Nornir: A Customizable Framework for Autonomic and Power-Aware Applications

D. De Sensi ✉, T. De Matteis, and M. Danelutto

Department of Computer Science, University of Pisa, Pisa, Italy,
{desensi, dematteis, marcod}@di.unipi.it

Abstract. A desirable characteristic of modern parallel applications is the ability to dynamically select the amount of resources to be used to meet requirements on performance or power consumption. In many cases, providing explicit guarantees on performance is of paramount importance. In streaming applications, this is related with the concept of *elasticity*, i.e. being able to allocate the proper amount of resources to match the current demand as closely as possible. Similarly, in other scenarios, it may be useful to limit the maximum power consumption of an application to do not exceed the power budget. In this paper we propose Nornir, a customizable C++ framework for autonomic and power-aware parallel applications on shared memory multicore machines. Nornir can be used by autonomic strategy designers to implement new algorithms and by application users to enforce requirements on applications.

Keywords: autonomic, power-aware, quality of service, framework

1 Introduction

Nowadays, sensors, social network interactions and heterogeneous devices interconnected in the *Internet of Things* are continuously producing unbounded streams of data. In *Data Stream Processing* applications, this flow of information must be gathered and analyzed "on the fly" in order to produce timely responses. Systems for high-frequency trading, health-care, network security and disaster managements are typical examples: a massive flow of data must be processed in real-time to detect anomalies and take immediate actions.

Usually, the development of stream processing applications requires to exploit parallel and distributed hardware in order to meet *Quality of Service* (QoS) requirements of high throughput (i.e. applications must be able to cope with high volume of incoming data) and low latency (i.e. results must be computed in a short period of time). Due to their long-running nature (24hr/7 days), stream processing applications are naturally affected by "ebbs and flows" in the input rate and workload characteristics. These variations need to be sustained to provide the QoS required by the users without interruptions. However, run as fast as possible is not a viable solution in a world in which power consumption management has become a major concern for data centers due to economic cost, reliability problems and environmental reasons. In other cases, explicitly application's power consumption may be useful to do not exceed the available power budget. Not being able to enforce such requirement may lead to hardware failures and to a system outage. *Autonomicity* (sometimes referred as *adaptivity* or *elasticity*) is a fundamental feature: applications must be able to autonomously adjust their resources usage (i.e. their *configuration*) to accommodate dynamic requirements and workload variations by maintaining the desired QoS in terms of performance and/or power consumption.

Existing Stream Processing Systems (SPSs) fall short in handling this problem. Delegating the decisions to the user (like in [1]) or to applications (as in [19]) are not wise decisions since they will require a human intervention or a deep knowledge of the parallel computation to the application programmer. On the other hand, in the literature there are plenty of proposals of autonomic algorithms (e.g. [11], [6], [17], [10], [18]). Implementing such strategies is a cumbersome and error-prone duty for the application programmer, that has to deal with many architectural low-level issues related to hardware mechanisms management like voltage, frequency, cores topology, etc. Even interfacing with applications in order to collect monitoring data may not be an easy task. Indeed, in many cases the proposed strategies are only simulated or, even when actually implemented, they are embedded inside the application code and it is very difficult to port them on different applications. For these reasons, we believe that providing a customizable framework would allow the autonomic strategies designers to just focus on the algorithm, exploiting the infrastructure provided by the framework to collect the data and to apply the decisions. This is a fundamental step for building efficient autonomic techniques and for their wide adoption.

In this paper, we propose Nornir, a customizable C++ framework for autonomic and power-aware parallel applications on shared memory multicore machines[1]. Our focus is on applications composed by a single, parallel functionality (an *operator*). The support for applications that can be expressed as the composition of different operators will be included in future releases of the framework. Nornir can be used by different actors:

- Autonomic strategy *designers* can customize every aspect of Nornir: the monitoring, the management of hardware mechanisms and the planning policies. The designer can just focus on the implementation of his new autonomic strategy by using the provided set of resource management mechanisms and the application monitoring infrastructure. *Designers* can develop strategies to explicitly control power consumption, performance or both of them.
- Application *programmers* can use it to interface an already existing application to Nornir. Nornir also provides a programming interface for parallel applications, to be used if the application needs to be written from scratch.
- Application *users* specify requirements on performance and/or power consumption of their applications. Nornir will be in charge of monitoring the

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

application execution and selecting its appropriate configuration (e.g. number of cores, clock frequency, etc...) to enforce the imposed requirements.

Currently, different state of the art autonomic techniques have been already implemented in Nornir, allowing the algorithm *designer* to compare his new algorithm with other existing ones.

The paper is organised as follows. In Section 2 we outline the related work. In Section 3 we describe how the *user* can express requirements on his application by using Nornir. In Section 4 we show how the *programmer* can interface a new or an existing application to Nornir and section 5 describes how Nornir can be customised by autonomic strategies *designers.* Some experimental results will be shown in Section 6 and conclusions are eventually drawn in Section 7.

2 Background and Related Work

An autonomic or autonomic system is able to alter his behavior according to QoS requirements and to the surrounding conditions in order to achieve some goal, without any human intervention. Altering the behavior usually implies changing the *configuration* of the application, e.g. the amount of used resources.

Existing algorithms are usually time-driven and, at each time step, act by following a generic *Monitor-Analyze-Plan-Execute* (MAPE) loop [14]. In the *Monitor* phase, various measurements are collected from the application (e.g. performance and power consumption). In the *Analyze* phase monitored data, collected at the current and previous time steps, is compared against the user's requirements. If requirements are violated, the *Plan* phase a new optimal resources allocation will be computed. This planned decision is communicated to the *Execute* phase, that applies the new resources allocation to the application.

Different autonomic strategies have been proposed, to satisfy user's requirements in terms of performance ([17], [11], [18], [16]), power consumption ([10]) or both of them ([6], [9]). Such requirements are usually enforced even in presence of workload fluctuations or external interferences. However, in many cases, these techniques are only simulated or implemented for specific applications.

In literature, some proposed framework target a problem similar to the one we are addressing in this work [20], [12], [15]. However, they provide very limited customization opportunities, are quite outdated and the source code is not publicly available. Moreover, they do not provide any explicit support for streaming applications. The work most similar to ours is *SEEC* [13]. In this work, the authors describe the design of a framework for self-aware computing. Such framework is customizable, allowing the autonomic strategy *designer* to specify custom monitoring and execution mechanisms. Nevertheless, there are some limitations with respect to our work. First of all, there isn't an explicit concept of *stream.* As shown in [9] this can lead to unnecessary reconfigurations, since it would not be possible to know whether workload fluctuations are caused by intrinsic changes in the application or by changes in the arrival rate of data to the application. In addition to this SEEC allows the customisation of the *Monitor* and *Execute* parts but provides its own *Plan* algorithm. Albeit being a flexible strategy, it is not possible to replace it with a different one. On the other hand, in Nornir this aspect is customizable as well. This is an important feature since allows the strategy *designer* to quickly prototype and validate his own planning strategies and to easily compare it with other existing ones. Lastly, the implementation of the *SEEC* framework is not publicly available.

3 User

The *user* needs to detail which kind of constraints should be enforced by Nornir on his application by specifying them through an `XML` file. Requirements can be expressed on the metrics reported in Table 1.

Metric	**S**	**Description**
Bandwidth	B	Number of stream elements processed per second (number of iterations executed per second for non streaming, iterative, applications).
Latency	L	Time required to process a single stream element.
Completion Time	T	Time required to process all the elements on the stream$^{\not\infty}$.
Utilisation Factor	ρ	Percentage of time spent doing useful work (i.e. processing stream elements). $100 - \rho$ is the percentage of time wasted by the application waiting for new data to arrive from the stream.
Power Consumption	P	Since current operating systems don't provide mechanisms to monitor the individual power consumption of each application, this may correspond to the system level power consumption.
Energy	E	Power integrated over time$^{\not\infty}$.

Table 1: Parameters that can be controlled by the *user*. $\not\infty$ = Meaningful only if the stream has finite size. The *user* needs to specify the expected stream length.

Despite the target of this work is towards Data Stream Processing applications, Nornir can manage generic iterative applications, for example by enforcing requirements on the latency of one iteration or on power consumption. It is possible to express constraints on more than one metric at the same time, for example by asking Nornir to find the configuration characterized by the lowest power consumption among those with a bandwidth higher than a certain threshold. Similarly, the *user* can ask Nornir to find the most performing configuration among those characterized by a power consumption lower than a specified bound.

The `XML` file can also be used to specify other parameters, like the autonomic strategy to be used, on which executors Nornir should operate, the duration of the MAPE step (i.e. the *control step*), etc... The following code snippet shows a configuration file example, used to ask Nornir to find the best performing configuration characterised by a power consumption lower than 50 Watts and using a control step of 500 milliseconds:

```
<?xml version="1.0" encoding="UTF-8"?>
<nornirParameters>
    <requiredBandwidth>MAX</requiredBandwidth>
    <powerBudget>50</powerBudget>
    <samplingIntervalSteady>500</samplingIntervalSteady>
</nornirParameters>
```

4 Application Programmer

A controlled parallel application is coupled with a Manager, which is in charge of executing the MAPE control loop. The Manager runs in a separate thread/process and interacts with the application to gather monitoring data and to apply reconfiguration decisions (e.g. changing the number of threads) to enforce the user's requirements.

Nornir offers different possibilities to the application *programmers* for realizing this interaction, allowing to chose the desired tradeoff between configuration optimality and required programming effort. In the following, we will discuss these different opportunities.

Application written from scratch The *programmer* can write a parallel application by using the parallel programming interface provided by Nornir. This interface allows the *programmer* to write both structured (i.e. parallel patterns based) and unstructured applications expressed as a graph of concurrent activities. By doing so, Nornir can access many internal features of the runtime, thus extending its monitoring capabilities and being able to operate on additional executors. Details about this API can be found in [5].

Application written using a supported framework Nornir can easily interface with existing applications written in one of the supported parallel programming environments. At the time being, the only supported framework is FastFlow[2]. FastFlow is a pattern based parallel programming framework, particularly suited for parallel streaming applications. In this case is sufficient for the *programmer* to provide to the Manager a handler to the application, as shown in the following code snippet:

```
Parameters p("parameters.xml"); // Load Nornir parameters.
ManagerFarm<> m(&farm, p);      // farm = FastFlow Application.
m.start();                      // Start application.
m.join();                       // Wait for application end.
```

Instrumented application If the application is implemented with a non-supported framework, the *programmer* can interface it to a Nornir Manager running in a separate process as a server. The application will act like a client, by inserting few instrumentation calls in his application, as shown in the following snippet:

[2] `http://calvados.di.unipi.it/`

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

```
1 Monitor r("parameters.xml");
2 StreamElement* s;
3 while(s = receive()){
4   r.begin();
5   process(s);
6   r.end();
7 }
8 r.terminate();
```

On the left, we have the original, already existing, streaming applications and on the right the same application after it has been instrumented. In line 1 the application opens a connection towards the manager and sends to it the parameters (e.g. QoS requirements). Then, for each stream element, after receiving it from the stream (line 3), the processing is wrapped between 2 calls (lines 4 and 6). By doing so, the performance of the application will be monitored and the data will automatically flow towards the Manager. Eventually, in line 8, the connection with the Nornir Manager is closed. Note that this approach only requires inserting 4 instrumentation calls in the already existing application.

Black-box application In some cases the *programmer* may not have the possibility to instrument and recompile his application. In such cases, the only way Nornir has to monitor the application performances is to rely on performance counters, for example by monitoring the number of assembler instructions executed per time unit (i.e. instructions per second (IPS)). Accordingly, the *user* should express his performance requirements for the application in terms of IPS. Correlating the IPS to the actual application bandwidth is not an easy task and not so intuitive from the *user* perspective. Moreover, as shown in [13] performance counters may not be a good performance proxy. For these reasons, this approach should only be used if none of the previous ones can be adopted. Suppose that the *user* wants to specify some constraint on his `streamprocessing` application. Then, he can run it by using the Nornir applications launcher:

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

Note that this doesn't require any intervention from the *programmer*.

To summarize, Nornir provides different solutions to interact with applications. The optimal solution would be to program the streaming application with the provided programming API or to use a supported framework. By accessing the runtime support, Nornir can also access other executors (Section 5.3) that would not be available otherwise (e.g. changing the number of threads), thus improving the quality of the selected configuration, as shown in [3]. If it is not possible to rewrite the application by using a different framework, the *programmer* can just insert few instrumentation calls inside the application, allowing Nornir to monitor it. Eventually, if even the instrumentation is not feasible (e.g. because the *programmer* can't or doesn't want to change the application code and/or recompile it), Nornir can still manage the application, not requiring any programming effort. However, we can only monitor system performance counters and we lose the concept of *stream*. Moreover, expressing performance constraints in this scenario could be not intuitive from the *user* perspective.

5 Strategy Designer

In this section, we describe the design of the framework, focusing on how it can be customized by the autonomic strategy *designer*. The general architecture of Nornir is depicted in Figure 1.

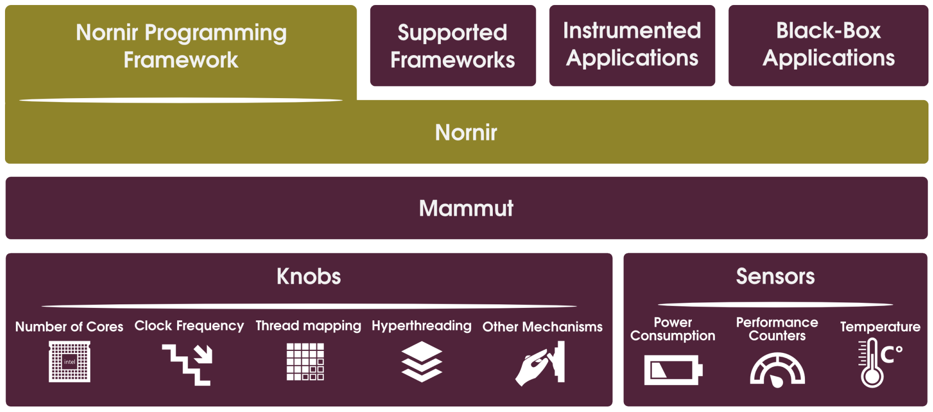

Fig. 1: General Architecture of Nornir Framework.

In the upper layer, we have the different types of applications that can be interfaced to Nornir (Section 4). Nornir interacts with the system knobs and sensors (e.g. power consumption one), by using Mammut [7][3]. Mammut is an object-oriented C++ framework allowing a transparent and portable monitoring of system sensors as well as management of several system knobs.

The following code snippet shows a simplified version of the main parts of Nornir implementation[4]:

```
typedef enum{
  KNOB_VIRTUAL_CORES = 0,
  ...
  KNOB_NUM
}KnobType;

class Manager{
  ...
  void run(){
    while(isRunning()){
      sleep(_parameters.samplingInterval);
      ApplicationSample s = getSample(); // Monitor
      storeSample(s);
      KnobsValues k = _selector->getNextKnobsValues(); // Analyze & Plan
      for(uint i = 0; i < KNOB_NUM; i++){
        _knobs[i]->changeValue(k[i]); // Execute
      }
    }
  }
```

[3] http://danieledesensi.github.io/mammut/
[4] Actual implementation consists of approximately 18000 lines of code

```
20    virtual ApplicationSample getSample() = 0;
21  };
22
23  class Knob{
24    ...
25    std::vector<double> _knobValues;
26    virtual void changeValue(double v) = 0;
27  };
28
29  class Selector{
30    ...
31    virtual KnobsValues getNextKnobsValues() = 0;
32  }
```

Source Code 1.1: Simplified version of the main parts of Nornir implementation.

The meaning of this code snippet will become more clear after the end of this section. For the moment, we can focus on the MAPE (lines 10-18) loop. In the remaining part of this section, we describe how each of the 3 steps is designed and how they can be customized by the autonomic strategy designer.

5.1 Monitor

As we shown in Section 4, the *user* can get the highest benefits from using Nornir, by using it on an application written with the Nornir parallel programming interface or on an application written by using one of the supported frameworks. At the moment, this only includes FastFlow. To interface Nornir with other runtimes, the *designer* needs to define a new manager for the new runtime support, by defining a subclass of the Manager class and implementing the `getSample` function (Code 1.1, line 20). In this function the *designer* should implement the retrieval of a new monitored sample from the runtime (i.e. the metrics in Table 1 and additional custom values). This function will be called by Nornir (line 12) and the sample will be stored (line 13) in order to be accessible from the *Plan* phase (Section 5.3).

5.2 Execute

To implement a new executor, the *designer* must define a subclass of the Knob class (Code 1.1, lines 23-27). In the constructor, the `_knobValues` vector must be populated with the set of values that the knob can assume. When the planning phase terminates, the function `changeValue` will be called by the manager on all the available knobs (lines 15-17), with the parameter `v` corresponding to the value that that specific knob must assume according to the planning algorithm. By implementing the function `changeValue`, the *designer* specifies the actual code to be executed when a request to change the value of that knob is received by the Manager. For example, if the *designer* wants to implement a knob to set the DRAM frequency, in the `changeValue` function he will insert the code to perform this action. The new Knob object must then be created and added to the `_knobs` array (used in line 16). Moreover, a new enumeration value must be assigned to this knob (lines 1-5). Currently, the following knobs are implemented:

Number of Cores Turns off (or on) some cores. If possible (e.g. for FastFlow applications), it will also change the number of threads used by the application (without stopping or restarting it), to have one thread on each active core. Otherwise, more threads will contest for the same core. Threads will be allocated to cores through the *Threads Mapping* knob, while this knob only enforces the specified number of cores to be active. If the number of threads is changed, the *application programmer* must ensure the correctness of the computation (i.e. if the application maintains an internal state, the semantic of the computation must be preserved after a reconfiguration).

Hyperthreading Level Number of hardware threads contexts to use on each physical core.

Threads Mapping Once the number of cores to use has been decided, this knob can be used to apply a given placement. For example, to place them on a set of cores sharing some resources (e.g. last level caches) for minimizing power consumption, or to place them on a set of cores with the minimum amount of shared resources, wasting more power but improving performance.

Clock Frequency Operates on the clock frequency (and voltage) of the cores, allowing to trade a decreased performance for a lower power consumption.

5.3 Analyze and Plan

To define a custom planning policy, the *designer* must define a subclass of the Selector class (Code 1.1, lines 23-27) and implement the `getNextKnobsValues` function. In its own Selector the *designer* can access different information provided by the superclass, like: parameters specified by the *user* through the `XML` file, the current configuration of the application and statistics about the previous monitored samples, to be used during the *Analyze* phase. This information is kept consistent by Nornir and should be exploited by the algorithm *designer* to decide the next configuration. Once the decision is made, the next values of each knob are stored into a KnobsValues object, an array of values (one for each knob) which can be accessed by using the enumeration values identifying the type of the knob (lines 1-5). The returned object will then be used to set the appropriate values on the available knobs (lines 9-11).

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

```
class SelectorDummy: public Selector{
  ...
  KnobsValues getNextKnobsValues(){
    KnobsValues k(KNOB_VALUE_RELATIVE);
      if(_samples->average().latency < 100){
        k[KNOB_VIRTUAL_CORES] = 25; k[KNOB_FREQUENCY] = 50;
      }else{
        k[KNOB_VIRTUAL_CORES] = 80; k[KNOB_FREQUENCY] = 100;
      }
10          return k;
11    }
12  };
```

Nornir will then automatically translate the percentage values for number of cores and frequencies in real values, according to the availability of resources on the target architecture. Alternatively, it is possible to directly express absolute values for the knobs. By replacing `KNOB_VALUE_RELATIVE` with `KNOB_VALUE_REAL` in line 4, Nornir will interpret line 6 as "*Run the application on 25 cores and set their frequency to 100Hz*". `_samples` contains the moving average (simple or exponential) of the monitored data. The type of moving average as well as the size of the moving window (or the exponential parameter) can be specified through the `XML` file.

This is the most flexible choice from the *designer* perspective since he can implement different strategies from scratch. However, it is also possible to customize some of the strategies already provided by Nornir. The following state of the art autonomic strategies are already implemented in Nornir: **i)** Two online learning strategies to enforce requirement on bandwidth and power consumption [9], [8]; **ii)** A planning strategy mixing offline and online prediction [18]; **iii)** An heuristic strategy [16] to enforce bandwidth requirements; **iv)** Two heuristics for utilisation factor optimisation [4], [5]. Being able to implement such a spectrum of different techniques, ranging from heuristics to online machine learning proves the generality and flexibility of our design.

6 Results

Nornir already provides several autonomic strategies for streaming applications. We will show the results obtained by using the algorithm described in [9] on a network monitoring application [4]. This application analyses all the packets traveling over a network, applying *Deep Packet Inspection* techniques to identify possible security threats. For our experiment we used synthetic traffic data, while the arrival rates are those of a real backbone network[5]. We asked Nornir to always guarantee a bandwidth equal to the input one, (i.e. to do not drop any input stream element), while minimizing power consumption. The application ran for 24 hours, and the results are shown in Figure 2.

No packets were dropped during this experiment and, as shown in the top part of the figure, Nornir was able to reconfigure the application so to have a power consumption proportional to the actual input data workload to be processed. In the bottom part, we can see that this was possible since the number of used cores and the clock frequency was dynamically changed according to the workload intensity. When the autonomic strategy was not applied, the application was always characterized by the maximum power consumption during the 24 hours.

[5] `http://www.caida.org/data/realtime/passive/?monitor=equinix-chicago-dirA`, 24 hours of traffic between 03/01/2016 and 04/01/2016.

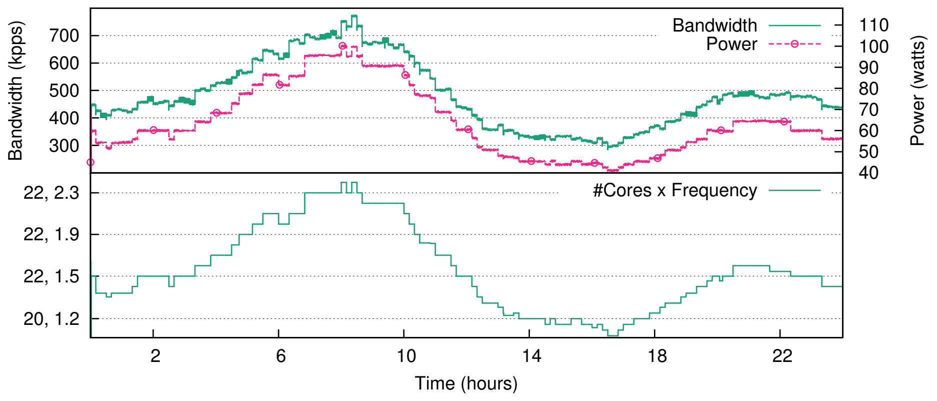

Fig. 2: Time behaviour of an application controlled by NORNIR, in presence of fluctuations in the input pressure. Bandwidth is expressed in thousands of packets per second.

7 Conclusions and Future Work

In this work we presented NORNIR, a framework to allow application's *users* to enforce performance and power consumption requirements on streaming applications. An application *programmer* will interface the *user*'s application to the NORNIR MANAGER. Moreover, thanks to a modular design, it is possible for an autonomic strategy *designer* to embed his own strategies inside NORNIR, by focusing only on the algorithmic part of his strategy, since the *monitor* and *execute* phases are managed by NORNIR. NORNIR already provides several autonomic strategies and supports different types of applications, proving its flexibility.

As a future work, we would like to support in NORNIR applications expressed as graphs of operators. In this case, decisions taken by an operator manager may influence the behavior of other parts of the computation, requiring to coordinate different managers to find agreements in reconfiguration decisions (e.g. *hierarchical managers*). Another important step would be extending the support to distributed memory architecture. Eventually, we will provide the user with information about the cost of reconfigurations, which may be helpful in mitigating their impact on performance [2].

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

References

1. Apache Storm: `http://storm.apache.org/` (2017)
2. Bertolli, C., Mencagli, G., Vanneschi, M.: A cost model for autonomic reconfigurations in high-performance pervasive applications. In: Proc. of the 4th ACM Intl. Workshop on Context-Awareness for Self-Managing Systems. pp. 3:20–3:29 (2010)

3. Danelutto, M., De Matteis, T., De Sensi, D., Torquati, M.: Evaluating concurrency throttling and thread packing on smt multicores. In: Proc. of the 25th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (2017)
4. Danelutto, M., De Sensi, D., Torquati, M.: Energy driven adaptivity in stream parallel computations. In: Proc. of 23th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing. pp. 103 – 110. IEEE, Turku, Finland (2015)
5. Danelutto, M., De Sensi, D., Torquati, M.: A power-aware, self-adaptive macro data flow framework. Parallel Processing Letters 27(01), 1740004 (2017)
6. De Matteis, T., Mencagli, G.: Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In: Proc. of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 13:1–13:12 (2016)
7. De Sensi, D., Torquati, M., Danelutto, M.: Mammut: High-level management of system knobs and sensors. SoftwareX 6, 150 – 154 (2017)
8. De Sensi, D.: Predicting performance and power consumption of parallel applications. In: Proc. of 24th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing. pp. 200 – 207 (Feb 2016)
9. De Sensi, D., Torquati, M., Danelutto, M.: A reconfiguration algorithm for power-aware parallel applications. ACM Trans. Archit. Code Optim. 13(4), 43:1–43:25 (2016)
10. Gandhi, A., Harchol-Balter, M., Das, R., Kephart, J., Lefurgy, C.: Power Capping Via Forced Idleness. In: Proc. of Workshop on Energy-Efficient Design (2009)
11. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. IEEE Transactions on Parallel and Distributed Systems 25(6), 1447–1463 (June 2014)
12. Goel, A., Steere, D., Pu, C., Walpole, J.: Swift: A feedback control and dynamic reconfiguration toolkit. Tech. rep. (1998)
13. Hoffman, H.: Seec: A Framework for Self-aware Management of Goals and Constraints in Computing Systems. Ph.D. thesis, Cambridge, MA, USA (2013)
14. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (Jan 2003)
15. Li, B., Nahrstedt, K.: A control-based middleware framework for quality-of-service adaptations. IEEE Journal on Selected Areas in Communications 17(9), 1632–1650
16. Li, J., Martínez, J.F.: Dynamic power-performance adaptation of parallel computation on chip multiprocessors. Proc. of Intl. Symposium on High-Performance Computer Architecture pp. 77–87 (2006)
17. Lohrmann, B., Janacik, P., Kao, O.: Elastic stream processing with latency guarantees. In: The 35th Intl. Conf. on Distributed Computing Systems (2015)
18. Mishra, N., Zhang, H., Lafferty, J.D., Hoffmann, H.: A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints. ACM SIGARCH Computer Architecture News 43(1), 267–281 (Mar 2015)
19. Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., Zhang, Z.: Timestream: Reliable stream computation in the cloud. In: Proceedings of the 8th ACM European Conference on Computer Systems. pp. 1–14. EuroSys '13, ACM, New York, NY, USA (2013)
20. Zhang, R., Lu, C., Abdelzaher, T.F., Stankovic, J.A.: Controlware: a middleware architecture for feedback control of software performance. In: Proceedings 22nd International Conference on Distributed Computing Systems. pp. 301–310 (2002)