Keep Calm and React with Foresight: Strategies for Low-Latency and Energy-Efficient Elastic Data Stream Processing

Tiziano De Matteis and Gabriele Mencagli

Department of Computer Science, University of Pisa, Largo B. Pontecorvo 3, I-56127, Pisa, Italy

{dematteis,mencagli}@di.unipi.it

Abstract

This paper addresses the problem of designing scaling strategies for *elastic* data stream processing. Elasticity allows applications to rapidly change their configuration on-the-fly (e.g., the amount of used resources) in response to dynamic workload fluctuations. In this work we face this problem by adopting the *Model Predictive Control* technique, a control-theoretic method aimed at finding the optimal application configuration along a limited prediction horizon in the future by solving an online optimization problem. Our control strategies are designed to address latency constraints, using Queueing Theory models, and energy consumption by changing the number of used cores and the CPU frequency through the *Dynamic Voltage and Frequency Scaling* (DVFS) support available in the modern multicore CPUs. The proactive capabilities, in addition to the latency- and energy-awareness, represent the novel features of our approach. To validate our methodology, we develop a thorough set of experiments on a high-frequency trading application. The results demonstrate the high-degree of flexibility and configurability of our approach, and show the effectiveness of our elastic scaling strategies compared with existing state-of-the-art techniques used in similar scenarios.

Categories and Subject Descriptors [*Information systems*]: Data streams

Keywords Data Stream Processing, Elasticity, Multicore Programming, Model Predictive Control, DVFS.

1. Introduction

Data Stream Processing [9] (briefly, DaSP) is a computing paradigm enabling the real-time processing of continuous data streams that must be processed on-the-fly with stringent Quality of Service (QoS) requirements. These applications are usually fed by potentially irregular flows of data that must be timely processed to detect anomalies, provide real-time incremental responses to the users, and take immediate actions. Typical application domains of this paradigm are high-frequency trading, network intrusion detection, social media, and monitoring applications like in healthcare and intelligent transportation systems.

Due to their long-running nature (24h/7d), DaSP applications are affected by highly variable arrival rates and exhibit abrupt changes in their workload characteristics. *Elasticity* is a fundamental feature in this context; it allows applications to scale up-/down the used resources to accommodate dynamic requirements and workload [14, 15] by maintaining the desired QoS in a cost-effective manner, i.e. without statically configuring the system to sustain the peak load. However, elasticity is a challenging problem in DaSP applications that maintain an internal state while processing input data flows. The runtime system must be able to split and migrate the data structures supporting the state while preserving correctness and performance. This problem has been studied extensively in the last years, with works that propose efficient state migration protocols and reconfiguration techniques [12, 15, 17].

Besides protocols and low-level mechanisms, the design of clever *scaling strategies* has acquired an increasing strategic significance owing to the high number of requirements that must be synergically taken into account when the strategy selects a new resource allocation (e.g., latency, throughput, resource and energy consumption). Avoiding unnecessary or too frequent reconfigurations by keeping the actual QoS close to user's specifications is a key point to deliver high-performance cost-effective solutions. In this work we propose *latency-aware* and *energy-efficient* scaling strategies with *predictive* capabilities. The main contributions of our work are the following:

- most of the existing scaling strategies have a *reactive* nature [12, 15, 17]. In contrast our approach is based on a control-theoretic method known as *Model Predictive Control* [11] (briefly, MPC), in which the strategies take into account the behavior of the system along a limited future time horizon to choose the reconfigurations to execute. As far as we know, this is the first time that MPC-based strategies have been applied in the DaSP domain;
- most of the existing strategies are mainly *throughput oriented* [14, 15, 20], i.e. they adapt the number of resources to sustain the actual input stream pressure. In many real-world scenarios this is not sufficient as the application must provide timely and fresh responses to the users. We tackle this problem by focusing explicitly on application *latency* and we study how to model and control it;
- we integrate our strategies with capabilities to deal with *energy consumption* issues. To this end, we target multicores with Dynamic Voltage and Frequency Scaling (DVFS) support.

We experiment our strategies in a high-frequency trading application based on both synthetic and real dataset workload traces. To provide a concise summary of the results and to easily compare different strategies, we analyze them in terms of the so-called SASO properties [14, 18]: *stability*, *accuracy*, *settling time* and *overshoot*. As a result, the configuration parameters of our strategies will exhibit a clear impact on these properties by rendering the tuning

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03... $15.00
DOI: http://dx.doi.org/10.1145/2851141.2851148

phase easier. Furthermore, we compare our predictive strategies with some existing reactive ones proposed in the literature. At the present this work covers single multicore systems, but we plan to extend it to distributed environments in the future.

The outline of this paper is the following. Sect. 2 introduces basic concepts about DaSP. Sect. 3 presents our MPC-based approach. Sect. 4 describes the runtime support. Sect. 5 provides a comprehensive analysis of our strategies and a comparison with the state-of-the-art. Finally, Sect. 6 reviews existing works and Sect. 7 concludes this work by highlighting our future research directions.

2. Data Stream Processing

In the last years several Stream Processing Engines (SPEs) have been proposed for developing and executing DaSP applications. Examples are IBM InfoSphere [2], Apache Storm [4] and Spark Streaming [3]. In these frameworks the programmer expresses applications as data-flow graphs of *operators* [9] interconnected by *data streams*, i.e. unbounded sequences of data items with a fixed structure like a record of attributes (*tuples*).

Operators belong to two different classes [10]: *stateless operators*, characterized by each input data item that produces an output independently of any previous input tuple, and *stateful operators* that require to keep an internal state. Notably, *partitioned-stateful operators* [9] ingest an input sequence of tuples belonging to different groups identified by a partitioning attribute (*partitioning key*). For these operators the internal state can be decomposed in partitions, each one assigned to a distinct group. Examples are user-defined operators that process network traces partitioned by IP address, or market feeds partitioned by stock symbol.

Due to the unlimited length of the stream, the semantics of stateful operators has been redefined with respect to their original counterparts in classic databases. One of the main solutions proposed in DaSP consists in using succinct data structures like synopses and sketches to maintain a concise summary of the stream characteristics (e.g., the statistical properties of the input elements). Alternatively, the operator processing logic can be applied on the most recent tuples only, which are maintained in a *window* buffer. According to the window boundaries, windows can be *time-based* or *count-based* and they usually have a sliding semantics.

As recognized by the literature, sliding windows are the predominant abstraction in DaSP [9]. Therefore, in the ensuing discussion we will focus on partitioned-stateful operators based on a sliding window model that represent the target of the most recent research [14].

2.1 Parallelism and Elasticity

To respect user QoS requirements, hotspot operators in data-flow graphs must be identified and internally parallelized using multiple functionally equivalent *replicas* that handle a subset of the input tuples [9]. While for stateless operators this parallelization is trivial, for stateful ones it is complex because the computation semantics depends on the order in which input items are processed and the data structures supporting the internal state updated. This implies that simple parallel implementations that randomly distribute input items to replicas that use independent *locks* to protect the state partitions cannot be used because they might alter the computation semantics. In fact, though locks allow the replicas to access and modify the state partitions atomically, the order in which input items within the same group are processed might be different with respect to their order of appearance in the input stream. As an example, this may happen if the processing time per tuple is highly variable. In that case, the output sequence of the parallelization can be different than the one of the original operator, thus violating the *sequential equivalence*.

The typical solution used in the literature [14, 15, 17, 28] is to make each state partition owned by a single replica. The distribution must guarantee that *all* the tuples with the same partitioning key are routed to the same replica. In this way the state partitions do not require to be protected by locks, and tuples within the same group are processed in the same order of appearance in the input stream*. The idea of this parallelization is sketched in Fig. 1.

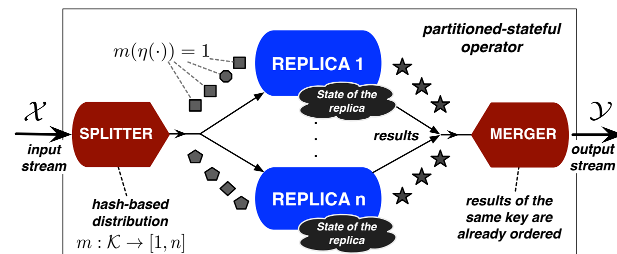


Figure 1: Functionalities of a parallel partitioned-stateful operator: the tuples with the same key attribute are processed by the same replica.

The operator receives an input stream of tuples $\mathcal{X}$ and produces an output stream of results $\mathcal{Y}$. For each tuple $x_i \in \mathcal{X}$ we denote by $\eta(x_i) \in \mathcal{K}$ its partitioning key, where $\mathcal{K}$ is the key domain. For each key $k \in \mathcal{K}$, $p_k \in [0, 1]$ denotes its relative frequency.

The replicas are interfaced with the input and the output streams through two functionalities called *splitter* and *merger*. The first is responsible for distributing each input tuple to the corresponding replica by using a hash function called *distribution function* $m : \mathcal{K} \rightarrow [1, n]$, where n is the actual number of replicas. The merger is in charge of collecting the results from the replicas.

Streaming applications are characterized by highly variable execution scenarios. The dynamicity depends on three different factors: *(D1)* the variability of the arrival rate, *(D2)* the variability of the key frequency distribution, and *(D3)* the variability of the processing time per tuple. The applications must tolerate all these aspects by keeping the QoS optimized according to the user criteria. In this paper we are interested in the *latency* (response time), i.e. the time elapsed from the reception of a tuple that triggers the operator internal processing logic and the production of the corresponding output. Latency requirements are usually expressed in terms of *threshold specifications*, e.g., by keeping the average latency under a user-defined threshold. Time periods in which these requirements are not met are considered *QoS violations*.

A simple solution to achieve the desired QoS is to configure the system to sustain the peak-load. This solution is usually too expensive in terms of resource consumption, and it can even be ineffective if a static distribution function is not unable to balance the workload during the entire execution duration, e.g., if the key frequencies exhibit wide variations at runtime. Instead, elastic operators can be intrinsically able to deal with all the *D1, D2, D3* factors [14, 23]. The scaling strategies that will be developed in this work will be able to change the *parallelism degree* n (i.e. the actual number of replicas), the *distribution function* m, and the *operating frequency* f used by the CPUs.

The scaling strategy monitors the actual behavior of an operator, detects if it is not currently able to meet the desired QoS, chooses a better configuration, and applies it with minimal impact on the computation performance. We will compare strategies according to the well-known SASO properties [14, 18]:

- *(P1) Stability*: the strategy should not produce too frequent modifications of the actual configuration;

*This partial ordering of results is usually accepted in many DaSP applications.

- *(P2) Accuracy*: the system should satisfy the QoS objectives by minimizing the number of QoS violations;
- *(P3) Settling time*: the strategy should be able to find a stable configuration quickly;
- *(P4) Overshoot*: the strategy should avoid overestimating the configuration needed to meet the desired QoS.

It is not often possible to optimize all these properties together, but we are interested in finding a strategy configuration that achieves the best trade-off.

2.2 Assumptions

Our methodology will be presented and experimented by assuming the following conditions:

1. *(A1) homogeneous replicas*: all the replicas of the same operator will be executed on homogeneous cores of the same shared-memory architecture based on multicore CPUs;
2. *(A2) single node*: in this paper will target homogeneous multi-core CPUs. We will study the impact of our approach on shared-nothing architectures like clusters in our future work.

3. Elastic Scaling based on MPC

MPC is a design principle for controllers widely adopted in the process industries [11]. The basic idea is that the controller's actions are taken by solving an optimization problem that explicitly uses a model to predict the future system behavior over a limited *prediction horizon*. The logic of a MPC controller is composed of three interconnected components sketched in Fig. 2.

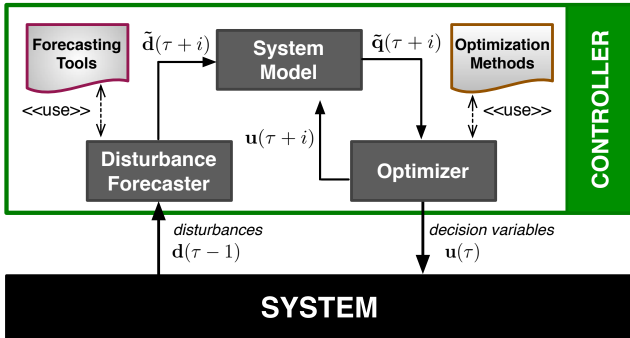


Figure 2: Internal logical structure of a MPC controller: disturbance forecaster, system model and optimizer components.

Disturbance forecaster. The controller is able to observe the last measurements $d(\tau-1)$ of the exogenous uncontrollable events affecting the system, the so-called measured *disturbances*, e.g., the arrival rate and the processing time per input tuple. These monitored data are collected periodically once per *control step* τ (a fixed time interval). Their future value $\widetilde{d}(\tau)$ is estimated with statistical forecasting tools based on a limited history of the past samples [19].

System model. The MPC approach is *model driven*: a system model is used to compare and evaluate alternative configurations. The model captures the relationship between *QoS variables* (e.g., latency, energy) and the current system configuration expressed by *decision variables* (e.g., parallelism degree, CPU frequency). The general structure of the model is as follows:

$$\widetilde{\mathbf{q}}(\tau) = \Phi\big(\mathbf{u}(\tau), \widetilde{\mathbf{d}}(\tau)\big) \tag{1}$$

where $\widetilde{\mathbf{q}}(\tau)$ and $\widetilde{\mathbf{d}}(\tau)$ are the predicted vectors of QoS variables and disturbances for step τ and $\mathbf{u}(\tau)$ is the decision variable vector.

Optimizer. The optimization problem solved by the MPC controller is defined as follows:

$$\min_{U^h(\tau)} \mathcal{J} = \sum_{i=0}^{h-1} L\big(\widetilde{\mathbf{q}}(\tau+i), \mathbf{u}(\tau+i)\big) \tag{2}$$

The result is an optimal *reconfiguration trajectory* $U^h(\tau) = (\mathbf{u}(\tau), \mathbf{u}(\tau+1), \ldots, \mathbf{u}(\tau+h-1))$ over a prediction horizon of $h \geq 1$ steps. The basic principle of MPC is that only the first element $\mathbf{u}(\tau)$ of the reconfiguration trajectory is used to steer the system to the next control step. Then, the strategy is re-evaluated at the beginning of the next control interval using the new disturbance measurements to update the forecasts. In this way the prediction horizon shifts forward by one step, from this the name *receding horizon*. The approach has led to very good results in a large number of applications over the last years [11], due to its intrinsic capability to incorporate a feedback mechanism.

3.1 Predictive Strategies for Elastic Scaling

To apply the MPC methodology, our intra-operator parallelization is enhanced with an additional functionality, i.e. the *controller* (see Fig. 2).

3.1.1 Measured disturbances

The controller observes the operator execution and acquires periodically measurements from the computation. Tab. 1 shows the basic measurements gathered by the controller. These metrics are relative to a control step (omitted to simplify the notation), i.e. at the beginning of a generic control step τ the updated measurements related to the previous step $\tau-1$ are available.

Symbol	Description
$T_{\mathcal{A}}, \sigma_{\mathcal{A}}$	Mean and standard deviation of the inter-arrival time per triggering tuple (with any key). The arrival rate is $\lambda = T_{\mathcal{A}}^{-1}$. These measurements are collected by the splitter.
$\{p_k\}_{k\in\mathcal{K}}$	Frequency distribution of the keys. Measured by the splitter.
$\{\mathbf{c}_k\}_{k\in\mathcal{K}}$	Arrays of sampled computation times per triggering tuple for each key, measured in *clock cycles*. Each $\mathbf{c}_k$ is an array of samples for key k, collected by the replica that owned that key during the last step.

Table 1: Basic monitored *disturbance* metrics gathered at the beginning of each control step by the controller.

We use the term *triggering tuple* to denote a tuple that triggers the internal processing logic of the operator. For window-based stateful operators a triggering tuple is any tuple that triggers a new window activation (according to the window triggering policy [9]). Non-triggering tuples are simply inserted into the corresponding window and it is reasonable to assume that they have a negligible computation cost. Computation times are collected in clock cycles to be independent from the used CPU frequency.

3.1.2 Derived metrics

The controller can derive various additional metrics from the basic ones. From the arrays of measurements $\mathbf{c}_k$ for each $k \in \mathcal{K}$, the controller measures the average number of clock cycles required to process a triggering tuple with key k. We denote it by $\mathcal{C}_k$, i.e. the mean value of the samples in the last array $\mathbf{c}_k$ gathered from the replicas. The mean computation time per triggering tuple of key $k \in \mathcal{K}$ is:

$$\widetilde{T}_k(\tau) = \frac{\widetilde{\mathcal{C}}_k(\tau)}{f(\tau)} \tag{3}$$

Since the computation times are collected in clock cycles, they need to be transformed in time by dividing for the used CPU frequency $f(\tau)$. This model is a good approximation for CPU-bound computations [26]. For memory-bound computations it may be less accurate. In the sequel we assume this model to be sufficiently accurate and we will investigate proper extensions in the future.

We define the (ideal) *service rate* as the average number of triggering tuples that the operator is able to serve per time unit provided that there are always new tuples to ingest. We are interested in the inverse of this quantity, i.e. the operator (ideal) *service time* $T_{\mathcal{S}}$. We denote by $w_k(\tau)$ the "weight" of the key $k \in \mathcal{K}$, defined as the product between the key frequency and the mean computation time per triggering tuple of that key, i.e. $w_k(\tau) = \widetilde{p}_k(\tau) \times \widetilde{T}_k(\tau)$. The mean computation time per triggering tuple of any key can be calculated as follows:

$$\widetilde{T}(\tau) = \sum_{k \in \mathcal{K}} w_k(\tau) \tag{4}$$

Under the assumption that the load is evenly distributed among the $n(\tau)$ replicas, the service time of the operator is roughly equal to:

$$\widetilde{T}_{\mathcal{S}}(\tau) \approx \frac{\sum_{k \in \mathcal{K}} w_k(\tau)}{n(\tau)} = \frac{\widetilde{T}(\tau)}{n(\tau)} \tag{5}$$

This model requires that, given the actual keys frequency distribution, there exists a distribution function m^{τ} that allows the workload to be (quasi) evenly balanced among the replicas. As stated in the recent literature [14], this assumption practically holds in many real-world applications, where skewed distributions are common but a well balanced distribution function can usually be found[†]. This aspect will be analyzed in detail in Sect. 5.

3.1.3 Performance and energy models

The decision variables of the models are: the number of replicas $n(\tau)$ and the CPU frequency (GHz) $f(\tau)$ used by the operator during step τ. They are represented by vector $\mathbf{u}(\tau) = [n(\tau), f(\tau)]^T$. Since our goal is to deal with latency-sensitive applications, our choice is to directly manage and configure the CPU frequency. Leaving the DVFS control to the CPU frequency governor of a commodity OS could not provide sufficient guarantees from a performance viewpoint, and will not be considered.

The outputs of the models are the predicted values of the QoS variables with a given configuration of the operator. For the step τ we are are interested in the response time (latency) $\mathcal{R}_{\mathcal{Q}}(\tau)$ and the power used $\mathcal{P}(\tau)$.

Latency model. Analogously to [23], we use a Queueing Theory approach. The mean response time of the operator during a control step τ can be modeled as the sum of two quantities:

$$\mathcal{R}_{\mathcal{Q}}(\tau) = W_{\mathcal{Q}}(\tau) + T(\tau) \tag{6}$$

where $W_{\mathcal{Q}}$ is the *mean waiting time* that a triggered tuple spent from its arrival to the system to when the operator starts the execution on the corresponding triggered window.

To find the mean waiting time per triggering tuple, our idea is to model the operator as a $G/G/1$ queueing system, i.e. a single-server system with inter-arrival times and service times having general statistical distributions. An approximation of the mean waiting time for this system is given by Kingman's formula [21]:

$$\widetilde{W}_{\mathcal{Q}}^{K}(\tau) \approx \left(\frac{\widetilde{\rho}(\tau)}{1 - \widetilde{\rho}(\tau)}\right)\left(\frac{\widetilde{c}_a^2(\tau) + \widetilde{c}_s^2(\tau)}{2}\right)\widetilde{T}_{\mathcal{S}}(\tau) \tag{7}$$

where the input parameters are the following:

- the utilization factor of the operator during step τ, defined as the ratio between its service time and its inter-arrival time, i.e. $\widetilde{\rho}(\tau) = \widetilde{T}_{\mathcal{S}}(\tau)/\widetilde{T}_{\mathcal{A}}(\tau)$;
- the coefficient of variation of the inter-arrival time $c_a = \sigma_{\mathcal{A}}/T_{\mathcal{A}}$ and of the service time $c_s = \sigma_{\mathcal{S}}/T_{\mathcal{S}}$.

As most of the Queueing Theory results, Expr. 7 can be used for stable queues only, i.e. such that $\widetilde{\rho}(\tau) < 1$. This implies that the model can be used for evaluating the response time of configurations in which the operator is not a bottleneck, i.e. *we are implicitly optimizing throughput in all of our strategies*. Furthermore, a valuable property of Kingman's formula is that it is independent from the specific distributions of the inter-arrival time and service time. This is an important property, as in real systems these distributions are in general unknown and must be estimated from the actual observations in real-time. In our work we will make the following assumptions:

- by using Expr. 7, we are assuming that the mean waiting time of a parallel server with $n(\tau)$ replicas and overall service time $\widetilde{T}_{\mathcal{S}}(\tau)$ is roughly the same of the one of a sequential server with the same service time;
- the mean inter-arrival time for the next step $\widetilde{T}_{\mathcal{A}}(\tau)$ is forecasted using statistical filters while the coefficient of variation is kept equal to the last measured one, i.e. $\widetilde{c}_a(\tau) = c_a(\tau - 1)$;
- the coefficient of variation of the service time is estimated by $\widetilde{c}_s(\tau) = c_s(\tau - 1)$. So doing, we suppose that c_s is unaffected by changes in the parallelism degree. However, we can note that c_s depends on the way in which the keys are partitioned among the replicas. For the moment being we neglect this aspect that will be investigated in our future work.

It is well known that the Kingman model provides good accuracy especially for systems close to saturation [21], which is a good property since our goal is to avoid wasting resources and energy. In our strategies this model will be used *quantitatively*. To increase its precision we use the following feedback mechanism to fit Kingman's approximation to the last measurements:

$$\widetilde{W}_{\mathcal{Q}}(\tau) = e(\tau) \cdot \widetilde{W}_{\mathcal{Q}}^{K}(\tau) = \frac{W_{\mathcal{Q}}(\tau - 1)}{\widetilde{W}_{\mathcal{Q}}^{K}(\tau - 1)} \cdot \widetilde{W}_{\mathcal{Q}}^{K}(\tau) \tag{8}$$

The parameter e is the ratio between the mean waiting time during the past step $\tau - 1$ collected by the splitter functionality and the last prediction obtained by Kingman's formula. The idea is to adjust the next prediction according to the past error. A similar mechanism has been applied with good results to the problem of estimating the response time of a chain of operators in [23].

Power consumption model. The energy consumed is expressed as the product between the power used and the execution time. Owing to the infinite nature of DaSP computations, the minimization of the instant power (*power capping*) is the main solution to reduce energy consumption and cutting down operating costs.

We need to estimate the actual utilized power on multicore CPUs with DVFS support. We are not interested in knowing the exact amount of power, but only a proportional estimation such that we can compare different operator configurations. In particular, we do not consider the static power dissipation of the CPU and the one of the other remaining system components (e.g., RAM), but we will focus on the dynamic power dissipation originated from logic-gate activities in the CPU, which follows the underlying formula [5, 26]:

$$\widetilde{\mathcal{P}}(\tau) \sim C_{eff} \cdot n(\tau) \cdot f(\tau) \cdot \mathcal{V}^2 \tag{9}$$

The power required during step τ is proportional to the used number of cores, the CPU frequency and the square of the supply voltage $\mathcal{V}$, which in turn depends on the frequency of the processor.

[†]We highlight that we are not assuming that the workload is always perfectly balanced, but that we are able to properly balance it among n replicas by changing the distribution function when necessary.

C_{eff} represents the *effective capacitance*, a technological constant that depends on the hardware characteristics of the CPU.

It is worth noting that by monitoring the system utilization factor $\rho(\tau)$ only, we are not able to choose the most power-efficient configuration that meets the target response time requirements. In fact, from Expr. 7 we can derive the utilization factor necessary to guarantee a certain $\mathcal{W}_Q$. Several configurations can achieve the same or a similar utilization factor, e.g., by using 5 replicas with a frequency of 2GHz or 10 replicas at 1GHz. Therefore, our strategies will need to be further able to estimate the power consumed by different operator configurations expressed by all the feasible combinations of the number of used cores and the employed CPU frequency.

3.2 Optimization Problem

The MPC-based strategies solve at each step the optimization problem defined in Expr. 2. The cost function is the sum of the step-wise cost L over a horizon of $h \geq 1$ future steps. A general form of the step-wise cost can be expressed as follows, with $i = 0, \ldots, h-1$:

$$\begin{aligned} L(\widetilde{\mathbf{q}}, \mathbf{u}, i) = Q_{cost}\big(\widetilde{\mathbf{q}}(\tau+i)\big) + & \qquad \textit{QoS cost} \\ + R_{cost}\big(\mathbf{u}(\tau+i)\big) + & \qquad \textit{Resource cost} \\ + S^{w}_{cost}\big(\mathbf{\Delta_u}(\tau+i)\big) & \qquad \textit{Switching cost} \end{aligned} \tag{10}$$

The *QoS cost* models the user degree of satisfaction with the actual QoS provided by the operator. The *resource cost* models a penalty proportional to the amount of resources/energy consumed. The *switching cost* is a function of the vector $\mathbf{\Delta_u}(\tau) = \mathbf{u}(\tau) - \mathbf{u}(\tau-1)$, which models the penalty incurred in changing the actual configuration.

In the following, various MPC-based strategies will be designed by using different formulations of the three cost terms.

QoS cost. We focus on latency-sensitive applications in which the response time needs to be bounded to some thresholds. Exceeding those requirements must be considered a failure, which may lead to a system malfunction, a loss of revenue or to catastrophic events depending on the type of system controlled. The general objective is to minimize latency as much as possible. However, keeping the latency under a maximum threshold (a critical value for the user satisfaction) is often sufficient in many applications to provide fresh results to the users [31]. We model this requirement with a cost function defined as follows:

$$Q_{cost}\big(\widetilde{\mathbf{q}}(\tau+i)\big) = \alpha \, \exp\left(\frac{\widetilde{R}_Q(\tau+i)}{\delta}\right) \tag{11}$$

where $\alpha > 0$ is a positive cost factor. Such kind of cost heavily penalizes configurations with a latency greater than a threshold $\delta > 0$, as the cost increases rapidly. Therefore, keeping the latency under the threshold as long as possible is fundamental to minimize this cost.

Resource cost. The resource cost is defined as a cost proportional to the number of used replicas or to the power consumed, which in turn depends both on the number of used cores and the employed CPU frequency. We use two cost definitions:

$$R_{cost}\big(\mathbf{u}(\tau+i)\big) = \beta \, n(\tau+i) \qquad \textit{per-core cost} \tag{12}$$

$$= \beta \, \widetilde{\mathcal{P}}(\tau+i) \qquad \textit{power cost} \tag{13}$$

where $\beta > 0$ is a unitary price per unit of resources used (per-core cost) or per watt (power cost).

Switching cost. The switching cost term penalizes frequent and/or large modifications of the actual operator configuration. We use the following definition:

$$S^{w}_{cost}\big(\mathbf{\Delta_u}(\tau+i)\big) = \gamma \left(\left\|\mathbf{\Delta_u}(\tau+i)\right\|_2\right)^2 \tag{14}$$

where $\gamma > 0$ is a unitary price factor. This cost includes the Euclidean norm of the difference vector between the decision vectors used at two consecutive control steps. Quadratic switching cost functions are common in the Control Theory literature [11]. They are used to reduce the number of reconfigurations by avoiding the controller oscillating the configuration used and performing large changes in the values of the decision variables.

4. Runtime Mechanisms

In this paper we describe a prototypal version of our elastic strategies[‡] developed in the `FastFlow` framework [1] for parallel stream processing applications targeting shared-memory architectures. According to the model adopted by this framework, the splitter, replicas, merger and the controller are implemented as separated threads that exchange messages through lock-free shared queues [8]. Messages are memory pointers in order to avoid the overhead of extra copies. The centralized controller is executed by a dedicated control thread. This choice is in line with other approaches proposed recently [23]. Distributed controllers will be studied in the future. The splitter is interfaced with the network through a standard TCP/IP POSIX socket used to receive an unbounded stream of input tuples.

The reconfiguration mechanisms are in charge of applying the new configuration determined by the strategy for the current control step. They will be described in the rest of this section. Although we target shared-memory machines, the runtime interface has been designed in order to be easily implementable in distributed architectures in the near future.

4.1 Increase/Decrease the Number of Replicas

In the case of a change in the parallelism degree, the controller transmits a message containing the new distribution function m^{τ} to the splitter. The controller is responsible for creating the threads supporting the new replicas and for interconnecting them with the splitter, merger and the controller by creating proper shared queues. Furthermore, the splitter transmits *migration messages* to the replicas in order to start the state migration protocol, see Sect. 4.3. Only the replicas involved in the migration will be notified. Similar actions are performed in the case some replicas are removed by destroying the threads implementing the removed replicas and the corresponding `FastFlow` queues.

4.2 Heuristics for Load Balancing

The controller decides a new distribution function in two different cases: *i)* if a new parallelism degree must be used; *ii)* although the parallelism degree is not modified, a new distribution function may be needed to balance the workload among the actual number of replicas. In both the cases finding the best key assignment in terms of load balancing is a *NP-hard* problem equivalent to the *minimum makespan* [30]. Therefore, approximate solutions must be used. The pseudocode of the solution proposed in [30, Chapt. 10] is shown in Alg. 1, in which the distribution function is represented by a lookup table. The keys are ordered by their weight for the next step, see Sect. 3.1.2. Starting from the heaviest key, each key is assigned to the replica with the actual least amount of load. $\mathcal{L}_i(\tau) = \sum_{k|m^{\tau}(k)=i} w_k(\tau)$ denotes the load of the i-th replica.

Other heuristics have been studied in the past. In [28] the authors have tried to achieve good load balancing while minimizing

[‡]The code is available at: `https://github.com/tizianodem/elastic-hft`

Algorithm 1: Computing the distribution table.

Input: list of keys, their weight and the number of replicas N.
Result: a new distribution table.
1 Order the list of keys by their weight;
2 **foreach** *replica* $i \in [1, N]$ **do**
3 $\quad \mathcal{L}_i = 0$;
4 **end**
5 **foreach** $k \in \mathcal{K}$ *in the list* **do**
6 $\quad$ Assign k to replica j s.t. $\mathcal{L}_j(\tau) = \min_{i=1}^{N}\{\mathcal{L}_i(\tau)\}$;
7 $\quad$ Update the load of replica j, i.e. $\mathcal{L}_j(\tau) = \mathcal{L}_j(\tau) + w_k(\tau)$;
8 **end**
9 **return** the new distribution table;

the amount of migrated keys. The idea is to pair very loaded replicas with less loaded ones and exchange keys if the load difference of each pair is higher than a threshold. In Sect. 5.2 we will analyze in detail the impact of the number of moved keys in multicore architectures by comparing different heuristics.

4.3 State Migration

Each time the controller changes the distribution function, some of the data structures supporting the internal state must be migrated. This problem has been addressed in various works from which we have taken inspiration. Our goal is to design a *low-latency state migration protocol*. In particular, during the reconfiguration phase the splitter should not delay the distribution of new input tuples to the replicas not involved in the migration, i.e. they must always be able to process the input tuples without interferences. Furthermore, the replicas involved in the migration should be able to process all the input tuples of the keys not involved in the migration without interruptions.

Our migration protocol consists of several phases. In the first step the splitter receives from the controller the new distribution function m^τ and recognizes the migrated keys. It transmits a sequence of *migration messages* belonging to two different classes:

- `move_out(k)` is sent to the replica r_i that held the data structures corresponding to the tuples with key $k \in \mathcal{K}$ before the reconfiguration but will not hold them after the reconfiguration, i.e. $m^{\tau-1}(k) = i \wedge m^\tau(k) \neq i$;
- `move_in(k)` is sent to the replica r_j that will hold the data structures associated with key k after the reconfiguration (and did not own them before), i.e. $m^{\tau-1}(k) \neq j \wedge m^\tau(k) = j$.

The splitter sends new tuples using the new distribution function without blocking, while the state movements are carried out by the replicas in parallel. In contrast, in [14, 17] the operator experiences a downtime until the migration is complete by blocking some of the replicas and/or the splitter. Like in [15, 28] the replicas not involved in the migration maintain the same subset of keys and are able to process the input tuples without interferences.

Fig. 3 depicts the reconfiguration actions in the case of the migration of a key $k \in \mathcal{K}$ from replica r_i to r_j ①. At the reception of a `move_out(k)` message by replica r_i ②, that replica knows that it will not receive tuples with key k anymore. Therefore, it can safely save the state of that key (denoted by s_k) to a *backing store* used to collect the state partitions of the migrated keys and to synchronize the pairs of replicas involved in the migration.

The replica r_j, which receives the `move_in(k)` message ③, may receive new incoming tuples with that key before the state has been acquired from the backing store. Only when r_i has properly saved s_k into the repository ④ the state can be acquired by r_j ⑤. However, r_j is not blocked during this time period but accepts new tuples without interruptions. All the tuples with key k received in

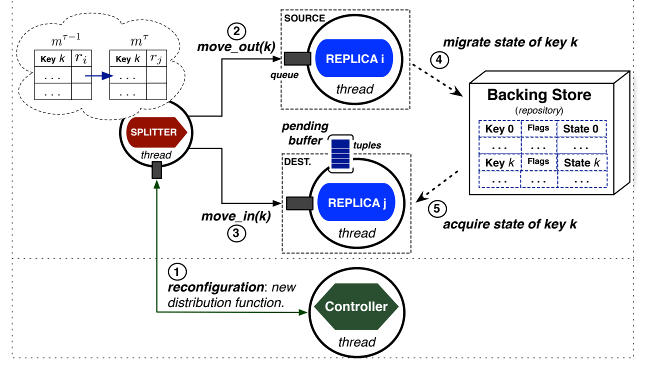


Figure 3: Example of state migration between replica r_i and replica r_j.

the meanwhile are enqueued in a temporary *pending buffer* until s_k is available. When that state is ready to be fetched, it is acquired by the replica and all the pending tuples of key k in the buffer are rolled out and processed in the same order in which they were sent by the splitter. This solution is similar to the one described in [28], except that the pending buffers are maintained in the replicas involved in the migration instead of having a centralized buffer kept in the splitter. Therefore, the splitter is able to route tuples to the replicas without delays introduced by the dequeuing of tuples from the pending buffer.

In our case, since the replicas are executed on the same shared-memory node *(A1)*, the repository is a shared memory area in which replicas exchange memory references to the data structures by avoiding the copy overhead. Instead, in the case of a distributed implementation the backing store can be implemented by back-end databases or using socket-/MPI-based implementations [14].

4.4 Frequency Scaling and Energy Measurement

Our MPC-based strategies can change the current operating frequency used by the CPUs. This does not affect the structure of the parallelization and can be performed transparently. To this end, the runtime uses the `C++ Mammut` library[§] (MAchine Micro Management UTilities) targeting off-the-shelf multicore CPUs. The controller uses the library functions to collect energy statistics. On Intel Sandy-Bridge CPUs this is performed by reading the Running Average Power Limit (RAPL) energy sensors that provide accurate fine-grained energy measurements [16]. On the same way, voltage values are read through specific model-specific registers (MSR).

5. Experiments

In this section we evaluate our control strategies on a DaSP application operating in the high-frequency trading domain (briefly, HFT). The code has been compiled with the `gcc` compiler (version 4.8.1) with the `-O3` optimization flag. The target architecture is a dual-CPU Intel Sandy-Bridge multicore with 16 cores with 32GB or RAM. Each core has a private L1d (32KB) and L2 (256KB) cache. Each CPU is equipped with a shared L3 cache of 20MB. The architecture supports DVFS with a frequency ranging from 1.2GHz to 2GHz in steps of 0.1GHz. In the experiments the TurboBoost feature of the CPU has been turned off. Each thread of the implementation has been pinned on a distinct physical core and hyper-threading CPU facility is not exploited.

[§]The Mammut library is open source and freely available at `https://github.com/DanieleDeSensi/Mammut`

5.1 Application Description

High-frequency trading applications ingest huge volume of data at a great velocity and process them with very stringent QoS requirements. Fig. 4 shows the kernel of the application that we use for the experiments.

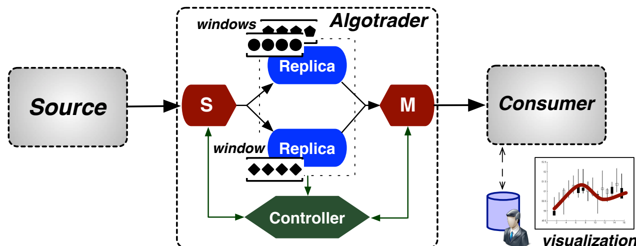


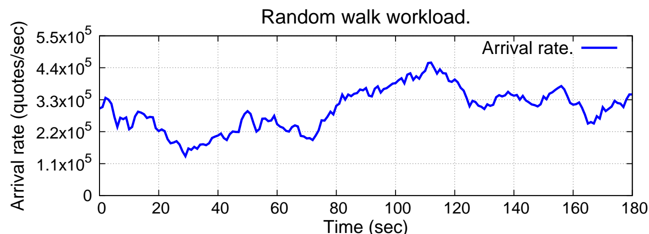
Figure 4: Kernel of a high-frequency trading application.

The *source* generates a stream a financial quotes, i.e. buy and sell proposals (bid and ask) represented by a record of attributes such as the proposed price, volume and the stock symbol (64 bytes in total). The *algotrader* operator processes quotes grouped by the stock symbol. A count-based sliding window of size $|\mathcal{W}| = 1,000$ quotes is maintained for each group. After receiving a slide of $\delta = 25$ new tuples of the same stock symbol, the computation processes all the tuples in the corresponding window. The operator aggregates quotes with a millisecond resolution interval and applies a prediction model aimed at estimating the future price of that stock symbol. The aggregate quotes (one per resolution interval) in the current window are used as input of the Levenberg-Marquardt regression algorithm to produce a fitting polynomial. For the regression we use the C++ library `lmfit` [6]. The results are transmitted to the *consumer* that consolidates them in a DB. The algotrader and the consumer are executed on the same machine while the source can be allocated on a different host, as it communicates with the algotrader through a TCP socket. The user can choose the parameters $|\mathcal{W}|$ and δ in order to achieve a desired level of results accuracy. The values used in the experiments are typical examples. In our evaluation we do not consider the consumer. We have dedicated 4 cores for the source, splitter, merger and the controller entities, which leaves 12 cores for replicas.

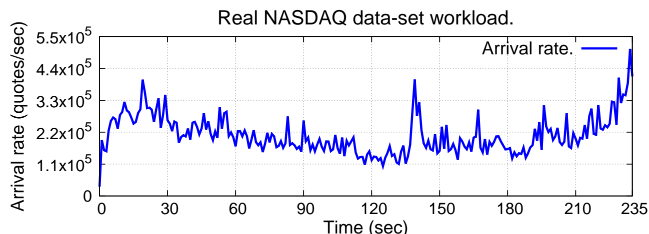
This application has all the variability issues introduced in Sect. 2.1. The source generates inputs with a time-varying arrival rate and frequency distribution of the stock symbols (*D1* and *D2*). Furthermore, the aggregation phase can change the number of quotes that are inputs of the regression. Therefore, also the computation time per window can vary during the execution *(D3)* analogously to the case in which time-based windows are used. We refer to a *synthetic workload* (Fig. 5a) and a *real dataset* scenario (Fig. 5b). In the latter, the quotes and their timestamps are the ones of a trading day of the NASDAQ market (daily trades and quotes of 30 Oct 2014[¶]) with 2,836 traded stock symbols. This dataset has a peak rate near to 60,000 quotes per second, with an average value well below this figure. To model future scenarios of millions of quotes per second (as in the case of options instead of quotes), we accelerate it 100 times to increase the need of parallelism in the algotrader. Fig. 5b shows the accelerated real workload.

Fig. 5a shows the arrival rate per second in the synthetic benchmark. The rate follows a random walk model and the key frequency distribution is fixed and equal to a random time instant of the real NASDAQ dataset. The skewness factor, i.e. the ratio between the most probable key and the less probable one, is equal to 9.5×10^4.

[¶]The dataset can be downloaded at the following url: `http://www.nyxdata.com`



(a) Random walk.



(b) Real dataset.

Figure 5: Arrival rate in the synthetic and real workload scenarios.

The execution of the synthetic benchmark consists in 180 seconds, while the one of the real dataset (Fig. 5b) is of 235 seconds equal to about 6 hours and half in the original non-throttled dataset.

5.2 Implementation Evaluation

The first set of experiments is aimed at analyzing our parallel implementation of the algotrader operator in order to assess the efficiency and the effectiveness of our design choices.

Overhead of the elastic support. To evaluate the impact of our elastic support we have measured the maximum input rate that the algotrader is able to sustain with the highest CPU frequency and the maximum number of replicas. We compare the implementation with the elastic support, i.e. the controller functionality and all the monitoring activities enabled (sampling interval of 1 second), with the one in which the elastic support has been turned off. In this experiment the algotrader is fed by a high-speed stream of quotes belonging to 2,836 keys all with the same frequency. The results of this experiment show a modest overhead in the algotrader ideal service time, bounded by $3 \div 4\%$. This demonstrates that the monitoring activity and the asynchronous interaction with the controller have a negligible effect on the computation performance.

Migration overhead. The MPC controller adopts the heuristic described in Alg. 1 to find a new distribution function. This heuristic (called *balanced* in the rest of the description) may move most of the keys in order to balance the workload perfectly among the replicas. We compare it with the heuristic studied in [28] (we call it *Flux*), which minimizes the number of migrated keys by keeping the unbalance under a threshold (we use 10%).

We analyze a scenario in which the workload is perfectly balanced until timestamp 30. At that time instant we force the input rate to suddenly change from 3×10^5 quotes/sec to 4×10^5 quotes/sec. The strategy detects the new rate and changes the number of replicas from 6 to 8 at timestamp 31. In this case the controller does not modify the CPU frequency. The steady-state behavior of the new configuration is reached at timestamp 36. Fig. 6 shows the average latency measured each 250 milliseconds (1/4 of the control step).

We identify three phases: a *first phase* in which the rate changes and the controller detects it at the next control step. During this phase both the heuristics produce very similar latency results that

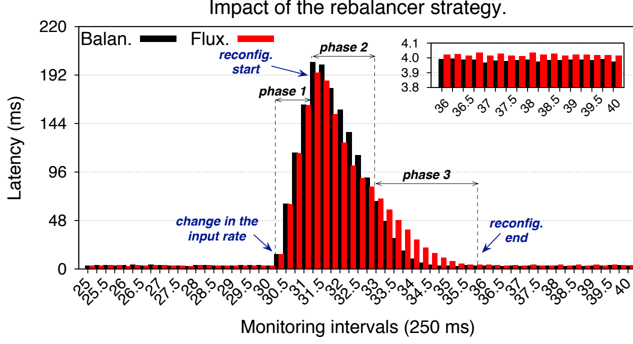


Figure 6: Impact of the rebalancing heuristics: latency measurements in the time instants before, during and after a reconfiguration.

grows because the operator becomes a bottleneck. In the *second phase* the controller triggers the reconfiguration and some keys are migrated ($2,460$ and 68 in the Balanced and the Flux heuristics). The Flux heuristic produces lower latency results because pending buffers are mostly empty during this phase. In the *third phase* (after timestamp 33), the replicas process the tuples that were waiting in the pending buffers. Owing to a better load balancing, the heuristic of Alg. 1 approaches the steady state faster. During the third phase the measured latency with the Flux heuristic is about double of the latency with the balanced one. Furthermore, the steady-state behavior after timestamp 36 shows a slight advantage ($1 \div 2\%$) compared with Flux (subplot in Fig. 6).

In conclusion, on multicores the balanced heuristic should be preferred in latency-sensitive applications since the latency spikes during the migration are modest and, notably, the new steady state can be reached faster. Of course this consideration does not hold in distributed-memory architectures, in which minimizing the number of moved keys is important because the state partitions are copied among replicas, and not transferred by reference.

On the optimization complexity. As an ancillary aspect, we note that the MPC optimization requires to explore all the admissible reconfiguration trajectories, whose number is $\mathcal{N}^h \times |\mathcal{F}|^h$. Therefore, we have an exponential increase with an increasing number of reconfiguration options and longer prediction horizons. To complete the optimization in a time negligible with respect to the control step, we use a Branch & Bound procedure that reduces the number of explored states of several orders of magnitude with respect to the theoretical number of trajectories. Details will be provided in future extensions of this paper.

5.3 Control Strategies Evaluation

In this section we study different MPC-based strategies based on the different formulations of the optimization problem proposed in Sect. 3.2. Tab. 2 summarizes them. The cost parameters α, β and γ require to be tuned properly. We use $\beta = 0.5$ while we adopt a different value of α according to the used workload trace. Each strategy is evaluated without switching cost ($\gamma = 0$) or with the switching cost (we use $\gamma = 0.4$) and different lengths $h \geq 1$ of the prediction horizon.

Name	Resource cost	alpha (rw)	alpha (real)
`Lat-Node`	per node (Expr. 12)	2	3
`Lat-Power`	power cost (Expr. 13)	2	4

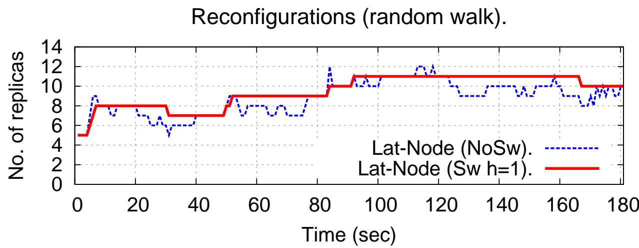
Table 2: Different MPC-based strategies studied in the experiments.

The arrival rate is predicted using a Holt Winters filter [13] able to produce h-step ahead forecasts by taking into account trend and cyclic nonstationarities in the underlying time-series; the frequencies and the computation times per key are estimated using the last measured values, i.e. $\widetilde{p}_k(\tau + i) = p_k(\tau - 1)$ and $\widetilde{\mathcal{C}}_k(\tau + i) = \mathcal{C}_k(\tau - 1)$ for $i = 0, \ldots, h - 1$. We use the balanced heuristic to derive the new distribution function at each step.

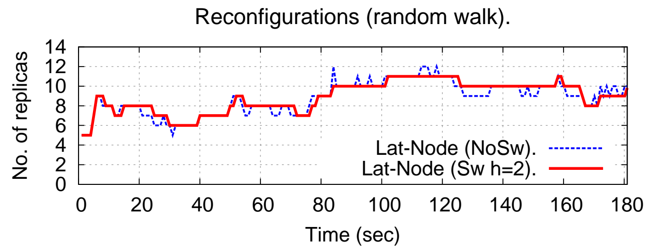
All the experiments have been repeated 25 times by collecting the average measurements. The variance of the results is very small: in some cases we will show the error bars.

5.3.1 Reconfigurations and effect of the switching cost

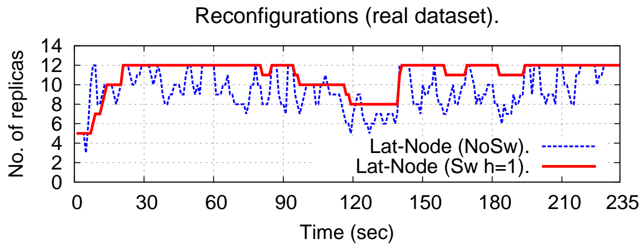
Fig. 7 shows the number of replicas used by the algotrader operator with the `Lat-Node` strategy without the switching cost (NoSw) and



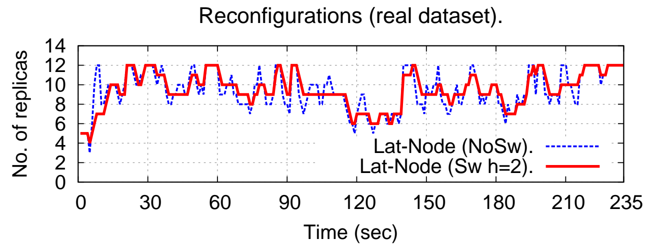
(a) `Lat-Node` NoSw vs. Sw ($h = 1$).



(b) `Lat-Node` NoSw vs. Sw ($h = 2$).



(c) `Lat-Node` NoSw vs. Sw ($h = 1$).



(d) `Lat-Node` NoSw vs. Sw ($h = 2$).

Figure 7: Used replicas per control step (1 sec). `Lat-Node` strategy: comparison without switching cost and with the switching cost and $h = 1, 2$.

with the switching cost (Sw) and different horizon lengths $h = 1, 2$. The reconfiguration sequence reflects the workload variability. The strategy changes only the number of replicas (the CPU frequency is fixed to 2 Ghz). Qualitatively similar results are achieved with the `Lat-Power` strategy (we omit them for the sake of space).

The combined effect of the switching cost and the horizon length is evident. The dashed line corresponds to the reconfigurations without switching cost ($\gamma = 0$ and $h = 1$) in which the cost function $\mathcal{J}$ is composed of the QoS cost and the resource cost terms. The MPC controller selects the number of replicas that minimizes this function by following the workload variability. It is worth noting that a prediction horizon longer than one step is useless without the switching cost enabled.

The solid line corresponds to the strategy with the switching cost enabled, which acts as a *stabilizer* by smoothing the reconfiguration sequence, i.e. it is a brake that slows the acquisition/release of replicas. By increasing the foresight of the controller the reconfigurations with the switching cost better approximate the sequence without the switching cost. The reason is that our disturbance forecasts are able to capture future increasing/decreasing trends in the arrival rate (Holt-Winters). During increasing trends, longer horizons allow the controller to anticipate the acquisition of new replicas. The opposite characterizes decreasing trends. Therefore, by increasing the horizon length the effect of the stabilizer is less intensive and a faster adaptation to the workload can be observed.

Fig. 8 summarizes the total number of reconfigurations. More reconfigurations are performed with the real workload, because of a higher variability of the arrival rate. Furthermore, more reconfigurations are needed by the energy-aware strategy `Lat-Power`. In fact, the space of possible reconfiguration options is larger, as the controller can also change the CPU frequency. In the number of reconfigurations we count any change in the number of replicas and/or in the CPU frequency. We do not consider the changes that affect only the distribution function.

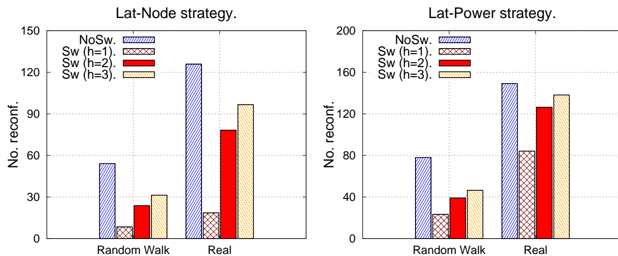


Figure 8: Number of reconfigurations.

The `Lat-Power` strategy can vary the number of replicas, the CPU frequency or both. Fig. 9 shows the types of reconfigurations performed. In general, changes in the CPU frequency are more frequent than reconfigurations of the number of replicas.

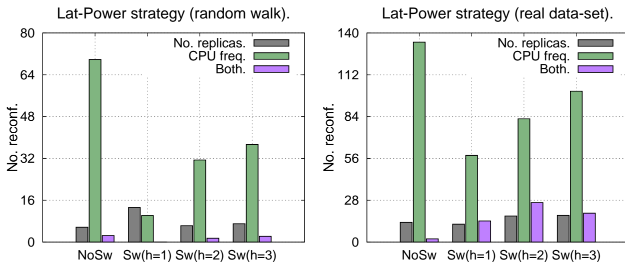


Figure 9: Types of reconfigurations.

In conclusion we can state the following important property: *the switching cost term allows the strategy to reduce the number and frequency of reconfigurations (P1). This effect is partially mitigated by increasing the horizon length.*

5.3.2 QoS violations

We detect a QoS violation each time the average latency measured during a control step is higher than a user-defined threshold θ. Figs. 10a, 10b and 10c show the latency violations achieved by the `Lat-Power` strategy in the random walk scenario with $\theta = 1.5$ ms. The figure reports the average latency per second without switching cost ($h = 1$) and with switching cost ($h = 1, 2$). We detect a QoS violation each time the solid line crosses the dashed line. The 95% confidence intervals (red intervals in the figure) are very small, demonstrating the small variance of the measurements.

Without switching cost we have more violations. In fact, the strategy chooses each time the minimal (smaller) configuration such that the latency threshold is respected. If the arrival rate predictions are underestimated we obtain over-threshold latencies. We achieve fewer violations by using the switching cost and the minimum horizon. This is an expected result, as this strategy overshoots the configuration *(P4)*. This is clear in Fig. 7a and 7c, where the reconfiguration sequences with switching cost are mostly on top the ones without it. This allows the strategy to be more capable of dealing with workload underestimation. Longer horizons provide intermediate results. Figs. 10d, 10e and 10f show the results with the real workload in which we use a higher threshold of 7 ms. We measure more violations because the input rate variability is higher and predictions less accurate.

Fig. 11 summarizes the results. In conclusion, *the switching cost allows the strategy to reach better accuracy (P2). This positive effect is partially offset by increasing the horizon length.*

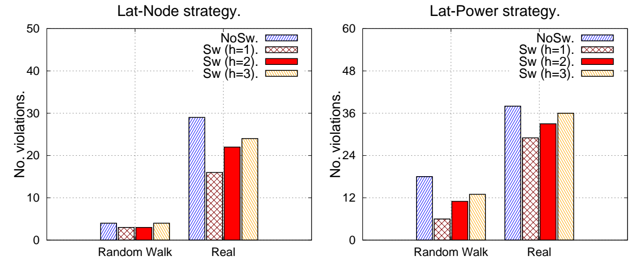


Figure 11: Number of QoS violations.

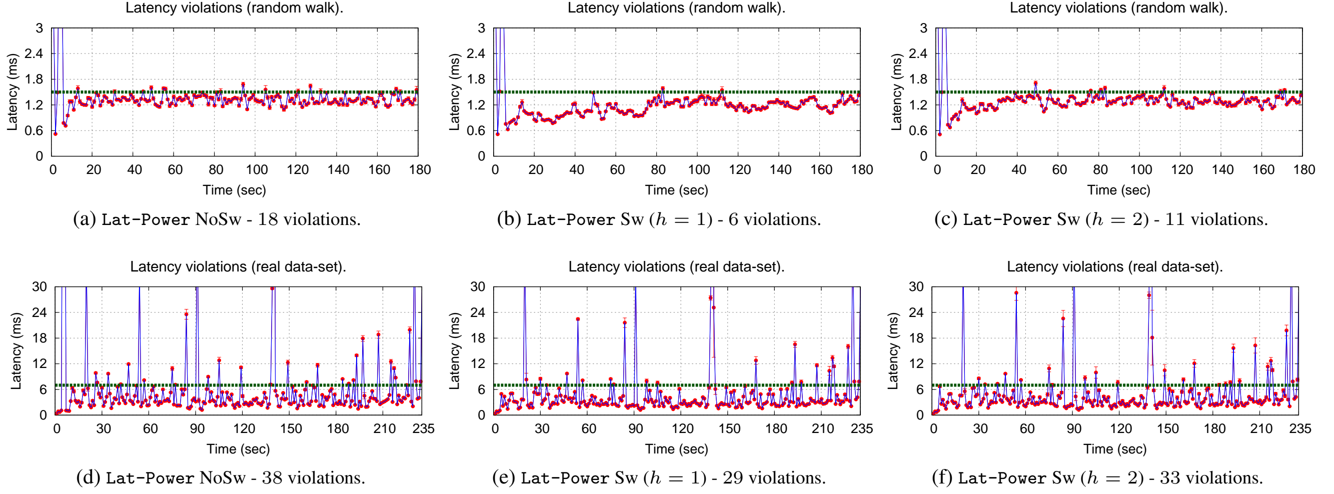
5.3.3 Resource and power consumption

We study the power consumption of the `Lat-Node` and the `Lat-Power` strategies. We consider the power dissipated by the cores, while the other components inside the chip (L3 cache) and off-chip (RAM) represent a constant term in the overall power consumption[‖].

We use the same frequency for the two chips of our multicore *(A1)*. Fig. 12 shows the watts without using the switching cost. We compare the strategies in which the controller changes the number of replicas configured at the highest frequency (`Lat-Node`) with the strategy using frequency scaling (`Lat-Power`). Fig. 12a shows the results in the random walk scenario. The watts with the energy-aware strategy (green lines) always stay below the consumption without frequency scaling, resulting in an average power saving of 18.2%. A similar behavior can be observed in Fig. 12b for the real workload where `Lat-Power` saves $10 \div 11$ watts (16.5% on average) than `Lat-Node`. This reduction is significant owing to the long-running nature of DaSP applications.

Fig. 13 summarizes the results. With longer horizons the strategies with switching cost consume more resources/power. For the

[‖] We measure the core counter. The overall socket consumption consists in additional $25 \div 30$ watts per step.

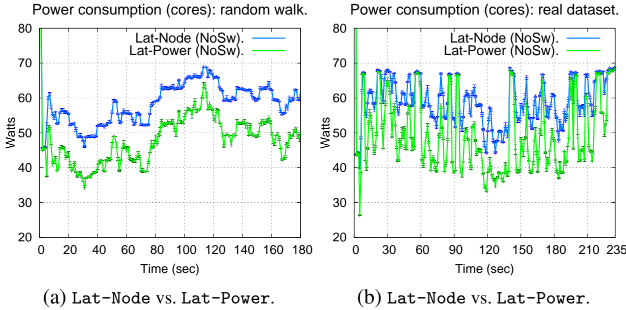


(a) `Lat-Power` NoSw - 18 violations. (b) `Lat-Power` Sw ($h = 1$) - 6 violations. (c) `Lat-Power` Sw ($h = 2$) - 11 violations.

(d) `Lat-Power` NoSw - 38 violations. (e) `Lat-Power` Sw ($h = 1$) - 29 violations. (f) `Lat-Power` Sw ($h = 2$) - 33 violations.

Figure 10: Latency violations. Strategy `Lat-Power` with random walk workload (a, b, c) and the real dataset (d, e, f).

real workload `Lat-Node` uses 14.76%, 1.87% and 0.94% more resources (respectively for $h = 1, 2, 3$) with respect to the strategy without switching cost. `Lat-Power` has an additional power consumption of 5.68%, 3.04% and 1.12% respectively.



(a) `Lat-Node` vs. `Lat-Power`. (b) `Lat-Node` vs. `Lat-Power`.

Figure 12: Power consumption (watts) of the `Lat-Power` and `Lat-Node` strategies without switching cost: random walk and real workload.

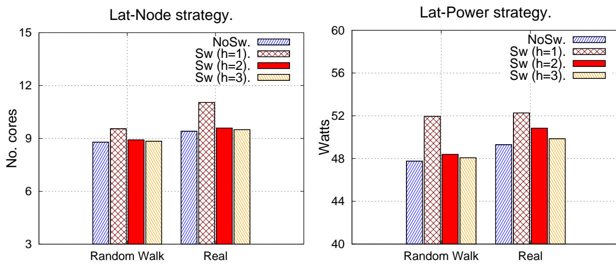


Figure 13: Resources/power consumed.

From these results we can derive the following important property of our strategies: *the use of the switching cost causes overshoot (P4). This can be mitigated by using a longer horizon*.

5.3.4 Reconfiguration amplitude

In cases of sudden workload changes, the strategy should be able to reach rapidly a configuration that meets the QoS requirements. If fews replicas are added/removed each time, this negatively impacts the settling time property as several reconfigurations are needed to reach the configuration able to achieve the desired QoS. Fig. 14 shows for each strategy the so-called *reconfiguration amplitude* measured over the entire execution. It consists in the Euclidean distance between the decision vector $\mathbf{u}(\tau)$ used at step τ and the one used at the previous step $\mathbf{u}(\tau - 1)$. The frequency values (from 1.2 Ghz to 2 Ghz with steps of 0.1) have been normalized using the rule $(f(\tau) - 1.2) * 10 + 1$ to obtain the integers from 1 to 9.

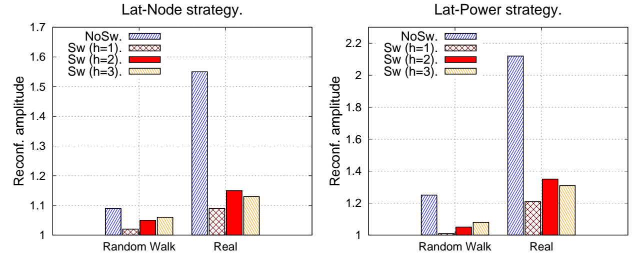


Figure 14: Settling time: random walk and real workload.

As it is evident from Fig. 14, the strategies with switching cost and $h = 1$ perform smaller reconfigurations. The highest amplitude is achieved by the strategy without the switching cost, which follows accurately the workload variations (more intensive in the real dataset). Therefore, *the switching cost reduces the average reconfiguration amplitude while better settling time (P3) can be achieved with longer prediction horizons*.

5.4 Comparison with Peak-load Overprovisioning

In this section we compare the best results achieved by our strategies against a static configuration in which the algotrader is configured to sustain the peak load. In this scenario state migrations are eventually performed at each step to maintain the workload balanced among replicas. However, the number of replicas and the CPU frequency are statically set to the maximum value throughout the execution. The results are depicted in Fig. 15.

The peak-load configuration allows to achieve the minimum number of QoS violations both in the synthetic and the real workload traces. With the real workload we have 7 and 18 more violations achieved by `Lat-Node` and `Lat-Power` respectively. How-

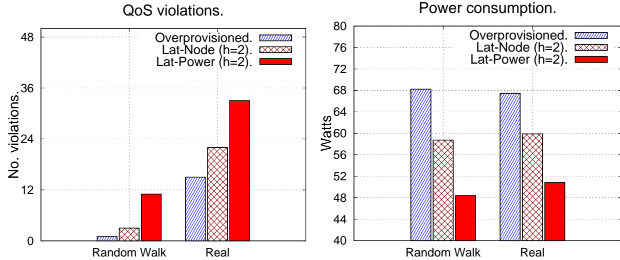


Figure 15: QoS violations and power consumption: comparison with the static peak-load overprovisioning strategy.

ever, this static configuration has the highest power consumption. The relative power savings for the synthetic and real workload are respectively of 14% and 12% with `Lat-Node` and 29% and 25% with `Lat-Power`.

5.5 Comparison with Similar Approaches

We conclude this paper by comparing our approach with two reactive strategies. The first is based on policy rules. It changes the number of replicas if the relative error between the measured latency and a required one is over a threshold. The second is the control algorithm described in [14] developed for the SPL framework [2] and targeting the SASO properties. This strategy changes the number of replicas based on a *congestion index* (i.e. the fraction of time the splitter is blocked in sending tuples to the replicas) and the recent history of the past actions. If a recent action has been taken without improving throughput, the control algorithm avoids executing it again. To adapt to fluctuating workload, the authors introduce mechanisms to forget the recent history if the congestion index or the throughput change significantly. The algorithm uses a *sensitivity* parameter to determine what a significant change means. Like for our parameters α, β and γ, tuning is required for the index and sensitivity parameters. Both the reactive strategies do not change the CPU frequency.

Tab. 3 shows a summary of the results for the real dataset scenario. We compare the results achieved by the two reactive strategies against our `Lat-Node` strategy with $h = 2$ (the best horizon in terms of SASO trade-off). We chose this strategy in order to change only the number of replicas in all the cases. For the comparison we change the control step length to 4 seconds because the SPL strategy behaves poorly with too small sampling intervals. This is a first evident shortcoming of this approach, which is unable to adapt to the workload with a fine-grained sampling. We found that the best values for the congestion index and the sensitivity parameter, according to the authors indications [14], are near to 0.1 and 0.9 respectively.

	No. reconf.	QoS viol.	No. replicas
Rule-based	47.42	76	6.89
SPL-strategy	40.18	230	4.63
Lat-Node	11.0	30	9.97
Peak-load	-	15	12

Table 3: Comparison: average values with 25 tests per strategy.

The results show that our approach is the winner. Fewer reconfigurations are performed (stability) with fewer QoS violations. The SPL strategy is essentially throughput oriented. Therefore, it gives no latency guarantee and produces more QoS violations. Our strategy obtains fewer violations by needing a slightly higher number of resources than the rule-based strategy. This is a confirm of the effectiveness of our approach and of its ability to respect latency requirements with minimal resource consumption.

6. Related Work

Nowadays SPEs are widely used, both through academia prototypes [12, 15] and open-source [3, 4] and industrial products [2, 7].

Elasticity is a recent feature of SPEs, based on the automatic adaptation to the actual workload by scaling up/down resources. Most of the existing strategies are reactive. In [12, 15, 17] the authors use threshold-based rules on the actual CPU utilization by adding or removing computing resources accordingly. Other works use complex metrics to drive the adaptation. In [23] the mean and standard deviation of the service time and the inter-arrival time are used to enforce latency constraints without predictive capabilities. In [14] the strategy measures a congestion index. By remembering the effects of the recent actions, the control algorithm avoids taking reconfigurations without improving throughput. Analogously to our work, this approach tries to target the SASO properties in the DaSP domain. However, our approach proposes a model-based predictive approach instead of a heuristic-based reactive one.

Other approaches try to anticipate the future by reconfiguring the system to avoid resource shortages. As far as we know, [22] is the first work trying to apply a predictive approach. They leverage on the knowledge of future resources and workload to plan resource allocation. The approach has been evaluated with oracle models that give exact predictions. Some experiments take into account possible prediction errors, but they do not use real forecasting tools. Moreover, the authors do not evaluate the SASO properties.

All the mentioned works except [23] are not optimized for low latency. In our past works we have investigated the use of MPC in the streaming context [24, 25] by focusing on the system throughput. In contrast, the strategies proposed in this paper explicitly takes into account latency constraints. In [17] the authors have studied how to minimize latency spikes during the state migration process. We have also studied this problem in this paper. In addition, we use a latency model to drive the choice of future reconfigurations by making our strategy fully latency-aware.

All the existing works take into account the number of nodes used. Our strategies are designed to address power consumption on DVFS-enabled CPUs. In [27] an energy minimization approach is used to properly annotate OpenMP programs. Therefore, it is not directly suitable for the streaming context. A work addressing power consumption in DaSP is [29]. Here the problem is tackled by a scheduler of streaming applications on the available machines. Thus, it does not propose scaling strategies.

7. Conclusions and Future Work

This paper presented a predictive approach to elastic data stream operators on multicores. The approach has two main novel aspects: it applies the Model Predictive Control method in the domain of data stream processing; it takes into account power consumption issues while providing latency guarantees. We validated our approach in a high-frequency trading application.

The high-frequency trading domain is a good candidate for the evaluation, since automatic trading applications have usually stringent QoS requirements in terms of worst-case latency bounds. However, our approach is not limited to this use case. The MPC methodology can be particularly beneficial in all the application domains of DaSP in which QoS violations are considered dangerous events that must be avoided to ensure a correct system behavior. Other examples are healthcare diagnostic systems that process sensor data in real-time to anticipate urgent medical interventions, and transportation monitoring systems, in which sensor data are analyzed to detect anomalous behaviors and to prevent catastrophic

scenarios. In these application contexts performance guarantees are fundamental, and a proactive strategy enabling elastic processing is of great importance to meet the performance requirements with high probability by reducing the operating costs.

In the future we plan to extend our work on shared-nothing machines (clusters). Furthermore, we want to integrate our MPC-based strategies in a complete graph context, in which different operators need to coordinate to find agreements in their reconfiguration decisions. Distributed optimization and Game Theory are possible theoretical frameworks to solve this problem.

Acknowledgments

This work has been partially supported by the EU H2020 project RePhrase (EC-RIA, ICT-2014-1). We want to thank Prof. Marco Vanneschi to have been an invaluable reference during our work.

References

[1] Fastflow (ff). `/http://calvados.di.unipi.it/fastflow/`.

[2] Ibm infosphere streams. `http://www-03.ibm.com/software/products/en/infosphere-streams`.

[3] Apache spark streaming. `https://spark.apache.org/streaming`.

[4] Apache storm. `https://storm.apache.org`.

[5] Enhanced intel speedstep technology for the intel pentium m processor, 2004. URL `ftp://download.intel.com/design/network/papers/30117401.pdf`.

[6] Joachim wuttke: lmfit a c library for levenberg-marquardt least-squares minimization and curve fitting, 2015. URL `http://apps.jcns.fz-juelich.de/lmfit`.

[7] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536229.

[8] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 662–673, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-32819-0.

[9] H. Andrade, B. Gedik, and D. Turaga. *Fundamentals of Stream Processing*. Cambridge University Press, 2014. ISBN 9781139058940.

[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-507-6.

[11] E. F. Camacho and C. Bordons, editors. *Model predictive control*. Springer-Verlag, Berlin Heidelberg, 2007.

[12] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM. doi: 10.1145/2463676.2465282.

[13] R. Fried and A. George. Exponential and holt-winters smoothing. In M. Lovric, editor, *International Encyclopedia of Statistical Science*, pages 488–490. Springer Berlin Heidelberg, 2014.

[14] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1447–1463, June 2014. ISSN 1045-9219.

[15] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, Dec. 2012. ISSN 1045-9219.

[16] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, Jan. 2012. ISSN 0163-5999. doi: 10.1145/2425248.2425252.

[17] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2737-4.

[18] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[19] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 187–198, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1636-1. doi: 10.1145/2479871.2479899.

[20] W. Hummer, B. Satzger, and S. Dustdar. Elastic stream processing in the cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(5):333–345, 2013. ISSN 1942-4795.

[21] J. F. C. Kingman. On queues in heavy traffic. *Journal of the Royal Statistical Society. Series B (Methodological)*, 24(2):pp. 383–392, 1962.

[22] A. Kumbhare, Y. Simmhan, and V. Prasanna. Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 344–353, May 2014. doi: 10.1109/CCGrid.2014.60.

[23] B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. In *The 35th International Conference on Distributed Computing Systems (ICDCS 2015)*, page to appear, 2015.

[24] G. Mencagli, M. Vanneschi, and E. Vespa. Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 11–18, July 2013. doi: 10.1109/HPCSim.2013.6641387.

[25] G. Mencagli, M. Vanneschi, and E. Vespa. A cooperative predictive control approach to improve the reconfiguration stability of adaptive distributed parallel applications. *ACM Trans. Auton. Adapt. Syst.*, 9 (1):2:1–2:27, Mar. 2014. ISSN 1556-4665. doi: 10.1145/2567929. URL `http://doi.acm.org/10.1145/2567929`.

[26] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 35–44, New York, NY, USA, 2002. ACM. ISBN 1-58113-483-5.

[27] 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 *Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures*, PARMA-DITAM '15, pages 19–24, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3343-6. doi: 10.1145/2701310.2701311.

[28] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36, March 2003.

[29] D. Sun, G. Zhang, S. Yang, W. Zheng, S. U. Khan, and K. Li. Re-stream: Real-time and energy-efficient resource scheduling in big data stream computing environments. *Information Sciences*, 319:92 – 112, 2015. ISSN 0020-0255. doi: http://dx.doi.org/10.1016/j.ins.2015.03.027.

[30] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001. ISBN 3-540-65367-8.

[31] U. Verner, A. Schuster, and M. Silberstein. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 120–129, New York, NY, USA, 2011. ACM.