Transparent Autonomicity for OpenMP Applications*

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

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

Abstract. One of the key needs of an autonomic computing system is the ability to monitor the application performance with minimal intrusiveness and performance overhead. Several solutions have been proposed, differing in terms of effort required by the application programmers to add autonomic capabilities to their applications. In this work we extend the NORNIR autonomic framework, allowing it to transparently monitor OpenMP applications thanks to the novel OpenMP Tools (OMPT) API. By using this interface, we are able to transparently transfer performance monitoring information from the application to the NORNIR framework. This does not require any manual intervention by the programmer, which can seamlessly control an already existing application, enforcing any performance and/or power consumption requirement. We evaluate our approach on some real applications from the PARSEC and NAS benchmarks, showing that our solution introduces a negligible performance overhead, while being able to correctly control applications' performance and power consumption.

Keywords: Power-Aware Computing · Autonomic Computing · OpenMP · Power Capping

1 Introduction

Adding autonomic capabilities to applications is an important feature of modern computing systems. Indeed, being able to automatically tune the application according to the user requirements would allow an optimal usage of the computing resources, with a consequent reduction of their power consumption. Autonomic capabilities are usually added to applications by having a separate entity (a *manager*) which periodically monitors the application and decides the action to take (e.g. reduce the resources allocated to the application) according to some requirements specified by the user. Such requirements can be usually expressed in terms of performance, power consumption, reliability, and others.

For performance monitoring purposes, interactions between the autonomic manager and the application can be implemented in several ways. The simplest solution would be to modify the application inserting some instrumentation

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

calls, which would collect the performance of the application and communicate this information to the autonomic manager, for example by using the *Heartbeat API* [15] or the NORNIR framework [9]. However, it is not always possible to modify the source code of the application, and this additional effort could discourage application programmers, limiting the adoption of such autonomic tools. On the other hand, other solutions monitor application performances without requiring any modification to the application source code. For example, this can be implemented by modifying the application binary to add instrumentation calls, either by using dynamic instrumentation tools like *PIN* [14] or by using static istrumentation tools such as *Maqao* [6] or *Dyninst* [7]. Alternatively, application performance may be inferred by analyzing performance counters (such as the number of instructions executed per time unit). However, by using such approach it would be difficult for the user to relate this performance information to the actual application performance (for example in terms of number of stream elements processed per time unit). Eventually, a last class of solutions modifies neither the application source code nor its binary, while still being able to monitor real application performance. These solutions can be used on applications implemented with specific programming frameworks, and interact with the runtime used by the application [10, 18], for example by intercepting some runtime calls.

In this work we will focus on this last class of solutions, by extending the NORNIR autonomic framework, allowing it to transparently interact with OpenMP applications. We will analyze our solution on different applications from the PARSEC [8] and NAS [5] benchmarks, showing that our implementation introduces a negligible performance overhead, while at the same time allowing the user to set arbitrary performance and power consumption requirements on such applications.

The rest of this paper is structured as follows. Section 2 briefly describes some existing works addressing autonomicity in OpenMP applications. In Section 3 we provide some background about the NORNIR framework and the OMPT API, which will be used to intercept OpenMP calls. In Section 4 we will describe the design and implementation of our solution and in Section 5 we will perform the experimental evaluation. Eventually, Section 6 concludes this work and outlines possible future developments.

2 Related Work

Different works deal with autonomic solutions for controlling performance and power consumption of applications, according to user requirements. In this section we will focus on the existing works targeting OpenMP applications.

Li et al. [13] target hybrid MPI/OpenMP applications, proposing an algorithm which applies Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Concurrency Throttling (DCT) to improve the energy efficiency of such applications. However, manual instrumentation by the programmer is required,

and no explicit performance and/or power consumption requirements can be specified by the user.

Other works [3, 17] propose extensions to the OpenMP annotations, to express explicit requirements in terms of power consumption, energy, or performance. Although such approaches are more expressive than the one presented in this work, they require to modify and recompile the application source code.

Wang et al. [18] apply clock modulation and DCT to OpenMP applications to reduce their energy consumption. On the other hand, in our work we interface OpenMP applications to the Nornir framework, allowing to enforce arbitrary constraints in terms of power consumption and performance, by using not only DCT and clock modulation but also DVFS and other mechanisms provided by the Nornir framework. Moreover, whereas in the work by Want et al. [18] the selection of the optimal concurrency level is done through a complete exploration of the search space, by using Nornir different algorithms can be applied to avoid such full exploration, thus reducing the time required to find the optimal resources configuration.

In addition to the aforementioned limitations, all the described approaches are implemented ad-hoc and do not rely on any general purpose autonomic framework. On the contrary, our approach relies on Nornir, extending the perks of the framework (e.g. the possibility to easily implement new autonomic algorithm) to any OpenMP application.

3 Background

In this section we provide some background about the Nornir framework and the OMPT API.

3.1 Nornir

Nornir[1] [9] is a framework for power-aware computing, providing the possibility to control performance and power consumption of applications running on shared memory multicore machines. Nornir provides a set of algorithms to control performance and power consumption of applications, in order to enforce requirements specified by the user. Internally, Nornir abstracts many low-level aspects related to interaction with both the underlying hardware and the application, and it can be easily customized by adding new control algorithms. Nornir acts according to the *Monitor, Analyze, Plan, Execute* (MAPE) loop. At each iteration of the MAPE loop (also known as *control step*), the application performance and power consumption is monitored, then appropriate decisions based on these observations are taken, and eventually these decisions are applied in the *Execute* phase. The MAPE loop is executed by a *manager* entity, which is executed as a separate thread/process.

[1] `https://github.com/DanieleDeSensi/nornir`

To perform the *Monitor* and *Execute* phases, the NORNIR *manager* needs to interact both with the machine it is running on, but also with the application it is controlling. To interact with the underlying hardware, NORNIR relies on the MAMMUT library [11], which abstracts in an object-oriented fashion the available hardware control knobs and monitoring interfaces. This allows an easy exploitation of many features required in power-aware autonomic computing, such as scaling the clock frequency of the cores, monitoring the power consumption, dynamically turning off CPUs, etc. On the other hand, to interact with the application, multiple possibilities are provided by NORNIR:

Black-box With this kind of interaction, the source code of the controlled application does not need to be modified, and NORNIR will monitor application performance by using hardware performance counters (e.g. number of instructions executed per time unit).

Instrumentation If users are willing to modify the application to be controlled, they could insert some instrumentation calls in the source code of the application, to track application progress (e.g., in streaming applications, the number of stream elements processed per time unit). Although this is more intrusive than the *Black-box* approach, in this case the user can express the performance requirements in a more meaningful way, rather than expressing them in terms of CPU instructions.

Runtime In some cases, NORNIR can directly interact with the runtime of the application, not requiring any modification to the application code but at the same time being able to collect high-level performance metrics, such as number of stream elements processed per time unit. Moreover, in this case it is also possible to exploit more efficient actuators, such as the *concurrency throttling* knob, which allows NORNIR to dynamically change the number of threads used by the application. Currently, NORNIR provides this possibility only for applications implemented using the *FastFlow* framework [2]. In this work, we will extend this possibility also to applications using OpenMP.

Nornir API Lastly, NORNIR also provides a programming API to implement parallel applications, relying on a runtime based on *Fastflow* [2]. This approach allows a fine-grained control on the application, but it is also the most intrusive one, since it requires the user to rewrite the application by using a different programming framework.

NORNIR limitations mostly depend on the limitation of the algorithms used for the *Analyze* and *Plan* phases. For example, one common assumption made by these algorithms is that the application can reasonably balance the workload among the threads. If this is not the case, this could affect the accuracy of these algorithms.

3.2 OMPT

The OpenMP Tools API (OMPT) [12, 4] is an Application Programming Interface for first-party performance tools. By using OMPT, it is possible to track

different events during the lifetime of an OpenMP application, such as tasks creation and destruction, OpenMP initialization, synchronizations, and others. To intercept these events, the OMPT user must define callbacks which will be invoked every time one of these events occurs. Then, these callbacks can be either statically linked to the application when it is compiled, or they can be dynamically loaded by specifying the dynamic library containing such user-defined callbacks in the `LD_PRELOAD` environment variable. By tracking these events, it would be possible to monitor the application progress and performance (e.g., in terms of number of OpenMP tasks executed per time unit), which is what is needed by Nornir to monitor an application and to apply autonomic decisions.

4 Design and Implementation

In this section we will describe how Nornir has been extended to transparently monitor OpenMP applications. First, because the OMPT API is not yet provided by most OpenMP implementations, we rely on an experimental LLVM-based implementation [1]. To interface the Nornir *manager* to the OpenMP application, we first intercept the initialization of the OpenMP application by using the OMPT API. When OpenMP is initialized, the *manager* is created and started as an external process. The *manager* will execute the MAPE loop and, at each iteration of the MAPE loop, in the *monitor* phase it will collect the application performance by sending a request to the application process. Every time a task is created, the event will be intercepted through OMPT. If a request by the Nornir manager was present, then the number of tasks executed per time unit will be communicated to the manager, otherwise the number of executed tasks will be stored locally. This interaction between the application and the *manager* is implemented by using the Riff library, which is a small library (provided by Nornir) for monitoring application performance, which was already used for *Instrumentation* interactions (see Section 3.1). This exchange between the OpenMP application and the Nornir manager is depicted in Figure 1.

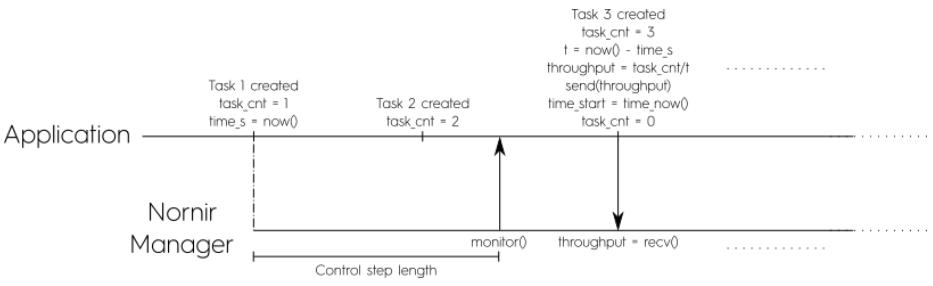

Fig. 1: Interaction between the OpenMP application and the Nornir manager.

However, this approach would not work for applications composed only of a single OpenMP parallel loop. In this case, the OpenMP runtime would create

Listing 1.1: Nornir configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<nornirParameters>
<requirements>
    <throughput>100</throughput>
</requirements>
</nornirParameters>
```

a number of tasks equal to the number of cores available on the machine, and then each task will execute different chunks of loop iterations. Since tasks are created only once, we would not be able to track application progress. To address this problem, we also need to track the events associated to the scheduling of chunks of loop iterations. However, this type of callbacks is not defined by the OMPT API specification. For this reason, we extended the LLVM-based OMPT implementation to also track the scheduling of chunks of iterations in OpenMP parallel loops. This modified OpenMP implementation has been released as open source [16] and is used by Nornir by default. It is worth remarking that if the application is composed of a single parallel loop and if static scheduling is used, then we would have the same problem, since only one chunk per thread will be generated, and we will not be able to track application progress.

To impose specific performance and power consumption requirements, the user needs first to build an XML file containing, among others, the minimum performance required (in terms of tasks or loop iterations processed per second) and the maximum allowed power consumption. The path of this file must be then specified in the `NORNIR_OMP_PARAMETERS` environment variable. For example, if the user wants his/her OpenMP application to execute 100 loop iterations per second, the XML file like the one in Listing 1.1 should be provided.

Then, the user needs to specify the path of the Nornir dynamic library and of the modified OpenMP implementation in the `LD_PRELOAD` environment variable. This process is wrapped in a script which is provided by Nornir and which sets these paths in a proper way according to the way Nornir was installed. For example, to run the `foo` OpenMP application enforcing the requirements specified in the `config.xml` configuration file, it is sufficient to run the command: `nornir_openmp foo config.xml`.

It is worth mentioning that the same approach could also be adopted for other frameworks (e.g. Intel TBB). To do that, we should locate the points in the runtime code where we could track application progress (e.g. where tasks are created), and then insert instrumentation calls in the same way we did for OpenMP. This could be either done by using similar profiling API, or by actually modifying the runtime source code.

5 Experiments

In this section we first evaluate the overhead introduced by Nornir (which also includes the overhead for intercepting OpenMP events). Then, we will show how by applying our approach it is possible to transparently enforce arbitrary performance and power consumption requirements on OpenMP applications. For our analysis we selected the *blackscholes* and *bodytrack* benchmarks from the PARSEC benchmark suite [8] and the *bt* and *cg* applications from the NAS benchmark [5]. We used the *native* input for the PARSEC applications, the class *B* input for *bt* and the class *C* input for *cg*. 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. We did not use the hyper-threading, and the applications used at most 24 cores in our experiments. The software environment consists of Linux 3.14.49 x86_64 shipped with CentOS 7.1 and `gcc` version `4.8.5`.

Every experiment has been executed a number of times, until the 95% confidence interval from the mean was lower than the 5% of the mean, and we report the entire distribution of results as a *boxplot* (e.g. see Figure 2). In a boxplot, the upper and lower borders of the box represent the third ($Q3$) and first ($Q1$) quartile respectively. Being IQR the interquartile range (i.e. $Q3$ - $Q1$), the upper and lower whiskers represent the largest sample lower than $Q3 + 1.5 \cdot IQR$ and the smallest sample greater than $Q1 - 1.5 \cdot IQR$. All the points outside these whiskers are considered to be outliers and are plotted individually. The line inside the box represents the median and the small diamond represents the mean.

5.1 Overhead

To measure the overhead introduced by Nornir and OMPT, we first executed the applications in their default configuration (denoted as *Default*), without any kind of instrumentation and without enabling OMPT. Then, we use OMPT but we do not communicate any data to Nornir (denoted as *OMPT*). Eventually, we attach Nornir to the application, but we do not change its configuration. In this way, we can separately measure the overhead introduced by OMPT to intercept OpenMP calls and the overhead introduced by Nornir plus OMPT, including the overhead to communicate performance information between the application and the Nornir manager. We report the results of this analysis in Figure 2. We report on the x-axis the different applications, and on the y-axis the application throughput (in terms of tasks/iterations executed per time unit). The throughput is normalized with respect to the median throughout of the default execution (the higher the better), so that values lower than one represent a lower throughput with respect to the default execution.

As we can see from both the medians and the means, while for *blackscholes* and *bodytrack* there are no relevant differences, for *bt* and *cg* we have some

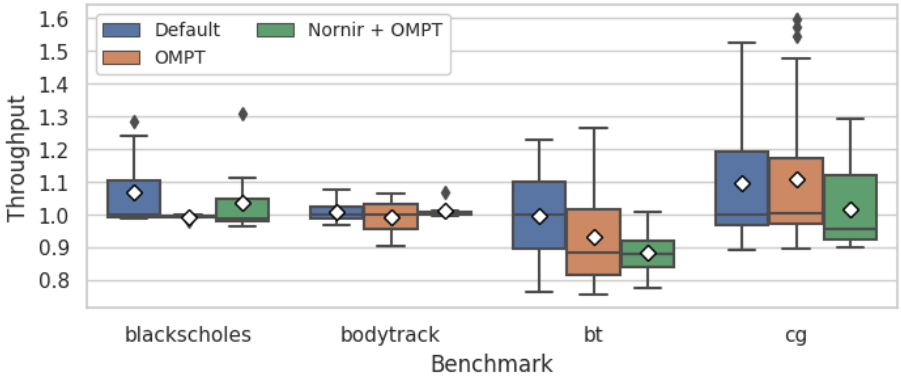

Fig. 2: Throughput comparison between the default execution and the execution where Nornir and OMPT are used. Throughput is normalized with respect to the default execution. The higher the better.

performance degradation. For *bt*, the performance degradation is less then 10%, which however seems to be caused by OMPT rather than by the communication of the performance information to Nornir. On the contrary, for *cg* we have an overhead lower than 5%, which the data show to be caused by Nornir.

5.2 Throughput and Power Consumption Requirements

We now analyze the ability of Nornir to set explicit performance and power consumption requirements, by using the performance information extracted with OMPT. To enforce performance and power consumption requirements we used one of the several algorithms provided by Nornir (`ANALYTICAL_FULL`). This algorithm tunes the number of cores used by the application and their clock frequency, searching for a configuration which satisfies the requirements expressed by the user. To avoid biases due to the selection of a specific requirement, we perform our test for different requirements. For example, being T the application throughput, we set as throughput requirements $0.2 \cdot T$, $0.4 \cdot T$, ..., T. A similar approach has been adopted for power consumption requirements[2].

We report in Figure 3 the results of this evaluation for performance requirements. We show on the x-axis the performance requirements expressed as a percentage of the maximum performance. On the y-axis we show the obtained performance normalized with respect to the requirement. Namely, 1.0 represents the requirement and values higher or equal than one mean that Nornir was able to satisfy the requirement. As shown in the plot, we were able to run the application so that its throughput is higher or equal than that required by the user.

[2] For power consumption requirements, we do not consider the 0.2 requirement since it can never be enforced, not even by using only one core at minimum clock frequency.

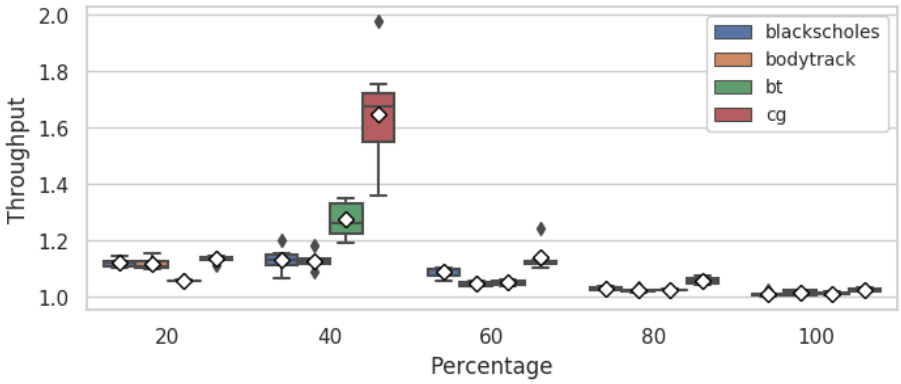

Fig. 3: Performance of the analyzed applications under different performance requirements. On the x-axis the performance requirements expressed as a percentage of the maximum performance. On the y-axis the obtained performance normalized with respect to the requirement (i.e. values higher than one mean that Nornir was able to satisfy the requirement).

In almost all the cases (with the exception of *bt* and *cg* on the 40% requirement), the achieved throughput was at most 20% higher than the user requirement.

Similarly, in Figure 4 we report the results of the evaluation for power consumption requirements. We show on the x-axis the power consumption requirements expressed as a percentage of the maximum power consumption. On the y-axis we report the obtained power consumption normalized with respect to the requirement. Namely, 1.0 represents the requirement and values lower or equal than one mean that Nornir was able to satisfy the power consumption requirement. Also in this case we were able to correctly enforce the user requirements, having a power consumption which is always lower or equal to that specified by the user. In all the cases except one (*blackscholes* for the 100% requirement), Nornir was able to find a configuration characterized by a power consumption at most 5% lower than that required by the user.

6 Conclusions and Future Work

When designing autonomic solutions, a relevant design decision is related to the way in which the application performance is monitored. Several solutions are possible, each requiring a different effort to the application programmer. In this work we analyze the possibility to intercept different events in OpenMP applications to track their performance. Such solution would not require any effort to the application programmer.

To implement this process we relied on the OMPT API, which allowed us to track OpenMP applications and to interface them to the Nornir framework, allowing us to transparently set arbitrary performance and power consumption

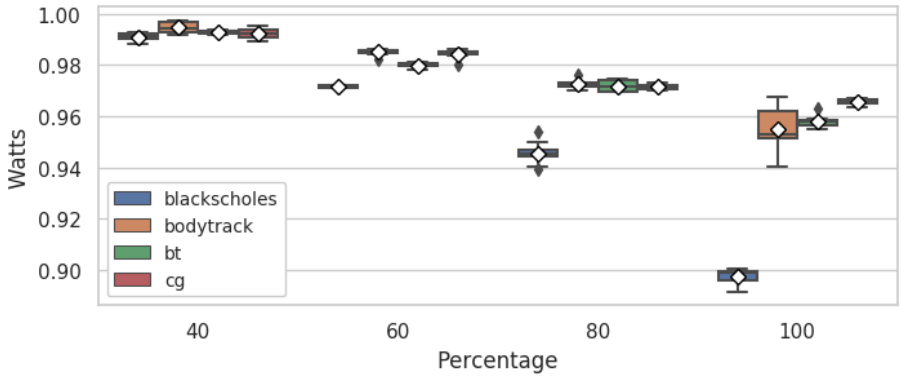

Fig. 4: Power consumption of the analyzed applications under different power consumption requirements. On the x-axis the power consumption requirements expressed as a percentage of the maximum power consumption. On the y-axis the obtained power consumption normalized with respect to the requirement (i.e. values lower than one mean that NORNIR was able to satisfy the requirement).

requirements on existing applications. To correctly monitor applications composed of a single parallel loop, we modified the OMPT backend to also track the scheduling of chunks of iterations in parallel loops. Moreover, all the developed code has been integrated into NORNIR, which is a publicly available open-source framework. Eventually, we showed that the introduced performance overhead is negligible and that we can correctly enforce arbitrary requirements.

In the future, we would like to extend the interaction with OpenMP also to the *execute* phase of the MAPE loop, by dynamically changing the number of threads used by the OpenMP runtime. Moreover, we would like to monitor the performance at a finer granularity, for example by intercepting individual iterations of the parallel loop rather than the scheduling of chunks of iterations.

References

1. LLVM runtime with experimental changes for OMPT. `https://github.com/OpenMPToolsInterface/LLVM-openmp`, 2019. [Online; accessed 12-June-2019].
2. Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley and Sons, Ltd, 2017.
3. Ferdinando Alessi, Peter Thoman, Giorgis Georgakoudis, Thomas Fahringer, and Dimitrios S. Nikolopoulos. Application-level energy awareness for openmp. In Christian Terboven, Bronis R. de Supinski, Pablo Reble, Barbara M. Chapman, and Matthias S. Müller, editors, *OpenMP: Heterogenous Execution and Data Movements*, pages 219–232, Cham, 2015. Springer International Publishing.
4. Martin Schulz Alexandre Eichenberger, John Mellor-Crummey. OpenMP Technical Report 2 on the OMPT Interface. `https://www.openmp.org/wp-content/uploads/ompt-tr2.pdf/`, 2019. [Online; accessed 12-June-2019].

5. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks - Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.
6. Denis Barthou, Andres Charif Rubial, William Jalby, Souad Koliai, and Cédric Valensi. Performance tuning of x86 openmp codes with maqao. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 95–113, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
7. Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.
8. 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*. ACM, 2008.
9. Daniele De Sensi, Tiziano De Matteis, and Marco Danelutto. Simplifying self-adaptive and power-aware computing with nornir. *Future Generation Computer Systems*, pages –, 2018.
10. Daniele De Sensi, Massimo Torquati, and Marco Danelutto. A reconfiguration algorithm for power-aware parallel applications. *ACM Transactions on Architecture and Code Optimization*, 13(4):43:1–43:25, dec 2016.
11. Daniele De Sensi, Massimo Torquati, and Marco Danelutto. Mammut: High-level management of system knobs and sensors. *SoftwareX*, 6:150 – 154, jul 2017.
12. Alexandre Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. Ompt: An openmp tools application programming interface for performance analysis. volume 8122, 09 2013.
13. D. Li, B. R. de Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos. Hybrid mpi/openmp power-aware computing. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.
14. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.
15. Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Controlling software applications via resource allocation within the heartbeats framework. In *49th IEEE Conference on Decision and Control (CDC)*, pages 3736–3741. IEEE, dec 2010.
16. Daniele De Sensi. Chunk scheduling callbacks for OMPT. `https://github.com/DanieleDeSensi/LLVM-openmp`, 2019. [Online; accessed 12-June-2019].
17. Rishad A. Shafik, Anup Das, Sheng Yang, Geoff Merrett, and Bashir M. Al-Hashimi. Adaptive energy minimization of openmp parallel applications on many-core systems. In *Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures*, PARMA-DITAM '15, pages 19–24, New York, NY, USA, 2015. ACM.
18. W. Wang, A. Porterfield, J. Cavazos, and S. Bhalachandra. Using per-loop cpu clock modulation for energy efficiency in openmp applications. In *2015 44th International Conference on Parallel Processing*, pages 629–638, Sep. 2015.