Service Level Objectives via C++11 Attributes

Dalvan Griebler[1], Daniele De Sensi[2], Adriano Vogel[1], Marco Danelutto[2], and Luiz Gustavo Fernandes[1]

[1] Pontifical Catholic University of Rio Grande do Sul (PUCRS)
dalvan.griebler@acad.pucrs.br
[2] Department of Computer Science, University of Pisa (UNIPI)

Abstract. In recent years, increasing attention has been given to the possibility of guaranteeing Service Level Objectives (SLOs) to users about their applications, either regarding performance or power consumption. SLO can be implemented for parallel applications since they can provide many control knobs (*e.g.*, the number of threads to use, the clock frequency of the cores, etc.) to tune the performance and power consumption of the application. Different from most of the existing approaches, we target sequential stream processing applications by proposing a solution based on C++ annotations. The user specifies which parts of the code to parallelize and what type of requirements should be enforced on that part of the code. Our solution first automatically parallelizes the annotated code and then applies self-adaptation approaches at run-time to enforce the user-expressed objectives. We ran experiments on different real-world applications, showing its simplicity and effectiveness.

Keywords: Parallel Programming, Adaptive and Autonomic Computing, Power-Aware Computing, Domain-Specific Language.

1 Introduction

The rich stream processing application domain motivated the creation of different parallel programming environments/tools to speed up the computation of data stream. In multi-core systems, this is typically exploited by using linear or non-linear pipeline structures [15]. To this purpose, state-of-the-art framework such as Streamit, TBB, and FastFlow provide different programming approaches and interfaces with a reasonable performance scalability for this domain [19, 16, 1]. Although these frameworks are equipped with high-level patterns implementation to express the parallelism, they are still closer to expert system programmers rather than to the application domain programmers. Seeking to provide domain-specific and suitable abstractions for stream parallelism, SPar was created. Different from these frameworks, application programmers are invited to express parallelism with SPar through C++11 annotation without the need for rewriting/restructuring the sequential source code semantics [10].

Moreover, stream processing applications usually have unpredictable load fluctuations and uncertain end of execution (may never end) characteristics [3].

Therefore, besides the need for improving performance through the efficient exploitation of the multi-core parallelism, there are other major concerns such adaptive and autonomic computing, power-aware computing, and efficient resource usage [9, 13]. In this direction, NORNIR was created to be a simple interface and runtime support to dynamically and automatically control the resources allocated to the application according to the user needs [7]. However NORNIR, like most existing self-adaptive solutions, only works on parallel applications.

In this paper, we propose the use of the Service Level Objective (SLO) concept [18] for sequential stream processing applications. The idea is that the programmer annotates the code by using the SPar language, synergistically specifying both the parallelism exploitation and the SLO. Based on the code annotations, SPar generates a parallel code with the NORNIR runtime system, which will dynamically adapt the parallel execution to meet the SLO. We simplify the SLO definition by introducing new attributes in SPar. The simplicity is delivered by integrating SPar annotation syntax with new attributes to specify SLO about throughput, power consumption, system utilization or a combination of these. Our approach could also be applied to REPARA project[3], which provides a set of C++11 attributes to introduce parallelism [6].

This paper is organized as follows. In Section 2 we analyze the related work in this area. Then, in Section 3 we introduce the SPar domain-specific language and in Section 4 we describe how we extended it to consider SLO. In Section 5 we perform our evaluation and, eventually, in Section 6 we draw the conclusions, and we outline some possible future directions for this work.

2 Related Work

In the literature, there are different studies targeting power consumption, throughput, and system utilization objectives. Among them, the approach of Maggio el al. [13] monitors generic applications and supports the specification of a target performance (throughput) in the parallel code. It efficiently manages the CPU cores, adapting the amount of resource usage needed. However, it supposes that the parallel application has already been implemented, and does not provide any mechanism to introduce SLO in sequential programs.

Concerning stream parallel processing for real-time data analytic, Floratou et al. [9] introduced the notion of self-regulation in Twitter's Heron framework, called Dhalion. The user defines a target throughput as an SLO parameter for Dhalion. The self-regulator engine handles the number of process and number of instances in a cloud infrastructure to provide the specified throughput. In the experiments, the results revealed that the system can dynamically adapt resources and automatically reconfigure to meet SLOs. We differently proposed three target SLOs (throughput, energy, utilization) to be expressed in sequential source codes for multi-core systems. Our adaptive runtime uses system knobs and applies machine learning algorithms to dynamically adapt CPU frequency and the number of active threads to meet SLO requirements.

[3] http://repara-project.eu/

There are studies focusing on high-level abstractions for energy saving on data parallelism [2, 17]. They provide compiler directives for expressing energy consumption and performance objectives in OpenMP. While Shafik et al. [17] can minimize energy consumption on both sequential and parallel applications, they do not provide any means to explicitly control the performance of the application. On the other hand, in Alessi et al. [2], OpenMPE is proposed adding a new construct and two clauses (objectives) for OpenMP. Their solution was implemented using a source-to-source compiler, which recognizes the new directives and control the number of threads used by OpenMP and applies DVFS to satisfy the SLOs expressed by the user. This is probably the closest work to the approach we are proposing in this work. The main difference is that, while Alessi et al. [2] targets batch applications (i.e. applications for which all the input data is already available in memory) implemented through OpenMP, we provide support for stream processing applications, exposing ad-hoc SLOs for these applications such as system utilization.

3 SPar: High-Level Stream Parallelism

SPar[4] is an internal Domain-Specific Language (DSL) designed to support high-level stream parallelism for application programmers [10]. With SPar, instead of rewriting the source code, the programmer introduces C++ annotations (standard `C++-11` [14]) using five attributes, representing the main properties of stream processing applications. The `ToStream` attribute identifies the beginning of a stream region, which can be viewed as an assembly line. The `Stage` attribute marks a stage in the assembly line and as many as necessary can be declared. Auxiliary attributes can be used inside the attribute list of an annotation sentence. The `Input` and `Output` respectively attributes are used to specify the input and output stream items, while the `Replicate` attribute is used for replicating stateless stages to increase the degree of parallelism.

Listing 1.1 provides a short code example annotated with SPar attributes. This example represents a typical use case of stream parallelism, where there is a sequence of operations to be performed on each stream element. The parallel activity graph produced by the SPar compiler for Listing 1.1 is shown in Figure 1. SPar generates the parallel code with the FastFlow library [1], which implements different parallel patterns [15] for stream processing computations. SPar compiler parses the code of Listing 1.1 and represents the code with an Abstract Syntax Tree (AST) [10]. Traversing the AST, it performs a semantic analysis of the attributes to further make the source-to-source transformation. In this step, SPar compiler finds the best parallel pattern that meets the parsed annotation schema. In the case of Listing 1.1, it will generate parallel code with three stages where one of them have replicated instances. There will be situations where there will be different compositions of stages and replicated instances. By default, elements are scheduled from the `ToStream` stage to the `Stage.x` stages in a round-robin way. However, it is possible to use an on-demand policy by specifying

[4] SPar website: https://gmap.pucrs.br/spar

the `-spar_ondemand` flag to the SPar compiler. If the data needs to be received from the last stage in the same order it was produced by the `ToStream` stage, the programmer can specify the `-spar_ordered` flag to the SPar compiler.

```
1  [[spar::ToStream]] while(1){
2    frame f = read_frame();
3    if(f.empty()) break;
4    [[spar::Stage,spar::Input(f),spar::
       Output(f),spar::Replicate(n)]]
5    for (int i=0; i<f.length(); i++) {
6      f[i] = convert(f[i]);
7    }
8    [[spar::Stage,spar::Input(f)]]{
9      write_frame(f);
10   }
11 }
```

Listing 1.1. SPar code example.

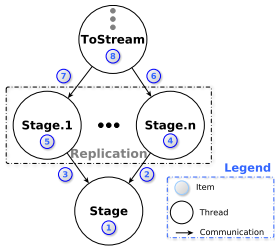

Fig. 1. Parallel activity graph.

4 Service Level Objective for Stream Parallelism

Service Level Objectives (SLOs) are traditionally included in Service Level Agreements (SLAs), which are contracts to manage the quality of service delivered by or received by a provider [18]. An SLA contract defines the acceptable levels of service by user and attainable levels of service by a provider. The SLO is a target value or a range of values for a certain level of service to be delivered and the level of service is measured by a Service Level Indicator (SLI). A typical structure of SLO can be written $SLI \leq target$ or $lower\_bound \leq SLI \leq upper\_bound$ [4]. When an SLO is violated, the system should autonomously react to guarantee the quality of service and SLA. Our design goal is to simplify the usability of SLO for stream parallel applications, on top of an existing parallel programming tool. Since SPar already provides high-level parallel programming abstractions and allows us to extend its annotations, we made our proof of concept on top of it. We will concentrate for the moment on three different SLOs (throughput, power, and utilization), which can be expressed by using standard `C++11` attributes that will be described in the following section.

4.1 Attributes

The proposed attributes have to be used along with a `ToStream` annotation, which identifies the beginning of a stream parallelism region, so that the SLO is applied to this particular region. Listing 1.2 presents how the code looks like when expressing a power consumption SLO of 60 watts. It is worth noting that, beside the `slo::Power` attribute, no other modification is required with respect to the original SPar code (Listing 1.1). The following list enumerates the attributes we added to SPar in this work, to support SLOs.

```
1  [[spar::ToStream,slo::Power(60)]] while
       (1){
2    frame f = read_frame();
3    if(f.empty()) break;
4    [[spar::Stage,spar::Input(f),spar::
       Output(f),spar::Replicate(n)]]
5    for (int i=0; i<f.length(); i++) {
6      f[i] = convert(f[i]);
7    }
8    [[spar::Stage,spar::Input(f)]]{
9     write_frame(f);
10   }
11 }
```

Listing 1.2. SPar code example with power consumption SLO.

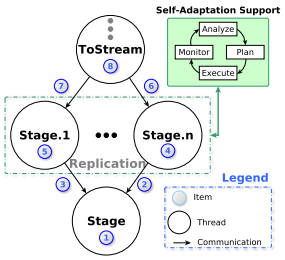

Fig. 2. Parallel activity graph with self-adaptation support.

slo::Power(<max-watts>) is the attribute used to specify the power consumption SLO. The user can specify the maximum power consumption in Watts. If no other attributes are specified, NORNIR will implicitly find the configuration with the highest throughput among those with a power consumption lower than `<max-watts>`.

slo::Throughput(<min-items/second>) is the attribute used to specify the application throughput SLO. The user can specify the minimum throughput required in items per second. If a power consumption SLO is not explicitly set, NORNIR will implicitly find the configuration with the lowest power consumption among those with a throughput greater than `<min-items/second>`.

slo::Utilization(min-percentage) is the attribute used to specify the application utilization SLO. The user can specify the minimum utilization required in percentage. Utilization represents the percentage of time that the system is active (i.e. actively processing input elements) over a time interval. Having a low utilization is often associated to a low power efficiency, since resources may be active but performing useless activities (i.e. actively waiting for new elements to be processed). If a power consumption SLO is not explicitly set, NORNIR will implicitly find the configuration with the lowest power consumption among those with a greater utilization number than `<min-percentage>`.

4.2 Implementation and Self-Adaptation Support with Nornir

In the SPar compiler, we registered the new SLO attributes and performed the semantic analysis traversing the source code AST. Since the SLO attributes belong only to the `ToStream` attribute list, we stored it along with the SPar AST [10]. In the source-to-source code transformation, we generate the same parallel patterns originally designed. However, we check if there is an SLO attribute to generate code with NORNIR in the situations where SPar annotation generates a stage with replicated instances.

To provide the specified SLO, we couple a self-adaptive runtime to the activity graph (Figure 2). In this work, we rely on the NORNIR self-adaptive runtime support [7]. NORNIR monitors the application throughout its entire execution, dynamically changing the number of resources used by the application to satisfy the requirements expressed by the user. For example, NORNIR may decide to reduce the number of replicated stages of the application to decrease its power consumption, or to increase the clock frequency of the cores to increase its throughput[5]. NORNIR can rely on different algorithms to decide how many resources to add/remove, either based on machine learning techniques [8] or on heuristics. In both cases, when the application starts, NORNIR spends some time in collecting data about the application in different configurations. These data are used to build prediction models which will be used to find the optimal configuration according to the objectives specified by the user. If no feasible solutions are found, NORNIR will select the configuration with performance and power consumption closest to the user requirements. More information about this algorithm can be found in [8].

Besides providing the possibility to control existing parallel applications (by inserting instrumentation calls in the existing code), NORNIR can also be used as a programming framework (by relying on the FASTFLOW framework) for implementing stream-parallel applications with an embedded self-adaptation support. In this work, we exploited this second possibility, so that SPar can translate sequentially annotated code into self-adaptive NORNIR parallel code.

While the integration with SPar allows to use NORNIR in a simple and transparent way, it is worth noting that NORNIR could also be used on other frameworks different from SPar.

5 Experiments

In this section, we evaluate our approach over some real-world applications. We will first introduce the considered applications. Then, we will compare the code generated by SPar with some handwritten parallel implementations, both regarding maximum performance achieved and in terms of productivity. Eventually, we will analyze the self-adaptation capabilities of our solution under different scenarios.

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 cores (12 per socket). The machine exposes 13 frequency levels, ranging from 1.2GHz to 2.4GHz, at steps of 0.1GHz. Each core has 2-way hyperthreading, 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 used Linux 3.14.49 x86_64 shipped with CentOS 7.1 and `gcc` version `4.8.5`. For all our experiments we disabled the hyper-threading feature.

[5] Since the number of replicas is dynamically changed during the execution, the number of replicas specified with the `spar::Replicate` attribute now represents the maximum number of replicas that can be active at any time.

5.1 Applications

In this section, we briefly describe the real-world application set, input loads, and parallel implementations. For a detailed description of how *Lane Detection* and *Person Recognition* have been parallelized by using SPar please refer to [12], while for *Pbzip2* more details can be found in [11].

Lane Detection is a video processing application to detect road lanes, implemented by using the OpenCV library. To introduce parallelism in the sequential code, we annotated it with SPar by identifying three stages: i) a first stage which reads the frames; ii) another stage, replicated a number of times, which processes the frames in parallel; iii) a last stage which displays the frames in the proper order, with the lanes properly marked. As input workload, we used a 22MB MPEG-4 video (640x360 pixels).

Person Recognition is an application used to recognize people in a video. The parallel structure of this application is similar to *Lane Detection*, with the middle stage detecting the faces from the crowd and searching in an image database to classify each face detected. As input workload, we used a 4.8MB MPEG-4 video (640x360 pixels) along with a training set of 10 face images of 150x150 pixels.

Pbzip application is a parallel implementation of the `bzip2` block-sorting files compressor[6]. This is a very coarse grained application characterized by a stream parallel programming model. We annotated the SPar version with three stages, where the middle stage is replicated. The input file to compress that we used for our experiments is a 6,3GB file containing a dump of all the abstract present on the English Wikipedia on 01/12/2015.

5.2 Comparison with Handwritten Implementations

Before evaluating the ability to satisfy SLO specified by the user we want to prove that, from a performance standpoint, the code generated by SPar is comparable with a handwritten implementation. On the other hand, we would like to show that our solution reduces the code intrusion required to transform a sequential application into a parallel one. As reference implementations for *Pbzip* we consider the original Pthreads version, while for *Lane Detection* and *Person Recognition* applications we consider the handwritten FastFlow versions described in [12].

Performance To measure the maximum performance, we executed both the reference and our solution generated versions by running them with 24 threads (to have at most one thread per core). For our generated version, we did not specify any SLO, but we still monitor the application by using Nornir. By doing so, we monitor both the overhead introduced by the interaction with the self-adaptive support and possible inefficiencies in the generated code. As shown by

[6] http://compression.ca/pbzip2/

	Pbzip2	Lane Detection	Person Recognition
Performance Improvement (%)	+0.48%	−1.45%	−0.92%
LOC Reduction (%)	−15.86%	−21.51%	−24.49%

Table 1. Performance improvement with respect to a handwritten implementation. Negative values mean that SPar version is slower than the handwritten one. For LOC Reduction, negative values mean that SPar version is more concise than the handwritten one.

the results in Table 1, for *Lane Detection* and *Person Recognition*, the overhead is negligible (below 1.5%). For *Pbzip2*, there is a slight improvement caused by the use of FastFlow as runtime support, while the reference implementation was based on Pthreads.

Code Intrusion To measure the code intrusion, we rely on Lines of Code (LOC) metric. Despite that LOC is not universally accepted, it is commonly used to compare different implementations of the same application [20]. For our measurements, we only considered the source files containing the code relevant for the parallelization. In all the cases, parallelizing an application by using SPar requires a lower code intrusion with respect to Pthreads or FastFlow [10, 11], since it usually only requires introducing some annotations in the already existing sequential code. Also, the SLOs attributes requires minimal effort.

5.3 Self-Adaptation Analysis

To analyze the self-adaptation capabilities of the parallel code automatically generated by our solution, we first require the application to have a greater throughput number than the sequential version while minimizing the power consumption.

The target of this first experiment is to prove that parallelization is not only useful for improving the performance of an application, but it can also be used to reduce its power consumption. In a nutshell, we want to prove that a parallel application with the same performance of the sequential one has lower power consumption. We show the results of this test in Table 2.

The interpretation we would like to give to these results is that, even if the performance of a sequential application is satisfactory, parallelizing it may still be useful for reducing its power consumption. This effect occurs since by increasing the number of replicas (and thus the number of cores used by the application), we can reduce the clock frequency while keeping the same performance. Since the power consumption increases linearly with the number of cores but more than quadratically with the clock frequency [5], running an application on more cores at a lower frequency is usually more energy efficient than running it on fewer cores at a higher frequency. Having tools and methodologies for doing

	Pbzip2	Lane Detection	Person Recognition
Power Consumption Reduction (%)	−9.43%	−10.37%	−7.39%

Table 2. Power consumption reduction obtained by a parallel application with the same throughput of the sequential one.

that automatically and with low code intrusion, like those we are describing in this work is of paramount importance for enabling such techniques in real-world scenarios.

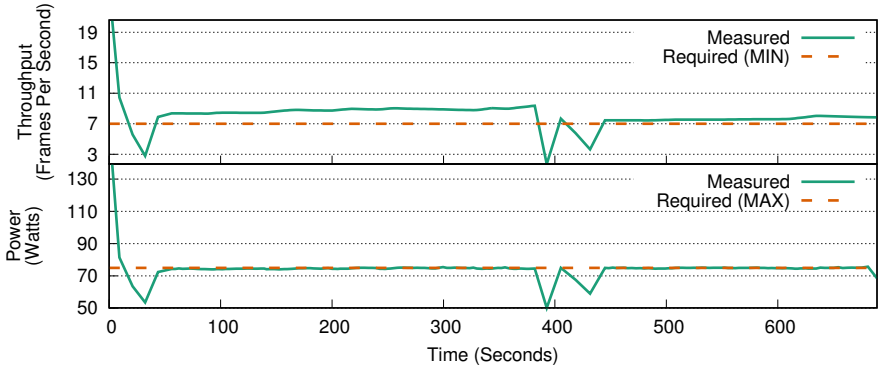

Fig. 3. *Person Recognition* application with `slo::Throughput(7)` and `slo::Power(75)`.

We now analyze the time behavior of different applications for different types of SLO. In Figure 3, we show how throughput and power change in time when the user requires a greater throughput number than 7 frames per seconds and a power consumption lower than 75 watts for the *Person Recognition* application. In the first 40 seconds of the execution, our runtime changes the configuration a few times to collect some data which will then be used to predict the best configuration according to the user requirements. This behaviour depends from the specific algorithm we used in Nornir and other algorithms, which avoid such fluctuations could be used as well. Around 390 seconds from the beginning, the application enters a different phase of its execution, and our runtime needs to update the prediction models by collecting new data. This different phase impacts in the throughput and power consumption fluctuations, occurring around 400 seconds from the beginning.

In Figure 4 we analyze a different scenario, where the user requires a greater throughput number than 20 blocks per second and power consumption lower than 65 watts for the *Pbzip* application. In this test, we add some external noise

to show that our runtime succeeds in providing the required SLO even in the presence of unexpected behaviors. In particular, besides the usual calibration done in the first seconds of execution, after 50 seconds from the start of *Pbzip*, we start another application on the same machine. Since the two applications share some resources (*i.e.*, cores, memory, among others), the throughput of *Pbzip2* starts to decrease. In response to this issue, our runtime recomputes the prediction models, now considering the presence of external interference. As a consequence, as we can see from the bottom part of Figure 4, our runtime increases the number of replicas of the middle stage from 12 to 14. When the other interfering application terminates (around 120 seconds from the start of *Pbzip2*), our runtime recomputes the models and decreases the number of replicas from 14 to 13. As we can see from the two upper parts of the figure, our runtime satisfies the user requirements throughout the entire execution (excepts for the phases where the models are computed), independently from the presence of other applications running on the system.

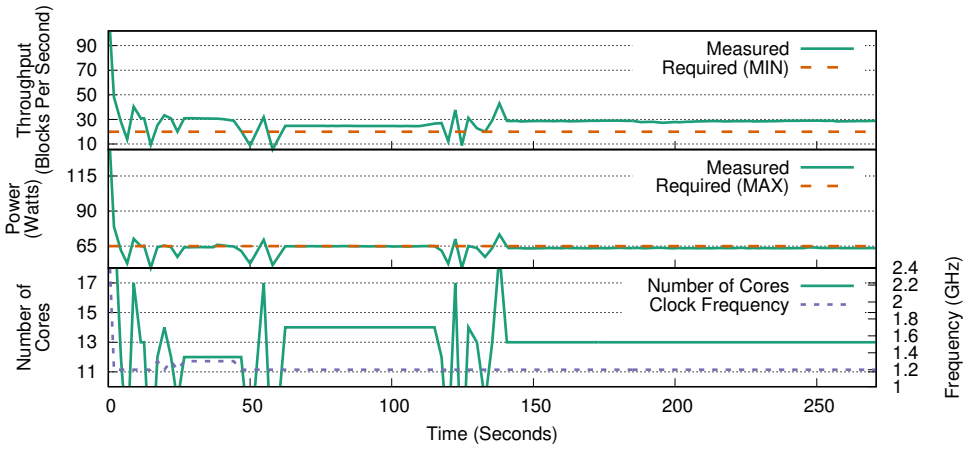

Fig. 4. *Pbzip2* application with `slo::Throughput(20)` and `slo::Power(65)`.

In the last experiment, which we report in Figure 5, we analyze the *Lane Detection* application, in a scenario where it produces no more than 50 frames per seconds. In such case, using all the available resources could be inefficient, since they could be idle for most of the time. To avoid such scenario, we set a utilization SLO of 80%. In the upper part of Figure 5, we report the utilization when an SLO is specified and when it is not specified. In the bottom part, we report the power consumption. As shown by the result when an SLO is not specified, the utilization would be around 20%. This utilization means that the threads of the application would spend 80% of the time waiting for new frames to arrive. By requiring a minimum utilization of 80%, our runtime decreases the number of resources allocated to the application, decreasing the power consumption from 90

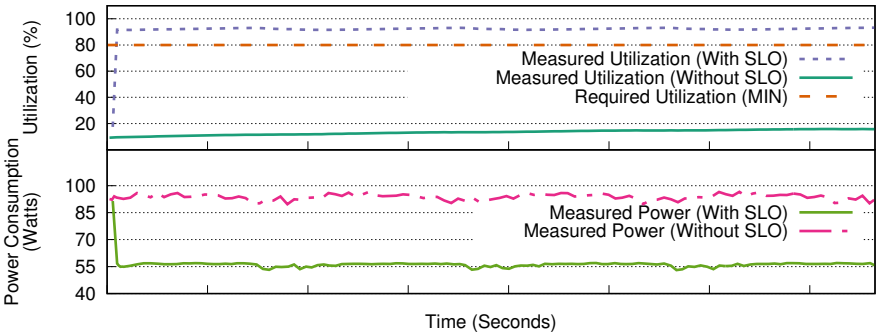

Fig. 5. *Lane Detection* application with `slo::Utilization(80)`.

watts to 55 watts. This event occurs without decreasing the overall performance of the application. Indeed, the threads still spend some time waiting for new data, but it is reduced from 80% to 5% (the utilization is around 95%).

6 Conclusions and Future Work

In this paper we provided the possibility to express SLO on the sequential code, using automatic parallelization and self-adaptation of resources such as the number of replicas (and, consequently, the number of cores) and clock frequency. We described how we extended the SPar source-to-source compiler to support different types of SLO and we performed an experimental evaluation showing the effectiveness of our approach. The results demonstrated that by using self-adaptation, under certain conditions, we reduced the power consumption up to 42%. Also, we reduced the power consumption by 9.06% while not decreasing the performance with respect to the sequential version. Lines of code are reduced 20% in average with respect to a handwritten implementation, which shows the simplicity of our solution.

As a future work, we intend to consider other types of SLO such as execution time and energy consumption (*i.e.*, integral of power consumption over time). Moreover, we would like to extend our work by considering more applications and different or more heterogeneous workloads.

Acknowledgement

This work has been partially supported by the EU H2020-ICT-2014-1 project RePhrase (No. 644235), FAPERGS 01/2017-ARD project ParaElastic, and CAPES scholarships.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: FastFlow: High-Level and Efficient Streaming on Multi-core. In: Programming Multi-core and Many-core Computing Systems. PDC, vol. 1, p. 14. Wiley (2014)

2. Alessi, F., Thoman, P., Georgakoudis, G., Fahringer, T., Nikolopoulos, D.S.: Application-Level Energy Awareness for OpenMP. In: International Workshop on OpenMP. pp. 219–232. Springer (2015)
3. Andrade, H.C.M., Gedik, B., Turaga, D.S.: Fundamentals of Stream Processing. Cambridge University Press, New York, USA (2014)
4. Beyer, B., Jones, C., Petoff, J., Murphy, N.R.: Site Reliability Engineering. O'Reilly, Boston, USA (2016)
5. Chandrakasan, A.P., Brodersen, R.W.: Minimizing Power Consumption in Digital CMOS Circuits. Proceedings of the IEEE 83(4), 498–523 (1995)
6. Danelutto, M., Garcia, J.D., Sanchez, L.M., Sotomayor, R., Torquati, M.: Introducing Parallelism by Using REPARA C++11 Attributes. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 354–358. IEEE (2016)
7. De Sensi, D., De Matteis, T., Danelutto, M.: Simplifying Self-Adaptive and Power-Aware Computing with Nornir. Future Generation Computer Systems pp. – (2018)
8. De Sensi, D., Torquati, M., Danelutto, M.: A Reconfiguration Algorithm for Power-Aware Parallel Applications. ACM Transactions on Architecture and Code Optimization 13(4), 43:1–43:25 (dec 2016)
9. Floratou, A., Agrawal, A., Graham, B., Rao, S., Ramasamy, K.: Dhalion: Self-Regulating Stream Processing in Heron. Proceedings of the VLDB Endowment 10, 1825–1836 (2017)
10. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: SPar: A DSL for High-Level and Productive Stream Parallelism. Parallel Processing Letters 27(01), 20 (2017)
11. Griebler, D., Filho, R.B.H., Danelutto, M., Fernandes, L.G.: High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. International Journal of Parallel Programming pp. 1–19 (2018)
12. Griebler, D., Hoffmann, R.B., Danelutto, M., Fernandes, L.G.: Higher-Level Parallelism Abstractions for Video Applications with SPar. In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing. pp. 698–707. ParCo'17, IOS Press, Bologna, Italy (2017)
13. Maggio, M., Hoffmann, H., Santambrogio, M.D., Agarwal, A., Leva, A.: Controlling Software Applications via Resource Allocation within the Heartbeats Framework. In: IEEE Conference on Decision and Control. pp. 3736–3741. IEEE (2010)
14. Maurer, J., Wong, M.: Towards Support for Attributes in C++ (Revision 6). Tech. rep., The C++ Standards Committee (2008)
15. McCool, M., Robison, A.D., Reinders, J.: Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann, MA, USA (2012)
16. Reinders, J.: Intel Threading Building Blocks. O'Reilly, USA (2007)
17. Shafik, R.A., Das, A., Yang, S., Merrett, G., Al-Hashimi, B.M.: Adaptive Energy Minimization of OpenMP Parallel Applications on Many-Core Systems. In: Parallel Programming and Run-Time Management Techniques. pp. 19–24 (2015)
18. Sturm, R., Morris, W., Jander, M.: Foundations of Service Level Management. SAMS, Boston, USA (2000)
19. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A Language for Streaming Applications. In: Proceedings of the International Conference on Compiler Construction. pp. 179–196. Springer, Grenoble, France (2002)
20. Weyuker, E.J.: Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering 14(9), 1357–1365 (1988)