. 1: Nornir architecture.

Nornir architecture is depicted in Figure 1. Since in this work we would like to provide a solution which could be better than Intel RAPL power capping while not being worst in terms of user effort, we will focus on the case where the user can not modify the application, forcing Nornir to monitor the application only through hardware performance counters.

Nornir works by following a classical *Monitor-Analyze-Plan-Execute* (MAPE) autonomic loop where, at fixed timesteps, it monitors the current performance and power consumption of the application in its current configuration. Based on this knowledge it decides if and how to change the number of resources allocated to the application, by using DVFS to scale the clock frequency and thread packing to change the number of used cores. In the *Monitor* phase, instructions per second and power consumption are collected by using Mammut library [14]. Among others, this library is also used by Nornir in the *Execute* phase to apply DVFS and thread packing.

To set a specific power cap, it is sufficient to use an executable file provided by Nornir, which takes as arguments the process identifier (*pid*) of the process we would like to control and the value of the power cap we would like to set. Note that at the moment it is only possible to control one application at a time. In the future, we will extend this approach to control multiple concurrent applications.

4 Algorithm Design

At each timestep, when Nornir executes the *Analyze* and *Plan* phases, our algorithm is invoked. Based on the information gathered in the *Monitor* phase, we must decide which frequency f and how many cores n we would like our application to use. Since both f and n have an impact on both the performance and the power consumption, our algorithm must estimate how performance and power consumption change when f and n are increased or decreased.

Concerning the performance modeling, since we are monitoring the application by using hardware performance counters, performance in our case are represented by the number of instructions executed per time unit. We denote with $I(n, f)$ the number of instruction executed for a given number of cores and clock frequency, with $\overline{n}$ the number of cores currently used and with $\overline{f}$ the clock frequency currently used. Taking inspiration from the performance model presented in [10], we assume $I(n, f)$ to scale linearly with both n and f, i.e.

$$I(n, f) = I(\overline{n}, \overline{f}) \cdot \frac{n \cdot f}{\overline{n} \cdot \overline{f}} \tag{1}$$

Accordingly, given the current measurement $I(\overline{n}, \overline{f})$ we can estimate the performance of any other configuration by assuming that the performance will change proportionally to the changes in the number of used cores n and the clock frequency f. It is worth noting that we are assuming that the application linearly scales, i.e. by doubling the number of cores we would double the instructions executed per time unit. Of course, this is not always the case and this approximation is more severe as larger is the distance between the current configuration and the predicted one. However, as we will see in Section 5, this is not an issue in practice since, even if the algorithm selects a wrong configuration, another decision will be taken in the following time step. As a consequence, after a small number of steps, the algorithm will get closer to the correct configuration.

Concerning the power consumption $P(n, f)$, it is composed by a static quantity (which does not depend on n and f) and a dynamic quantity [7, 3, 17][1]. Since the dynamic power is also dependent on the supply voltage v, we must include it into our equation, i.e.:

$$P(n, f, v) = P_{static} + P_{dyn}(n, f, v) = P_{static} + \alpha \cdot C \cdot n \cdot v^2 \cdot f \tag{2}$$

C and α represent the capacitance of the circuit and the activity factor (i.e. the fraction of the gates which are active, on average). However, the voltage v

[1] Actually, static power could change when changing the frequency f. However, this is a common approximation and, as we will see in Section 5, it does not alter the accuracy of our algorithm.

usually depends on the frequency f and the number of cores n. Accordingly, we can rewrite the equation as:

$$P(n,f) = P_{static} + P_{dyn} = P_{static} + \alpha \cdot C \cdot n \cdot V(n,f)^2 \cdot f \tag{3}$$

where $V(n,f)$ is a function which returns the voltage associated to a specific n and f. This function, in tabular form, is computed and stored by Nornir when it is first installed on the system by using Mammut [14]. On our system, Mammut computes the voltage by accessing the `PERF_STATUS[47:32]` MSR register. P_{static} is constant, and it is computed and stored by Nornir when it is installed on the system, by measuring the average idle power consumption on a one minute interval. Accordingly, we only need to estimate $P_{dynamic}$. Because n, f and $V(n,f)$ are known for all the configurations, we only need to estimate $\alpha \cdot C$. This can be done starting from Equation 3 by considering the power consumption in the current configuration:

$$\alpha \cdot C = \frac{P(\overline{n},\overline{f}) - P_{static}}{\overline{n} \cdot V(\overline{n},\overline{f})^2 \cdot \overline{f}} \tag{4}$$

Because all the needed quantities are known, we can estimate the power consumption in any configuration as:

$$P(n,f) = P_{static} + (P(\overline{n},\overline{f}) - P_{static}) \cdot \frac{n \cdot V(n,f)^2 \cdot f}{\overline{n} \cdot V(\overline{n},\overline{f})^2 \cdot \overline{f}} \tag{5}$$

It is worth noting that this approach does not require any application characterization. Nornir will monitor the application throughout its execution, selecting the optimal number of cores and frequency according to the predictions made by the performance and the power consumption models.

5 Experimental Evaluation

We validate our algorithm by comparing it against two algorithms: i) Intel *RAPL* power capping, which applies DVFS and clock modulation to control the power consumption of the system; ii) *Online Learning* [13], which uses an online learning approach where a part of the application execution is used to collect data about different configuration and to build a prediction model, which will be used to select the optimal configuration. This algorithm is one of those already provided by Nornir.

We selected the *blackscholes*, *bodytrack* and *streamcluster* benchmarks from the PARSEC benchmark suite [5] . All the experiments have been executed on a Dual-socket NUMA machine with two Intel Xeon E5-2695 Ivy Bridge CPUs running at 2.40GHz featuring 24 hyper-threaded cores (12 per socket). Each hyper-threaded core has 32KB private L1, 256KB private L2 and 30MB of L3 shared with the cores on the same socket. The machine has 64GB of DDR3 RAM and a Thermal Design Power (TDP) of 230 Watts. We did not use the hyper-threading, and the applications used at most 24 cores in our experiments. Due to hardware limitations, on this machine, it is not possible to set the frequency of each core individually. However, this is not a problem since our prediction models assume that the frequency of all the cores will be the same. The software environment consists of Linux 3.14.49 x86_64 shipped with CentOS 7.1 and `gcc` version `4.8.5`. When using Intel RAPL power capping, we split the power budget evenly among the two packages (CPU). For example, when setting a 100 Watts power cap, we will set a 50 Watts power cap on each CPU. We express the power budget as a percentage of the TDP. For example, a power budget of 10% represents a 23 Watts power budget. We consider up to a 50% power budget because we did not observe any significant difference above that level. For all the approaches, we enforce the power cap on a window of one second. The static power was $\sim$ 37 Watts on the machine we used for our experiments. All the power consumption data presented include static power. According to the specifications of this processor, power capping values cannot be lower than $\sim$ 64 Watts for each package. However, we experimentally found that the actual limit which can be reached by Intel RAPL power capping is around $\sim$ 30 Watts for each package.

We evaluated each algorithm over each application for different power budgets, by analyzing the following two metrics:

Violation Let us suppose that the application runs for s seconds, that the power cap required by the user was c Watts and that the measured power consumption at a given time t is $P(t)$. We also define with V the set of samples t such that $P(t) > c$, i.e. the set of samples where the power cap was violated. Then, this metric is defined as:

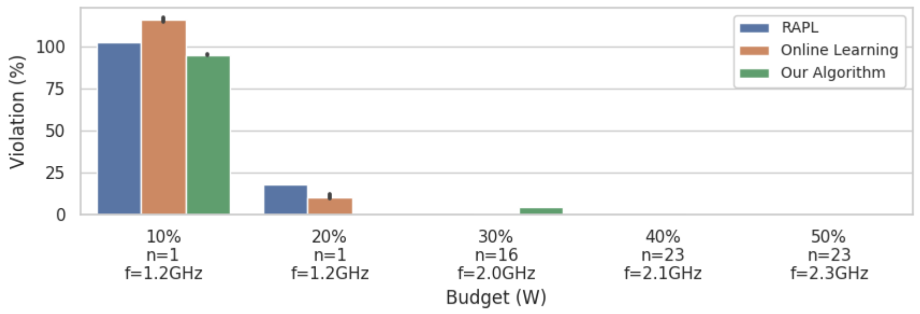
$$\frac{\sum_{t \in V} (P(t) - c)}{s}$$

This metric includes both the number and the amplitude of the power budget violations and represents the average violation of the power cap. A lower value implies a better algorithm.

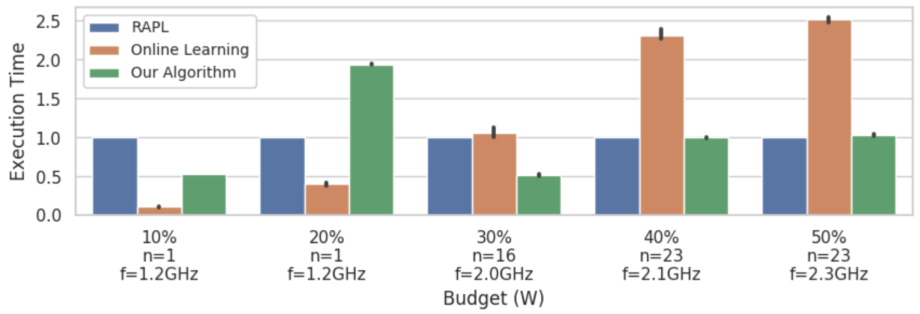
Execution Time The performance of the application, expressed as the execution time normalized to the execution time when using Intel RAPL power capping. A lower value implies a better algorithm. This metric will be shown only for experiments where RAPL correctly enforce the power cap. Indeed, when the power budget is exceeded performance would be higher than the performance achieved by solutions which properly enforce the power cap, and the comparison in such cases would not be fair.

For each of the following plots we show on the x-axis the maximum power budget we set for the application and on the y-axis the value for the metric. On the x-axis label we have on the three lines:

1. The power budget expressed as a percentage of the TDP.
2. The average number of cores set by our algorithm.
3. The clock frequency set by our algorithm.



(a) Budget violations (higher = better).

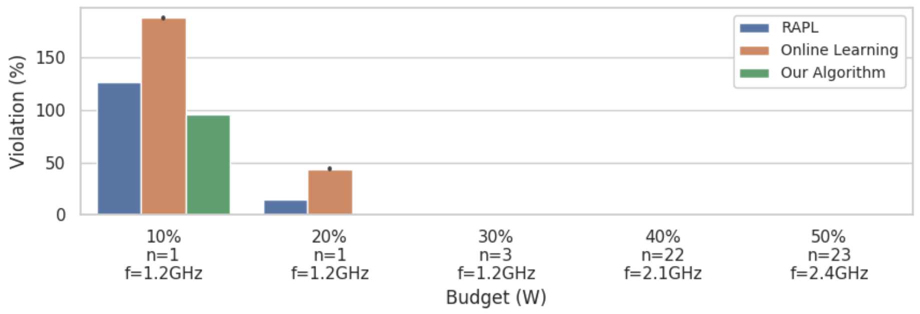


(b) Execution time (lower = better).

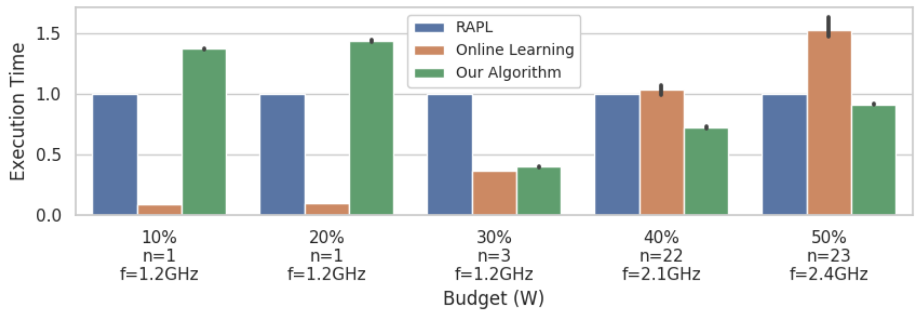
Fig. 2: Analysis of different algorithms on the *blackscholes* application.

The black vertical bar on the top of the histograms represents the 95% confidence interval from the mean.

We report in Figure 2 the analysis for the *blackscholes* application. By analyzing the plots, we see how *RAPL* fails in enforcing power caps with a budget lower or equal than 20%. Whereas no algorithms can reach the 10% power budget, our algorithm correctly enforces the 20% power budget, because it reduces the number of cores allocated to the application to one, while with *RAPL* all the available cores are used. By observing the performance results, our algorithm is always characterized by the best performance, except for the 10% and 20% cases, because *RAPL* and *Online Learning* violate more often the available power budget, using more resources and obtaining higher performance. For the 30% case our algorithm outperforms the *online learning* and *RAPL* algorithms, finding configurations that, while still satisfying the required power cap, are characterized by a higher performance (up to 2x). The reason why our algorithm performs better than the *online learning* one is that the latter needs more time to find a suitable configuration, due to the training phase it needs to perform to gather data about different configurations. Since during the training phase some low-performing configurations may be visited, this increases the execution time. Moreover, the algorithm needs to be trained again for different



(a) Budget violations (higher = better).



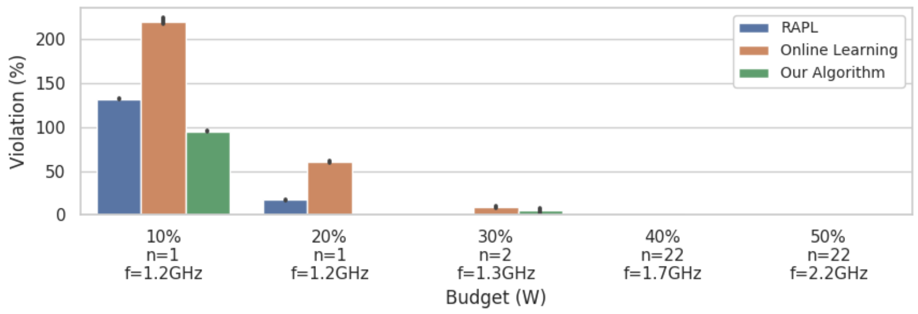
(b) Execution time (lower = better).

Fig. 3: Analysis of different algorithms on the *bodytrack* application.

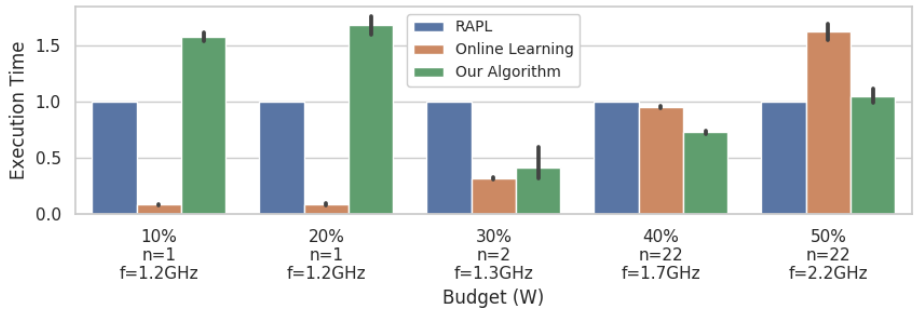
application phases, introducing additional overhead. For power budgets higher than 30% all the algorithm can properly enforce the power cap.

Similar results have been obtained for *bodytrack*, as shown in Figure 3. In this case, the performance gap between our algorithm and the *online learning* one are even more evident, with a speedup higher than $\sim 2X$ when the power cap is set to 30%. Moreover, our solution can find configurations which are more performing than those selected by *RAPL*, even for higher power budgets (40% and 50%). This happens for the same reason why *RAPL* fails in enforcing low power budgets, i.e. since it only uses DVFS, the set of choices it can make are much more limited compared to our algorithm.

Eventually, Figure 4 reports the results for the *streamcluster* application. Even in this case, the results reflect what we observed for the other two applications, with our algorithm being able to enforce 20% power caps, and providing more than $\sim 2X$ performance improvement on higher power caps compared to *RAPL*.



(a) Budget violations (higher = better).



(b) Execution time (lower = better).

Fig. 4: Analysis of different algorithms on the *streamcluster* application.

6 Conclusions and Future Work

In this work, we presented a power capping algorithm which used DVFS and thread packing to extend the range of reachable power caps compared to RAPL. We implemented this algorithm in the NORNIR framework, and we used its ability to control applications to test our algorithm. We then compared our algorithm with RAPL and with another state of the art approach, showing that it can satisfy the required power cap even when RAPL is not able to do so. Moreover, even when both algorithms correctly enforce the power budget required, there are cases where our algorithm can find configurations characterized by better performance, in some cases more than $\sim 2X$ more performing than those found by the other two algorithms.

In the future, we would like to extend this work for controlling multiple concurrent applications, possibly by having different power budgets for different applications according to their importance/priority. Moreover, we would like to extend the algorithm to also consider other control mechanisms such as Dynamic Clock Modulation (DCM) or DVFS for memory and *uncore* components.

References

1. Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*, chapter 13. 2014.
2. Ferdinando Alessi, Peter Thoman, Giorgis Georgakoudis, Thomas Fahringer, and Dimitrios S Nikolopoulos. Application-level Energy Awareness for OpenMP. In *11th International Workshop on OpenMP, IWOMP*, pages 219–232, Cham, 2015.
3. Pedro Alonso, Manuel F. Dolz, Rafael Mayo, and Enrique S. Quintana-Ort. Modeling power and energy of the task-parallel cholesky factorization on multicore processors. *Computer Science - Research and Development*, 29(2):105–112, 2014.
4. L.A Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec 2007.
5. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int. Conf. on Parallel Architectures and Compilation Techniques*, 2008.
6. T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A dynamic voltage scaled microprocessor system. In *ISSCC 2000*, pages 294–295, Feb 2000.
7. A.P. Chandrakasan and R.W. Brodersen. Minimizing power consumption in digital cmos circuits. *Proc. of the IEEE*, 83(4):498–523, Apr 1995.
8. Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & Cap: adaptive DVFS and thread packing under power caps. In *Proc. of the 44th Annual IEEE/ACM Intl. Symposium on Microarchitecture*, December 2011.
9. Stefano Conoci, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Adaptive performance optimization under power constraint in multi-thread applications with diverse scalability. In *Proceedings of ICPE '18*, pages 16–27, 2018.
10. M. Danelutto, D. De Sensi, and M. Torquati. Energy driven adaptivity in stream parallel computations. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Intl. Conf. on*, Turku, Finland, March 2015. IEEE.
11. H David, E Gorbatov, U R Hanebutte, R Khanna, and C Le. RAPL: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, 2010.
12. D. De Sensi, T. De Matteis, and M. Danelutto. Simplifying self-adaptive and power-aware computing with nornir. *Future Generation Computer Systems*, 2018.
13. D. De Sensi, M. Torquati, and M. Danelutto. A reconfiguration algorithm for power-aware parallel applications. *ACM TACO*, 13(4), 2016.
14. D. De Sensi, M. Torquati, and M. Danelutto. Mammut: High-level management of system knobs and sensors. *SoftwareX*, 6:150 – 154, 2017.
15. Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. *SIGARCH Comput. Archit. News*, June 2007.
16. Xing Fu, Xiaorui Wang, and Charles Lefurgy. How much power oversubscription is safe and allowed in data centers. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, pages 21–30, 2011.
17. N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, Dec 2003.
18. Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Power capping: A prelude to power shifting. *Cluster Computing*, 11(2):183–195, June 2008.
19. Edward J. Lucente. The coming "c" change in data centers, 2010. `http://www.hpcwire.com/2010/06/15/the_coming_c_change_in_datacenters/`.

20. M. Maiterth, G. Koenig, K. Pedretti, S. Jana, N. Bates, A. Borghesi, D. Montoya, A. Bartolini, and M. Puzovic. Energy and power aware job scheduling and resource management: Global survey initial analysis. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 685–693, May 2018.
21. X. Wang, M. Chen, C. Lefurgy, and T. W. Keller. Ship: A scalable hierarchical power control architecture for large-scale data centers. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):168–176, Jan 2012.
22. Q. Wu, Q. Deng, L. Ganesh, C. H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song. Dynamo: Facebook's data center-wide power management system. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 469–480, Seoul, Korea, June 2016.
23. Huazhe Zhang and Henry Hoffmann. A quantitative evaluation of the rapl power control system. 2014.