Noname manuscript No.
(will be inserted by the editor)

Simplifying and Implementing Service Level Objectives for Stream Parallelism

Dalvan Griebler · Adriano Vogel · Daniele De Sensi · Marco Danelutto · Luiz G. Fernandes

Received: date / Accepted: date

Abstract An increasing attention has been given to provide Service Level Objectives (SLOs) in stream processing applications due to the performance and energy requirements, and because of the need to impose limits in terms of resource usage while improving the system utilization. Since the current and next generation computing systems are intrinsically offering parallel architectures, the software has to naturally exploit the architecture parallelism. Implement and meet SLOs on existing applications is not a trivial task for application programmers, since the software development process, besides the parallelism exploitation, requires the implementation of autonomic algorithms or strategies. This is a system-oriented programming approach and requires the management of multiple knobs and sensors (*e.g.*, the number of threads to use, the clock frequency of the cores, etc.) so that the system can self-adapt at run-time. In this work, we introduce a new and simpler way to define SLO in the application's source code, by abstracting from the programmer all the details relative to self-adaptive system implementation. The application programmer specifies which parts of the code to parallelize and the related SLOs that should be enforced. To reach this goal, source-to-source code transformation rules are implemented in our compiler, which automatically generates self-adaptive strategies to enforce, at run-time, the user-expressed objectives. The experiments highlighted promising results with simpler, effective, and efficient SLO implementations for real-world applications.

Keywords Parallel Programming · Stream Processing · Self-Adaptive · Domain-Specific Language · Power-aware Computing

Dalvan Griebler, Adriano Vogel, Luiz G. Fernandes
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS)

Daniele De Sensi and Marco Danelutto
Department of Computer Science, University of Pisa (UNIPI)

Dalvan Griebler
Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM) E-mail: dalvan.griebler@acad.pucrs.br

1 Introduction

Service-oriented approach influenced new system models like the cloud computing one [7]. Since a service can be represented by software components, functions, or a sequence of commands, this can help to improve the expressiveness and methodology of parallel software design. The service behavior can also be evaluated from different perspectives through the concept of Quality-of-Service (QoS), which identifies non-functional attributes. Performance metrics like throughput are used to measure QoS or to establish requirements between providers and clients. The programmer may know about the requirements of the components and how to improve the QoS of them. Consequently, the programmer can define Service Level Objectives (SLOs) for each one of these components so that they behave as expected in a Service Level Agreement (SLA) or in a contract [33].

Since the new and next-generation computing systems are intrinsically offering parallel architectures, the software has to naturally exploit the architecture parallelism. Implement and meet SLOs on existing applications is not a trivial task for application programmers, because the usual software development process, besides the parallelism exploitation, also requires the implementation of autonomic algorithms or strategies. This is a system-oriented programming approach and requires the management of multiple knobs and sensors (*e.g.*, the number of threads to use, the clock frequency of the cores, etc.) simultaneously so that the system self-adapts at run-time.

In stream processing applications, parallelism is typically exploited by using linear or non-linear pipeline pattern compositions [27]. To this purpose, parallel programming framework such as StreamIt [35], Intel TBB [30], and FastFlow [2,1] provide different programming approaches and interfaces with a reasonable performance scalability for this domain. Although these frameworks are equipped with high-level pattern implementations 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 [17] was created. It improves application programmers' productivity through a `C++11` annotation-based language, which does not require to rewrite/restructure the sequential source code [17]. Moreover, it is important to highlight that stream processing applications are usually characterized by unpredictable load fluctuations and uncertain end of execution (may never end) [4]. However, none of these parallel programming alternatives automatically deals with this service-oriented behavior, since they are not able to guarantee SLOs due to the static resource assignment (*e.g.*, a fixed amount of threads).

Besides the need for improving performance through the efficient exploitation of the multi-core parallelism, there are also other major concerns such as power-aware computing and efficient resource usage [16,25]. To address these needs, the Nornir framework was created, providing runtime support to dynamically and automatically control the resources allocated for the application according to the user needs [12]. However Nornir, like most existing self-adaptive solutions, only works on parallel applications and requires sequential code refactoring.

To simplify the specification of SLOs in sequential stream processing applications, differently from NORNIR and state-of-the-art parallel programming frameworks, we proposed a set of SLO attributes which can be inserted along with SPAR's stream parallelism annotations in the sequential code. In addition to that, we introduce a programming methodology where the programmer specifies which source code regions can be parallelized and the requirements that should be enforced. We implemented source-to-source code transformation rules in the SPAR compiler to automatically generate the self-adaptive strategies that enforce the user-expressed objectives at run-time. The proposed energy-aware SLO attributes were implemented using NORNIR's runtime support and studied in the previous work [18]. Our approach could also be applied to other frameworks, for example to the framework designed by the REPARA project[1], which provides a set of C++11 attributes to introduce generic parallelism [11]. Moreover, we implement some SLOs on NORNIR and others on top of FASTFLOW, to demonstrate that this can be applied to different runtimes, with a different implementation complexity according to the used abstraction. The major contributions of this paper are summarized as follows:

– A new set of SLO attributes semantically defined by using standard C++11 and SPAR annotations.
– Design and implementation of new self-adaptive strategies using the FASTFLOW framework.
– The implementation of the new SLOs in the SPAR compiler with source-to-source transformation rules, targeting self-adaptive strategies with NORNIR and FASTFLOW back-ends.
– An experimental evaluation with real-world applications, comparing our implementations with some state-of-the-art solutions.

We structured our paper as follows. We first present the related work in Section 2. The next section (3) describes SPAR. Section 4 details the proposed SLOs for stream parallelism and its implementation. In Section 5, a set of experiments are analyzed and discussed. Finally, Section 6 makes the conclusions of the paper.

2 Related Work

In the literature, there are different studies targeting power consumption, throughput, and system utilization objectives. Among them, the approach of Maggio et al. [25] 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.

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

Some existing algorithms do not explicitly model the power consumption of applications, thus only providing the possibility to specify performance SLOs [14,24]. In some cases, it is not even possible to enforce a specific performance requirement, but only to run the application in the most efficient [32] or the most performing configuration [29,10,34]. Other works provide SLO on power consumption and/or application performance by acting on mechanisms different from those considered in this work, such as caches [37] or network interfaces [36]. Another alternative approach to the presented problem is to dynamically change the accuracy of the results computed by the application according to the user SLOs. This technique is known as *approximate computing* [22,38,5] and can be used to trade an increase on performance (or a decrease on power consumption) for a decrease on the quality of the results computed by the application. However, these approaches are usually focused on algorithms and techniques to provide SLOs rather than on programming abstractions to express SLOs, as we do in this work.

Concerning stream parallel processing for real-time data analytic, Floratou et al. [16] 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 processes 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 six target SLOs to be expressed in sequential source codes.

Some works focus on high-level abstractions for energy saving on data parallelism [3,31], by providing compiler directives for expressing energy consumption and performance objectives in OpenMP. While Shafik et al. [31] 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. [3], 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 controls 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. [3] target 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.

Eventually, many existing solutions are either simulated or validated on post-mortem data (i.e. they are executed after the application finished its execution, in a *what-if* analysis fashion) [28,24,34,15]. We believe that, although a simulation may provide a first approximation about the precision that the algorithm could have in enforcing the required SLOs, it would not take into account the run-time overhead of these methods. Differently from these works, in this paper, we describe and implement a solution which has been validated by controlling real applications throughout their execution. Finally, stream processing applications vary during the execution without a pre-defined end, which makes such approaches unfeasible to apply in our application domain.

3 SPar: High-Level Stream Parallelism

SPar[2] is a Domain-Specific Language (DSL) designed to support high-level stream parallelism for application programmers [17]. With SPar, instead of rewriting the source code, the programmer introduces C++ annotations (standard `C++-11` [26]) 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 workstation in this assembly line, which can be composed by as many as necessary. Auxiliary attributes can be used inside the attribute list of an annotation sentence. The `Input` and `Output` attributes are used to specify the input and output stream items respectively, while the `Replicate` attribute is used for replicating stateless stages to increase the degree of parallelism.

Listing 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 is shown in Figure 1. SPar generates the parallel code with the FastFlow library [1], which implements different parallel patterns [27] for stream processing computations. The SPar compiler parses the code of Listing 1 and represents the code with an Abstract Syntax Tree (AST) [17]. Traversing the AST, it performs a semantic analysis of the attributes to further make the source-to-source transformations. In this step, the SPar compiler finds the best parallel pattern that meets the parsed annotation schema. In the case of Listing 1, it will generate parallel code with three stages, where the middle one is replicated. Moreover, different compositions with sequential or replicated stages can be achieved. By default, elements are scheduled from the `ToStream` stage to the `Stage` in a round-robin way. However, it is possible to use an on-demand policy by specifying 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.

```cpp
[[spar::ToStream]] while(1){
  frame f = read_frame();
  if(f.empty()) break;
  [[spar::Stage,spar::Input(f),spar::Output(f),spar::Replicate(n)]]
  for (int i=0; i<f.length(); i++) {
    f[i] = convert(f[i]);
  }
  [[spar::Stage,spar::Input(f)]]{
    write_frame(f);
  }
}
```

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

Listing 1 SPar example: image processing representation with stream parallelim.

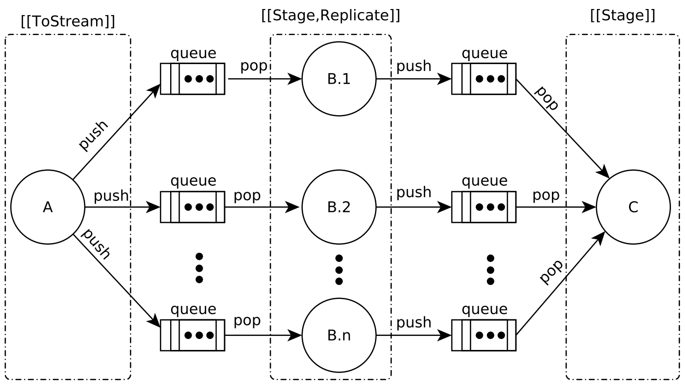


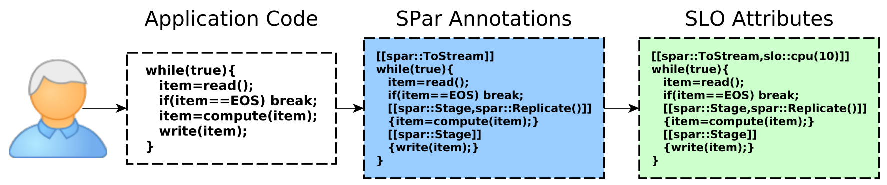
Fig. 1 SPar runtime: activity graph and communication queues.

Note that the `Replicate` attribute applies the replication role over the `Stage` in Figure 1. Each replicated stage has its own input and output lock-free queues. The first stage executes the code inside the `ToStream` region, which generates stream items for the subsequent stages. In the default configuration of SPar runtime, the stages actively try to push or pop stream items from the queues. If the queue is full or empty, the stage thread executes an active loop, trying to push or pop until it eventually succeeds. Every time that a given stage fails in perform push or pop, the stage generates a push or pop lost event. This may generate an extra overhead for coarse-grain computations. Therefore, users may set the SPar runtime to behave in a blocking mode through the *spar_blocking* compiler flag. In this case, the stage thread will not stay in a loop, it will wait until it can perform push or pop in the shared queue.

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 (QoS) established between customers and providers [33]. An SLA contract defines the level of service which is acceptable by the user and attainable by the provider. The SLO is a target value or a range of values for a certain level of service to be delivered. The level of service is measured by a Service Level Indicator (SLI). A typical structure of SLO can be written $SLI \leq target$ or $lower\_bound \leq SLI \leq upper\_bound$ [6]. When an SLO is violated, the system should react to guarantee the quality of service and SLA. Our design goal is to simplify the usability of SLO in stream processing applications.

Figure 2 depicts our proposed methodology to express SLOs in the application source code. The first step in the developing process is to code the stream processing application (not needed for legacy applications). After, the programmer inserts the SPar annotations to express the stream parallelism. This can be done following the recommendations of SPar's annotation methodology [17]. Lastly, the programmer can insert SLO attributes along with SPar's annotations in the source code. Therefore, the only requirement is to choose the SLO metric and its initial target value. No extra details must be provided by the application programmers, which can spend most of their time in coding the sequential application. Consequently, in this work, we support the application programmers with an opportunity to express stream parallelism with SPar and define a target QoS through SLO attributes.



```
while(true){
   item=read();
   if(item==EOS) break;
   item=compute(item);
   write(item);
}
```

```
[[spar::ToStream]]
while(true){
   item=read();
   if(item==EOS) break;
   [[spar::Stage,spar::Replicate()]]
   {item=compute(item);}
   [[spar::Stage]]
   {write(item);}
}
```

```
[[spar::ToStream,slo::cpu(10)]]
while(true){
   item=read();
   if(item==EOS) break;
   [[spar::Stage,spar::Replicate()]]
   {item=compute(item);}
   [[spar::Stage]]
   {write(item);}
}
```

Fig. 2 Our methodology to define SLOs for stream parallelism.

The SLO attributes are proposed to be used along with a `ToStream` annotation, which identifies the beginning of a stream parallelism region. Therefore, the SLO is applied to this particular region. Listing 2 demonstrates the definition of a power consumption SLO of 60 watts in line. It is worth noting that besides the `slo::Power` attribute, no other modification is required with respect to the original SPar code (Listing 1). While multiple SLOs attributes could be used together, there are only a few meaningful combinations. Usually, the user may need to express one SLO on performance and one SLO on power consumption. Other combinations are possible (e.g. `slo::Throughput` and `slo::Utilization` at the same time), but since they are two different representations of performance, they could conflict between each other. Additionally, adding too many constraints could lead to situations where there would be no feasible solutions, and it may be complex for the application programmer to find the right SLO values. Table 1 describes the SLO attributes proposed in this work. The attributes belong to the `slo` namespace and accept one argument, which is a value defining the target SLO.

```
[[spar::ToStream,slo::Power(60)]] while(1){
  frame f = read_frame();
  if(f.empty()) break;
  [[spar::Stage,spar::Input(f),spar::Output(f),spar::Replicate(n)]]
  for (int i=0; i<f.length(); i++) {
    f[i] = convert(f[i]);
  }
  [[spar::Stage,spar::Input(f)]]{
    write_frame(f);
  }
}
```

Listing 2 SPar code example with power consumption SLO.

Name	Argument	Description
`slo::Throughput`	`(min-items)`	The user can specify the minimum throughput required in items per second. The respective environment variable is `SLO_THROUGHPUT`.
`slo::Power`	`(max-watts)`	The user can specify the maximum power consumption in Watts. The respective environment variable is `SLO_POWER`.
`slo::Utilization`	`(min-%)`	The user can specify the minimum runtime system utilization required in percentage (from 1 to 100). In our case, it represents the percentage of time that the system is active (i.e. actively processing input elements) over a time interval. The respective environment variable is `SLO_UTILIZATION`.
`slo::Latency`	`(max-time)`	The user can specify the maximum latency in milliseconds. This latency refers to the time taken for an item passing from a stage to another one. The respective environment variable is `SLO_LATENCY`.
`slo::CPU`	`(max-%)`	The user can specify the maximum CPU utilization in percentage (from 1 to 100%). The respective environment variable is `SLO_CPU`.

Table 1 SLO attributes for SPar.

4.1 SLO Implementations

SLO attributes in SPar were initially proposed in our previous work [18]. In this paper, we extend that work by providing new self-adaptive strategies and SLO attributes (`slo::CPU`, `slo::Latency`). Moreover, we add a new algorithm for enforcing the `slo::Throughput` SLO when it is not combined with power consumption SLO. The compiler will decide which strategy to generate when performing the source-to-source code transformations.

We first explain the self-adaptive strategies to meet the so-called energy-aware SLOs, which rely on the Nornir runtime support [12]. 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 the application throughput.

Moreover, Nornir can rely on different algorithms to decide how many resources to add/remove, either based on machine learning techniques [13] or on heuristics. When machine learning techniques are used, when the application starts, Nornir performs a lightweight training phase by testing different configurations and collecting application data/performance indicators. The results collected are used to build prediction models which are used to find the optimal configuration according to the objectives specified by the user. If no feasible solution is found, Nornir selects the resources configuration characterized by performance and power consumption as close as possible to the user requirements.

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 embedded self-adaptation support. We exploited this second possibility so that SPAR can translate sequentially annotated code into self-adaptive NORNIR parallel code. All details relative to the use of NORNIR are abstracted and made simple along with the stream parallelism.

In addition to the NORNIR's strategies, we also provide new self-adaptive strategies for `slo::CPU`, `slo::Latency`, and `slo::Throughput`, which rely on the SPAR runtime system (which is built on top of FASTFLOW), as illustrated in Figure 3. Observe that this activity graph is a way of simplification from the SPAR runtime system presented in Figure 1. The strategy follows the MAPE approach [23] with a feedback closed-loop [21]. The Monitor entity periodically collects data from the sensors, which can be originated from the application or from the operating system. These data are used by the Analyze phase, which interprets the data and extracts relevant statistics.

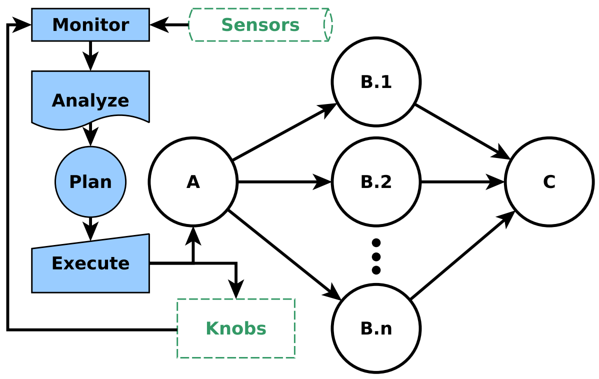


Fig. 3 SLO implementation by using sef-adaptive strategies.

Afterward, the Plan phase decides if the SPAR runtime system must be adapted to meet the specified SLO. It may be impossible for a strategy to achieve a given SLO. In such a case, the strategy attempts to reach the SLO as close as possible. Since stream processing applications may have load fluctuations, unnecessary adaptations should be avoided. In our strategies, we used a threshold, which is a percentage number that can be tolerated when the actually monitored metric is higher but close to the target SLO. The default threshold is 20%, such value was ascertained in [39] as a suitable one for stream processing applications. Moreover, for the sake of flexibility, users may customize the threshold value using the `SLO_THRESHOLD` environment variable. The Execute entity receives the planned action and applies the adaptation by sending instructions to the SPar runtime system (*e.g.*, tasks/items distribution) and system knobs (*e.g.*, adapt the number of active replicas). Although we used this same idea for implementing all new SLOs, each SLO has its specific self-adaptive strategy (*i.e.*, Analyze and Plan phases), described in the following section.

4.1.1 `slo::Latency`

In a previous work [39], we have shown the possibility to manage the latency by adapting the number of replicas. In this work, we extend the previous study by implementing it in the SPar compiler as well as providing an SLO option. During our study, we have seen how the number of replicas affects the latency of stream items. Additionally, it is a presumably difficult task for programmers to manually adapt their software at run-time based on the latency SLO constraints and on the actual application latency. As a consequence, we aim at abstracting from programmers the impact of the number of replicas in latency.

We implemented a strategy for the SPar's runtime that monitors and manages the latency of stream items by autonomously adjusting the number of replicas. Considering the representation in Figure 3, the stage A adds a timestamp to the stream items and the Monitor entity collects from a sensor that is inside the stage C, where the latency of the stream items is measured. In the Analyze and Plan phases, the latency information from the Monitor entity is compared to the SLO, and the Plan phase decides whether to change or not the number of replicas, based on the tolerated threshold. Eventually, Execute entity sends instructions to control knobs which changes the number of active replicas and stage A which distributes the items among the active replicas from the stage B.

4.1.2 `slo::Throughput`

Our self-adaptive strategy for the `slo::Throughput` SLO is based on the number of items processed per second. The self-adaptive strategy is able to increase or decrease the throughput with the number of replicas control knob. In this SLO implementation, accordingly to Figure 3, the Monitor entity periodically gets the number of items per second from the sensor that we installed in stage C. For each iteration, the throughput is the result of dividing the number of processed items by the time that it was taken. Throughput rates are then stored and accessed by the Analyze phase, which provides useful data statistics to the Plan phase decide if an adaptation is required.

The Plan phase also has a maximum value for the number of replicas, which is defined according to the machine's processing capabilities, gathered by another sensor that extracts hardware information. The Execution entity is updated by the Plan phase in order to send information to the control knob, which increases or decreases the number of replicas depending on the need. In addition, the Execution entity will inform the stage A (Figure 3), which implements the task scheduler in the SPar runtime system.

4.1.3 `slo::CPU`

In the stream processing domain, several SLOs can be relevant for defining performance/efficiency objectives. This occurs because in stream processing applications, differently from other application domains, the maximum amount of resources available are not always used nor needed. Continuously use the total resources capacity tends to reduce the efficiency of the system. Also, using the maximum resources does not actually mean that a stream processing application will achieve the best performance [13]. Performance is complex in the stream processing domain because the workload trend varies in a timely fashion according to variable input rates, volumes, resources availability, and different performance objectives. Consequently, we are employing efforts to enforce performance goals for enabling a customizable execution of stream processing applications and their unique characteristics. It is also relevant to allow programmers to define objectives regarding the consumption of resources.

Therefore, we provide an option to define the CPU utilization (`slo::CPU`) SLO when running a given application. Although there are available OS-level tools for controlling the CPU usage (*e.g.,* CPUlimit [9]), such tools are arguably not flexible. Considering the dynamic nature of stream processing applications, we expect to adapt the degree of parallelism of the application at run-time for optimizing the CPU utilization and meet the target SLO. The implemented self-adaptive strategy follows the schema sketched in Figure 3.

Differently from the previous SLOs, the Monitor entity is periodically getting the current CPU utilization from the sensor, which is reading it from the operating system. The Analyze phase calls the Monitor entity for providing CPU utilization statistics to the Plan phase, which aims to decide whether the number of replicas should be increased or decreased. To avoid oscillation and instability regarding the replicas reconfiguration, a threshold (described in Section 4.1) value is used so that the number of replicas is not increased when the utilization is close to the SLO. Finally, the Execution entity simply sends this information to the system knob apply an action (increase, decrease, or stay as it is) as well as to the task scheduler in the stage A to manage stream item in a correct manner.

4.2 Source-to-Source Transformations

Self-adaptive strategies for each SLO attribute are automatically generated during the program compilation. We used the SPar compiler to implement the source-to-source code transformations. This required to add a new compilation step inside the compiler when performing the transformations from the SPar annotations to parallel patterns. In this step, the compiler builds the SPar runtime system with the communications, scheduling, and synchronizations. Based on the semantics previously specified, we added semantics-checking for the SLO attributes to ensure correct code generation. All transformations and analysis are performed in the AST and the parallel code generation is based on transformation rules [17]. In addition to that, we also implemented transformation rules to generate the appropriate self-adaptive strategy for each SLO attribute annotated in the source code.

Figure 4 depicts a high-level representation of the source-to-source transformations targeting the proposed SLOs. This occurs after the semantic analysis where the Annotation Abstract Syntax Tree (AAST) is built from the source codes. The AAST also contains an internal representation of the SLO attributes. The compiler checks if SLO attributes were specified in the specific `spar::ToStream` node. In this case, the appropriate rules are applied as shown in Figure 4.

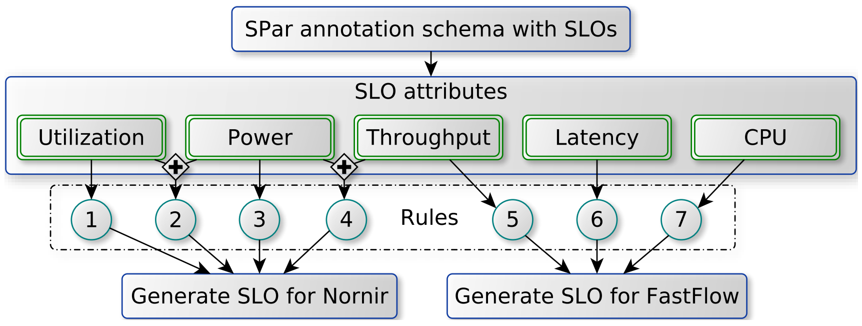


Fig. 4 Source-to-source transformation for the SLO attributes with SPar.

As presented in Figure 4, we implemented seven transformation rules to implement the proposed SLO attribute declarations. The first four rules will generate parallel code with SLO strategies to be executed with the Nornir framework's back-end. On the other hand, the last three rules will generate parallel code with the new SLO strategies (proposed in this paper) for the FastFlow framework. When using FastFlow's back-end, the task scheduler thread also hosts the self-adaptive strategies. On the other hand, in Nornir, this is managed by an extra thread. Besides, Nornir and FastFlow have different programming interfaces, the parallel patterns are conceptually similar. Modifications were only necessary to accommodate the proper routine names. Considering that parallel patterns were implemented in the previous source-to-source transformation step based on the rules already designed in [17], here we concentrate on the transformation rules related to the SLOs. It is important to note that here we describe only the meaning of the transformations required, the implementation details are arguably not relevant for presenting a simplified description. The transformations performed are the following:

1. Implement the following transformation steps for the `slo::Utilization` attribute: a) insert the routine which implements the SLO utilization strategy in the Nornir library before the declaration of `spar::ToStream`; and b) give as a parameter the attribute argument to be the target SLO for the strategy routine.

2. Implement the following transformation steps for the `slo::Utilization` and `slo::Power` attributes: a) insert the routine which implements the SLOs utilization and power strategies in the Nornir library before the declaration of `spar::ToStream`; and b) give as parameters the attribute arguments to be the target SLO for the strategy.
3. Implement the following transformation steps for the `slo::Power` attribute: a) insert the routine which implements the SLO power strategy in the Nornir library before the declaration of `spar::ToStream`; and b) give as a parameter the attribute argument to be the target SLO for the strategy routine.
4. Implement the following transformation steps for the `slo::Throughput` and `slo::power` attributes: a) insert the routine which implements the SLOs throughput and power in the Nornir library before the declaration of `spar::ToStream`; and b) give as parameters the attribute arguments to be the target SLOs for the strategy routine.
5. Implement the following transformation steps for the `slo::Throughput` attribute: a) insert the routine which implements the SLO throughput strategy in the FastFlow library before the declaration of `spar::ToStream`; and b) give as a parameter the attribute argument to be the target SLO for the strategy routine.
6. Implement the following transformation steps for the `slo::Latency` attribute: a) insert the routine which implements the SLO latency strategy in the FastFlow library before the declaration of `spar::ToStream`; and b) give as a parameter the attribute argument to be the target SLO for the strategy routine.
7. Implement the following transformation steps for the `slo::CPU` attribute: a) insert the routine which implements the SLO CPU utilization strategy in the FastFlow library before the declaration of `spar::ToStream`; and b) give as a parameter the attribute argument to be the target SLO for the strategy routine.

After the SLO transformation rules were applied, the parallel pattern generated in the AST is built with these rules' configurations, either for the Nornir's self-adaptive manager or for the implemented SPar's manager. The SPar's manager runs a MAPE feedback closed-loop in the generated SPar's task scheduler, which is on top of the FastFlow library. We also support the `-spar_blocking` and `-spar_ordered` compilation flags that are natively supported in SPar (see Section 3). These compilation flags were used for the experiments in the next section.

5 Experiments

In this section, we first introduce the considered real-world applications. Then, we will compare the code generated by SPar with handwritten parallel implementations for these applications, both regarding maximum performance achieved and productivity. Also, we analyze the self-adaptation capabilities automatically generated by the SPar compiler under different scenarios. The experiments have been executed in the following two machines:

– **M1** is a machine equipped with 32 GB of RAM memory and two Intel(R) Xeon(R) CPU E5-2620 v3 2.40 GHz processors (12 cores-24 hardware threads). The operating system used was Ubuntu Server 64 bits with the kernel 4.4.0-59-generic. The GCC version used was the 5.4.0 using the compiler -O3 flag.
– **M2** is 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.

5.1 Applications

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 [19], while for *Pbzip2* more details can be found in [20].

Lane Detection is a video processing application to detect road lanes, implemented by using the OpenCV library. To introduce parallelism in the sequential code, it is annotated 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) the 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[3]. This is a very coarse-grained application characterized by a stream parallel programming model. The SPar version is annotated 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.

[3] `http://compression.ca/pbzip2/`

5.2 Comparison with Handwritten Implementations

Before evaluating the ability to satisfy SLO specified by the user, we want to show 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 [19].

Performance To measure the maximum performance presented in Table 2, we executed both the reference and our solution generated versions by running them with 24 threads (to have at most one thread per core). The reported results refer to machine M2. For our generated version, we did not specify any SLO, but we still monitor the application by using Nornir. By doing so, we monitored both the overhead introduced by the interaction with the self-adaptive support and possible inefficiencies in the generated code. As shown by the results in Table 2 relative to the energy-aware SLOs, 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 and its optimizations as runtime support in Nornir, while the reference implementation was based on Pthreads.

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

Table 2 Performance improvement with respect to a handwritten implementation using FastFlow for the energy-aware SLOs. Negative percentages are the overheads (means slower) added by the SPar and SLO abstractions. LOC Reduction, negative values mean that SPar with SLO attributes is more concise than the handwritten one.

Code Intrusion To measure the code intrusion, we rely on Lines of Code (LOC) metric in Table 2 for energy-aware SLOs. Despite that LOC is not universally accepted, it is commonly used to compare different implementations of the same application [40]. 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 with SLOs requires a lower code intrusion with respect to FastFlow [17,20]. Since SLOs can be defined by only inserting the objective through attributes, this practice reduces significantly the lines of code. The handwritten version increased significantly the lines of code because implementing a strategy requires implementing all the details relative to resource management and monitoring.

5.3 SLO Analysis

In this section, we analyze the use of the SLO attributes laying emphasis in the generated self-adaptive strategies. The main goal is to provide a discussion regarding the adaptivity and effectiveness of the SLO strategies with a set of stream processing applications.

The `slo::Throughput` SLO can be used in any application parallelized with SPar that has at least one replicated stage. In Figure 5 is shown the result of Pbzip with a target throughput of 40 tasks per second, representing an SLO defined by the user. Figure 5 shows the measured throughput compared to SLO as well as the number of replicas used on each monitoring step. It is possible to observe that the throughput oscillated significantly during the execution, which is caused by the application and its input load characteristic. Because of the throughput oscillations, the self-adaptive strategy needed to change the number of replicas several times responding to throughput fluctuations and pursuing performance optimization. In some events, it is possible to note SLO violations caused by the execution variation. The strategy responded to such variations, but sometimes it was not fast enough since such short-duration load spikes occur randomly and their prediction is not possible.

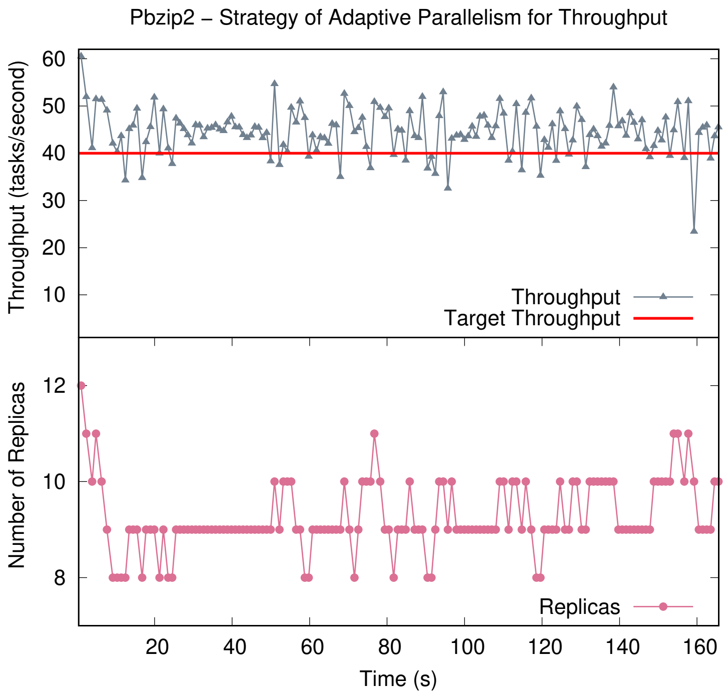


Fig. 5 M1-Characterization of *Pbzip2* application with `slo::Throughput(40)`.

Regarding the `slo::Latency` attribute, we tested this SLO under different configurations to evaluate if the strategy impacts on the application performance. For instance, we tested in a video streaming application using a file as an input to simulate a typical execution. A representative outcome of this experiment is shown in Figure 6 with a latency constraint of 1000 milliseconds, which simulates the definition of a representative SLO by the user.

In Figure 6, the strategy is characterized by the measured latency. We also presented the number of replicas used in different instants of the execution. Considering the results from Figure 6, we can identify that the latency varied during the execution because some frames require more time to be processed. Thus, causing unpredictable variations. Despite the adaptive strategy changed the number of replicas when necessary responding to the latency oscillations, some SLO violations occurred due to such short-duration fluctuations. Under a more stable workload trend, the adaptive strategy is expected to find a suitable number of replicas and maintain this number throughout the entire execution.

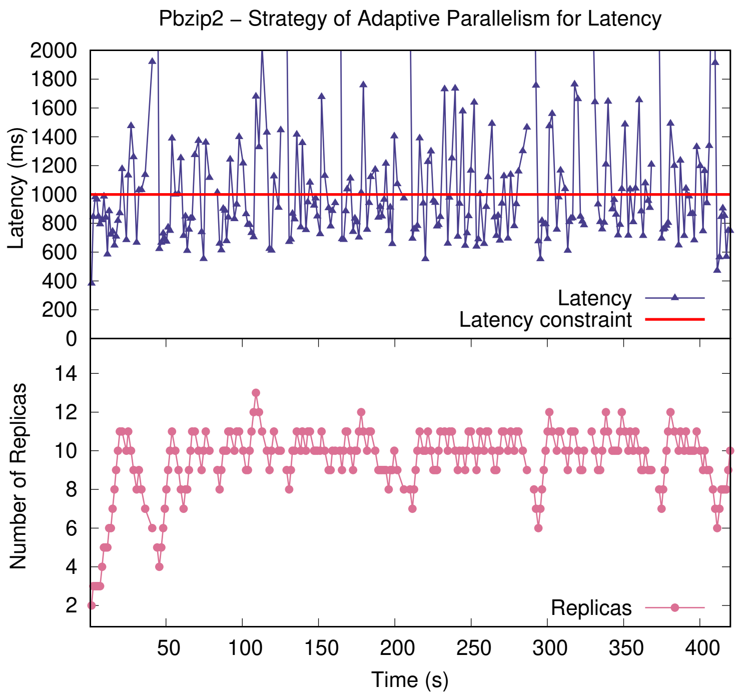


Fig. 6 M1-Lane Detection application with `slo::latency(1000 ms)`.

Concerning the use of the `slo::CPU` SLO, Figure 7 shows the execution of the Pbzip2 application with the attribute defining the maximum utilization to 60%, which is an empirically defined scenario, simulating an execution that could have a CPU load slightly higher than half of the machine's resources.

Such a scenario is representative of applications running on shared environments. We tested this SLO strategy with two representative threshold values, using the environment variable (`SLO_THREASHOLD`): 10 and 20%. These were the most suitable thresholds for stream parallelism, as seen in [39]. We also ran one variant using the blocking mode (`-spar_blocking` compilation option in SPAR) that tends to consume fewer CPU resources by only distributing new tasks upon requests from the active threads. The results are compared to the CPUlimit utility tool, which also was set to limit the CPU usage in 60%. For the tests using CPUlimit, we set a number of application threads equal to the number of hardware threads, which is what is done by default in several runtimes. The self-adaptive strategy, on the other hand, uses a custom number of active threads by changing the status of the replicas at run-time according to the heuristic policy implemented (Section 4.1.3).

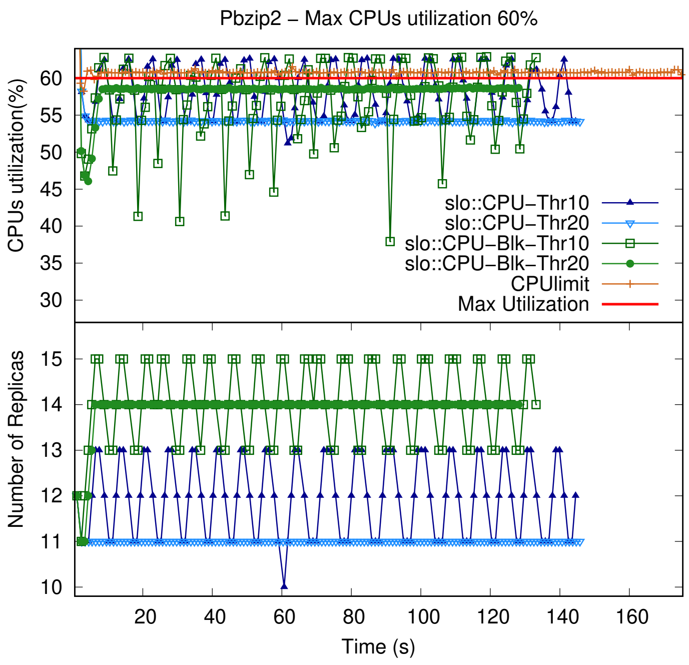


Fig. 7 M1-Characterization of *Pbzip2* application with `slo::CPU(60)`.

In the results from Figure 7, we can observe that CPUlimit was unable to enforce the required SLO. It is relevant to highlight that all executions presented a high CPU utilization in the first second. This event is caused by the application startup routines, such as threads and queues creation. The threshold of 10% introduced instability by triggering too frequent changes in the number of replicas, which also induced variation in CPU utilization. On the other hand, the threshold of 20% was the most accurate and stable one. By using the `-spar_blocking` compilation flag, it reduced the CPU utiliza-

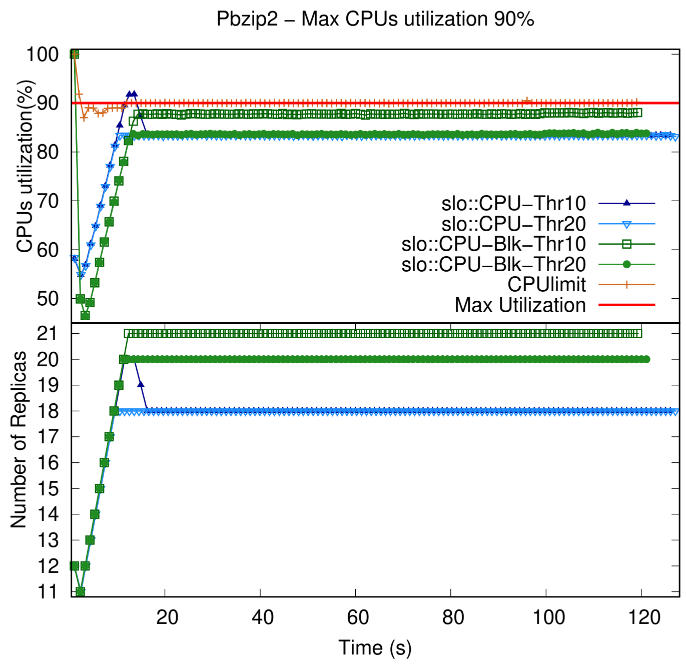


Fig. 8 M1-Characterization of *Pbzip2* application with `slo::CPU(90)`.

tion. Consequently, this resulted in an opportunity to use more replicas in the parallel region.

Figure 8 introduces the results of CPU utilization with a higher SLO value of at most 90% CPU utilization. Such a scenario was tested in order to evaluate the executions when almost all the machine resources available could be used. It tends to impact in the number of replicas used by the adaptive strategy. Comparing the different versions, we can visualize that the threshold 20% again caused fewer SLO violations by reaching a stable number of replicas after the first calibration phase. The CPUlimit presented oscillations in the utilization while the `-spar_blocking` compilation flag again enabled the use of additional replicas and avoided SLO violations.

We now show the results obtained by running with the `slo::CPU` SLO with all the considered applications. The results presented are an average of 10 executions. In Figure 9, is shown the throughput of the execution considering the three representative applications, and two representative `slo::CPU` SLO configurations: 60 and 90%. It is important to note that the SLO strategies are compared to a static degree of parallelism version using the CPUlimit for SPar and Intel TBB.

Considering the SLO of 60%, it is possible to identify a similar outcome regarding the different applications. In the self-adaptive executions, when using the `spar_blocking` compilation flag, it achieved a higher throughput rates than the default non-blocking execution. The self-adaptive strategy dynamically tunes the number of replicas resulting in the highest throughput rates. This result indicates that the way in which CPUlimit works (*i.e.*, continuously pausing and resuming the target process) causes performance overhead. CPUlimit in SPar had a lower throughput, while TBB and SPar Blocking achieved better performance.

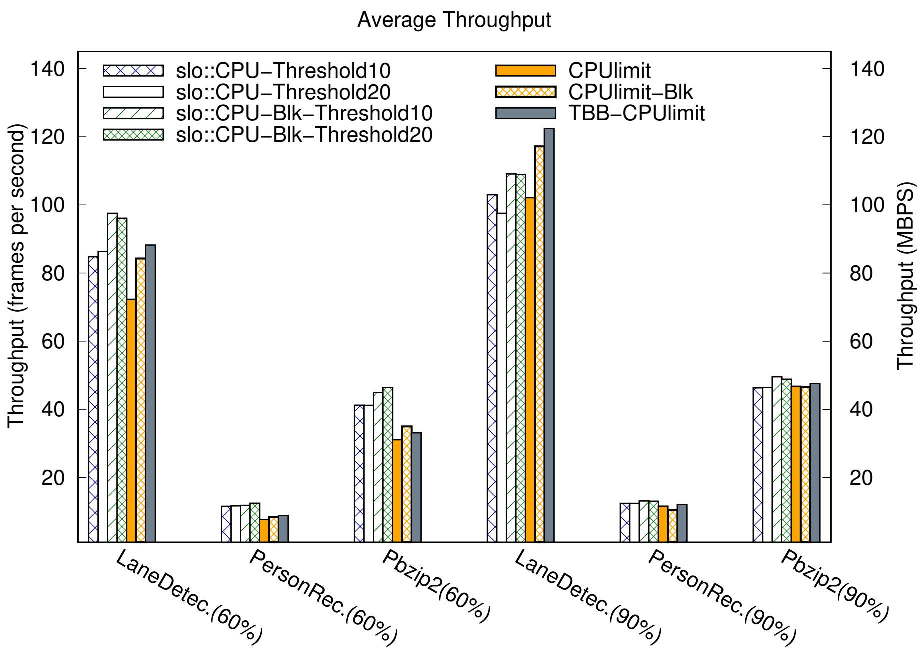


Fig. 9 M1-Throughput of applications. Left side in frames per second refers to video applications. Right side in MBPS is related to the throughput of Pbzip2.

The result of running with `slo::CPU` in 90% showed similar results with respect to 60%. Although the contrasts between our generated self-adaptive strategy and CPUlimit were smaller, our strategy again was significantly better in most cases. In Lane Detection with 90% CPU utilization SLO, both TBB and SPar blocking achieved the highest throughput. CPUlimit blocked significantly less the threads with the low CPU restriction of 90% CPU utilization SLO, which increased the application performance. In Lane Detection, the TBB version outperformed SPar because TBB improves the load balancing, while in Person Recognition and Pbzip2 both versions achieved similar performance. Considering the different applications and their execution characteristics, it is possible to note that CPUlimit performed better in those applications with a more balanced load, while performed worst in the irregular processing applications (Person Recognition). This indicates that CPUlimit is not a suitable alternative for limiting CPU utilization in stream processing applications, which are usually unbalanced because of their intrinsic dynamic nature.

In order to further characterize CPUlimit, we also evaluated the impact of the number of replicas. Figure 10 presents the results on Pbzip with a representative `slo::CPU` SLO of 60%. In this test, the results from our self-adaptive strategy are compared to a static number of replicas in SPar and TBB managed by CPUlimit. The throughput of our strategies is presented in all number of replicas because any of those numbers could be used during the execution, depending on the decisions made by the regulator algorithm. It is possible to note that the configuration using 12 replicas was the best CPUlimit configuration in SPar and TBB, although the self-adaptive strategy in blocking mode still achieved the highest throughput. Regarding CPUlimit, the blocking mode only achieved a better performance in specific cases comparing to the default non-blocking mode. Comparing the results where TBB outperformed SPar running with one application thread per hardware thread in Figure 9, the several number of replicas in TBB only won with 14 and 16 replicas. On the other hand, SPar with the blocking mode outperformed TBB in most cases.

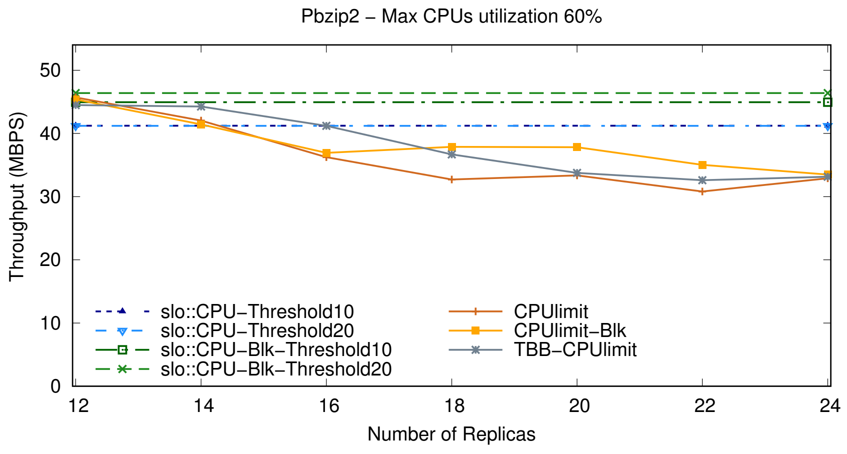


Fig. 10 M1-CPUlimit characterization with different number of replicas.

The outcome from Figure 10 highlights the correlation between the number of replicas and the application throughput, showing that using a tool like CPUlimit for limiting the CPU utilization SLO is inefficient in the stream processing context. The results indicate that even if CPUlimit is used, a suitable number of replicas has still to be found. However, finding a suitable number of replicas tends to be a complex task in stream processing applications. Additionally, the number of replicas often has to be adapted during execution according to performance or efficiency goals, because this class of applications runs without a defined end of the computation. Therefore, rerun the application multiple times until a suitable number of replicas is found, it becomes unfeasible for stream processing applications. Consequently, our strategy that dynamically adapts the number of replicas in SPar at run-time is a feasible and effective approach, which showed promising performance outcomes.

In Figure 11, we analyze a different scenario, where the user requires a throughput as well as an energy constraint. This scenario exploits the usage of energy strategies. The defined SLO throughput (`slo::Throughput`) was 20 tasks per second and power consumption (`slo::Power`) lower than 65 watts for the *Pbzip* application. In this test, we add some external noise to show that our strategy for controlling performance and energy succeeds in providing the required SLO even in the presence of unexpected behaviors.

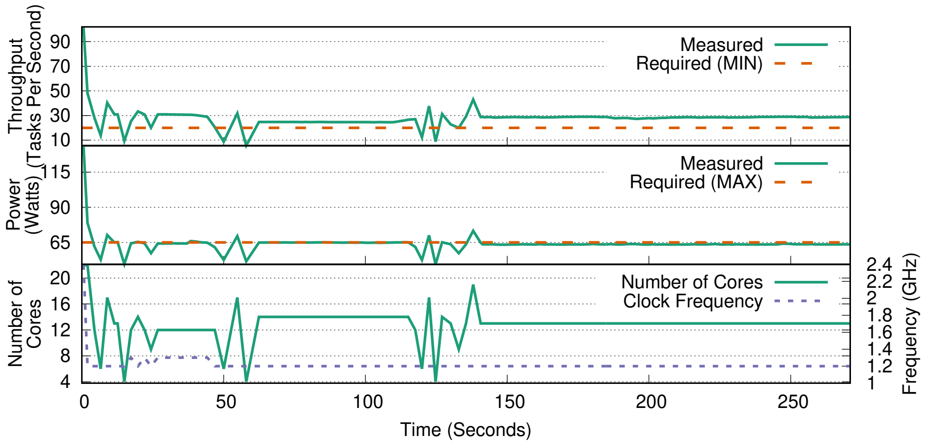


Fig. 11 M2-*Pbzip2* application with `slo::Throughput(20)` and `slo::Power(65)`.

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 generated code and the compatible 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 11, our generated strategy with its 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 generated strategy 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 generated strategy uses its runtime to satisfy 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.

In Figure 12, we analyze the *Lane Detection* application, in a scenario where it produces no more than 50 frames per seconds. In such a 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 (`slo::Utilization`) of 80%. In the upper part of Figure 12, 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 generated strategy decreases the number of resources allocated to the application, decreasing the power consumption from 90 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%).

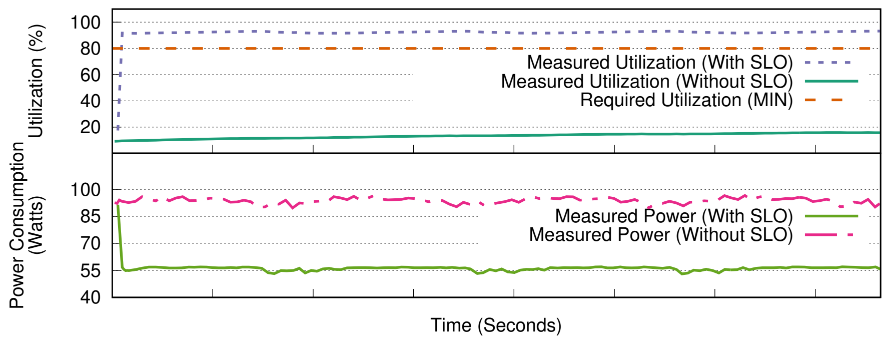


Fig. 12 M2-*Lane Detection* application with `slo::Utilization(80)`.

The target of the experiment in Table 3 is to demonstrate 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 show that a parallel application with the same performance of the sequential one has lower power consumption. We were able to limit the SLO throughput by using the `slo::Throughput` combined with `slo::Power`.

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 quadratic with the clock frequency [8], running an application on more cores at a lower frequency is usually more energy efficient than

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

Table 3 Power consumption reduction obtained by a parallel application with the same throughput of the sequential one.

running it on fewer cores at a higher frequency. Having tools and methodologies for doing that automatically and with low code intrusion, like those we proposed through SLO attributes in this work is of paramount importance for enabling such techniques in real-world scenarios.

6 Conclusion

In this work, we presented a new and simpler way to express SLOs in sequential source codes. Our approach was designed to target stream processing applications along with its parallelization support. We validated this approach by implementing it in the SPar language and compiler, which now recognizes the C++11 SLO attributes and automatically performs source-to-source transformations to the self-adaptive strategies implemented in the FastFlow and Nornir libraries. The main advantage is that application programmers can now simply define SLOs by inserting the attributes in the source code and the compiler generates the appropriate self-adaptive strategy to meet the target SLO. This new approach does not require from programmers implementation expertise either system resource management.

Moreover, our implemented solution has proven to be efficient and offers many opportunities to improve the QoS in stream processing applications. We were able to reduce power consumption and increase performance in certain cases. Regarding the CPU utilization SLO, the performance was improved in most cases compared with CPULimit. While our strategy relies on changing the number of replicas, CPULimit works at the operating system level limiting the CPU utilization by continuously pausing and resuming the target process. Although the goals and efforts in this work were more in the abstraction of SLOs implementation, we visualize a set of future works. For instance, a deep performance validation can be conducted to cover different workloads and stream processing scenarios. We also plan to implement other self-adaptive strategies and SLOs. We would like to refine our `slo::CPU` and `slo::Latency` SLOs to combine them with power consumption SLOs. Eventually, our approach could be extended to other computing environments such as cloud or cluster architectures.

Acknowledgements This study was partially funded by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, and by the FAPERGS 01/2017-ARD project ParaElastic (No. 17/2551-0000871-5). We would like to thank Laboratrio de Alto Desempenho (LAD) from PUCRS for partially providing computing resources.

References

1. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and Many-core Computing Systems*, volume 1 of *PDC*, page 14. Wiley, 2014.
2. M. Aldinucci, M. Meneghin, and M. Torquati. Efficient Smith-Waterman on Multi-core with FastFlow. In *Proceedings of the Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 195–199, 2010.
3. F. Alessi, P. Thoman, G. Georgakoudis, T. Fahringer, and D. S. Nikolopoulos. Application-Level Energy Awareness for OpenMP. In *International Workshop on OpenMP*, pages 219–232. Springer, 2015.
4. H. C. M. Andrade, B. Gedik, and D. S. Turaga. *Fundamentals of Stream Processing*. Cambridge University Press, New York, USA, 2014.
5. J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe. Siblingrivalry. In *Proc. of the 2012 Intl. Conf. on Compilers, architectures and synthesis for embedded systems - CASES '12*, page 91, New York, New York, USA, Oct. 2012. ACM Press.
6. B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering*. O'Reilly, Boston, USA, 2016.
7. R. Buyya, C. Vecchiola, and T. Selvi. *Mastering Cloud Computing*. McGraw Hill, 2013.
8. A. P. Chandrakasan and R. W. Brodersen. Minimizing Power Consumption in Digital CMOS Circuits. *Proceedings of the IEEE*, 83(4):498–523, 1995.
9. CPUlimit. CPU Usage Limiter for Linux roadmap <http://cpulimit.sourceforge.net/>, 2018. Last access Dec, 2018.
10. M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1396–1410, Oct 2008.
11. M. Danelutto, J. D. Garcia, L. M. Sanchez, R. Sotomayor, and M. Torquati. Introducing Parallelism by Using REPARA C++11 Attributes. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 354–358. IEEE, 2016.
12. D. De Sensi, T. De Matteis, and M. Danelutto. Simplifying Self-Adaptive and Power-Aware Computing with Nornir. *Future Generation Computer Systems*, pages –, 2018.
13. D. De Sensi, M. Torquati, and M. Danelutto. A Reconfiguration Algorithm for Power-Aware Parallel Applications. *ACM Transactions on Architecture and Code Optimization*, 13(4):43:1–43:25, dec 2016.
14. C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGARCH Comput. Archit. News*, 42(1):127–144, Feb. 2014.
15. Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. A helper thread based edp reduction scheme for adapting application execution in cmps. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE Intl. Symposium on*, pages 1–14, April 2008.
16. A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-Regulating Stream Processing in Heron. *Proceedings of the VLDB Endowment*, 10:1825–1836, 2017.
17. D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005, March 2017.
18. D. Griebler, D. De Sensi, A. Vogel, M. Danelutto, and L. G. Fernandes. Service Level Objectives via C++11 Attributes. In *Euro-Par 2018: Parallel Processing Workshops*, page 12, Turin, Italy, August 2018. Springer.
19. D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes. Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing*, ParCo'17, pages 698–707, Bologna, Italy, September 2017. IOS Press.

20. D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes. High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming*, pages 1–19, February 2018.
21. J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
22. H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic Knobs for Responsive Power-aware Computing. *SIGPLAN Not.*, 46(3):199–212, 2011.
23. J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, Jan 2003.
24. J. Li and J. F. Martínez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. *Proc. of Intl. Symposium on High-Performance Computer Architecture*, pages 77–87, 2006.
25. M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Controlling Software Applications via Resource Allocation within the Heartbeats Framework. In *IEEE Conference on Decision and Control*, pages 3736–3741. IEEE, 2010.
26. J. Maurer and M. Wong. Towards Support for Attributes in C++ (Revision 6). Technical report, The C++ Standards Committee, 2008.
27. M. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, MA, USA, 2012.
28. P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker. Flicker: a dynamically adaptive architecture for power limited multicore systems. *ACM SIGARCH Computer Architecture News*, 41(3):13, July 2013.
29. K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Proc. of the 2011 IEEE Intl. Symposium on Workload Characterization*, IISWC '11, pages 116–125, Washington, DC, USA, 2011. IEEE Computer Society.
30. J. Reinders. *Intel Threading Building Blocks*. O'Reilly, USA, 2007.
31. R. A. Shafik, A. Das, S. Yang, G. Merrett, and B. M. Al-Hashimi. Adaptive Energy Minimization of OpenMP Parallel Applications on Many-Core Systems. In *Parallel Programming and Run-Time Management Techniques*, pages 19–24, 2015.
32. S. Sridharan, G. Gupta, and G. S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *Proc. of the 27th Intl. ACM Conf. on Intl. Conf. on supercomputing - ICS '13*, page 337, New York, New York, USA, June 2013. ACM Press.
33. R. Sturm, W. Morris, and M. Jander. *Foundations of Service Level Management*. SAMS, Boston, USA, 2000.
34. M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading. In *Proc. of the 13th Intl. Conf. on Architectural support for programming languages and operating systems - ASPLOS XIII*, volume 42, page 277, New York, New York, USA, Mar. 2008. ACM Press.
35. W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction*, pages 179–196, Grenoble, France, 2002. Springer.
36. E. Totoni, N. Jain, and L. V. Kalé. Power management of extreme-scale networks with on/off links in runtime systems. *TOPC*, 1(2):16, 2015.
37. E. Totoni, J. Torrellas, and L. V. Kale. Using an adaptive hpc runtime system to reconfigure the cache hierarchy. In *Proc. of SC 2014*, pages 1047–1058. IEEE Press, 2014.
38. V. Vassiliadis, K. Parasyris, C. Chalios, C. D. Antonopoulos, S. Lalis, N. Bellas, H. Vandierendonck, and D. S. Nikolopoulos. A Programming Model and Runtime System for Significance-aware Energy-efficient Computing. *SIGPLAN Not.*, 50(8):275–276, 2015.
39. A. Vogel, D. Griebler, D. D. Sensi, M. Danelutto, and L. G. Fernandes. Autonomic and Latency-Aware Degree of Parallelism Management in SPar. In *Euro-Par 2018: Parallel Processing Workshops*, page 12, Turin, Italy, August 2018. Springer.
40. E. J. Weyuker. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.