Proactive Elasticity and Energy Awareness in Data Stream Processing

Tiziano De Matteis and Gabriele Mencagli*

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

Abstract

Data stream processing applications have a long running nature (24hr/7d) with workload conditions that may exhibit wide variations at run-time. *Elasticity* is the term coined to describe the capability of applications to change dynamically their resource usage in response to workload fluctuations. This paper focuses on strategies for elastic data stream processing targeting multicore systems. The key idea is to exploit *Model Predictive Control*, a control-theoretic method that takes into account the system behavior over a future time horizon in order to decide the best reconfiguration to execute. We design a set of energy-aware proactive strategies, optimized for throughput and latency QoS requirements, which regulate the number of used cores and the CPU frequency through the *Dynamic Voltage and Frequency Scaling* (DVFS) support offered by modern multicore CPUs. We evaluate our strategies in a high-frequency trading application fed by synthetic and real-world workload traces. We introduce specific properties to effectively compare different elastic approaches, and the results show that our strategies are able to achieve the best outcome.

Keywords: Data Stream Processing, Elasticity, Model Predictive Control, Frequency Scaling.

1. Introduction

Data Stream Processing [1] (hereinafter DaSP) is a computing paradigm enabling the online analysis of live data streams processed under strict Quality of Service (QoS) requirements. These applications usually provide real-time notifications and alerts to the users in domains like environmental monitoring, high-frequency trading, network intrusion detection and social media.

Elasticity in DaSP is a vivid and recent research field. It consists in providing mechanisms to adapt the used resources in cases in which the workload fluctuates intensively. Such mechanisms are able to scale up/down the used resources on demand, based on the actual monitored performance [2]. This problem has been studied in the last years, with works proposing elastic approaches both for single nodes and distributed environments [3, 4, 5]. A review of these solutions is described in Sect. 6.

This paper provides advanced strategies that fill missing aspects of the existing work. Most of the elastic supports are *reactive* [3, 4, 5, 6], i.e. they take corrective actions based on the actual QoS measurements. In this paper we present *predictive* strategies that try to anticipate QoS violations. Furthermore, most of the existing approaches (see Sect. 6) are *throughput-oriented* and do not take into account explicitly the processing *latency* as the main parameter to trigger reconfigurations. In this paper we propose strategies that address both throughput and latency constraints. Finally, the existing approaches do not face *energy/power consumption* issues. In this paper we tackle this problem by targeting multicore CPUs with *Dynamic Voltage and Frequency Scaling* (DVFS) support.

The proactivity of our approach has been enforced using a control-theoretic method known as *Model Predictive Control* [7] (MPC), in which the system behavior over a future time horizon is accounted for deciding the best reconfigurations to execute. As far as we know, this is the first time that MPC has been used in the DaSP domain.

A first version of this work has been published in Ref. [8]. This paper extends this preliminary work by presenting two energy-aware strategies with different resource/power usage characteristics: the first targets *high-throughput*, while the second is oriented toward *low-latency* workload. Furthermore, we provide a detailed analysis of our runtime mechanisms for elasticity and a comparison with state-of-the-art techniques. Finally, in this paper we specifically study the complexity issues related to the online execution of our adaptation strategies by presenting a Branch& Bound approach to deal with this problem.

The outline of this paper is the following. Sect. 2 provides a brief overview of DaSP. Sect. 3 describes our strategies. Sect. 4 shows the details of the reconfiguration mechanisms. Sect. 5 analyzes our strategies and compares them with the state-of-the-art. Finally, Sect. 6 reviews similar research works and Sect. 7 concludes this paper.

2. Overview of Data Stream Processing

DaSP applications are structured as data-flow graphs [1] of core functionalities, where vertices represent *operators*

*Corresponding author, Phone: +39-050-2213132
Email addresses: `{dematteis, mencagli}@di.unipi.it` (Tiziano De Matteis and Gabriele Mencagli)

connected by arcs modeling *data streams*, i.e. unbounded sequences of data items (*tuples*).

Important in DaSP are stateful operators [1] that maintain an internal state while processing input tuples. A typical case is represented by *partitioned-stateful operators* [1], which are applied when the input stream conveys tuples belonging to different logical substreams. In that case, the operator can maintain a different internal state for each substream. Examples are operators that process network traces partitioned by IP address, or market feeds partitioned by a stock symbol attribute.

Owing to the fact that the significance of each input tuple is often time-decaying, the internal state can be represented by the most recent portion of each substream stored in a *sliding window* [1]. The window boundaries can be (*time-based*) or (*count-based*).

2.1. Intra-operator Parallelism

In this work we study elasticity for parallel partitioned-stateful operators, which represent the target of the most recent research [2, 9]. A parallel operator is composed of several functionally equivalent *replicas* [1] that handle a subset of the input tuples, as sketched in Fig. 1.

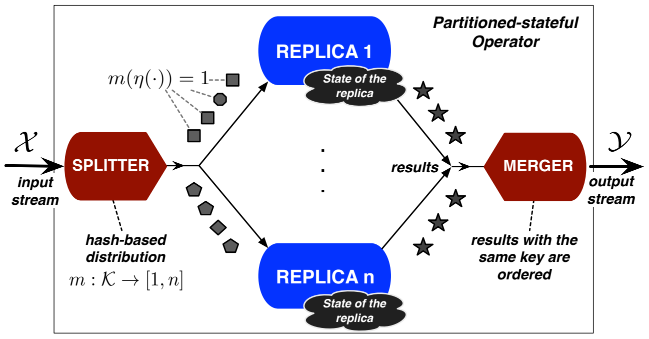


Figure 1: Parallel partitioned-stateful operator: all the tuples with the same key are routed to the same replica.

The operator receives a stream of tuples $\mathcal{X} = \{x_1, x_2, \ldots\}$ and produces a stream of results $\mathcal{Y} = \{y_1, y_2, \ldots\}$. For each tuple $x_i \in \mathcal{X}$, let $\eta(x_i) \in \mathcal{K}$ be the value of a partitioning key attribute, where $\mathcal{K}$ is the domain of the keys. 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 the *splitter* and the *merger* functionalities. The first is responsible for routing each input tuple to the corresponding replica using a (hash) *routing function* $m : \mathcal{K} \rightarrow [1, n]$, where n is the number of replicas. The merger collects results from the replicas and transmits them onto the output stream.

All the tuples with the same key are processed sequentially by the same replica in the arrival order. Therefore, no *lock* is needed to protect the state partitions since each partition is accessed exclusively by the same replica. Furthermore, this solution allows the ordering of results within the same group to be preserved (this can be a necessary property depending on the application semantics).

2.2. Motivations for Elastic Scaling

Real-world stream processing applications are characterized by highly variable execution scenarios. The dynamicity can be described in terms of three different factors:

1. *(D1) variability of the stream pressure*: the input rate can exhibit large up/down fluctuations;
2. *(D2) variability of the key distribution*: the frequency of the keys $\{p_k\}_{k \in \mathcal{K}}$ can be time-varying, making load balancing impossible to be achieved statically;
3. *(D3) variable processing time* per input tuple that may change during the application lifetime. This is possible for different reasons like the current system availability (sharing between applications), or for endogenous causes related to the elastic operator, e.g., the processing time may be dependent on the number of tuples maintained in the window to be processed.

Applications must tolerate these variability issues in order to keep the operator QoS optimized according to some user criteria. Our strategies will be designed to optimize two performance aspects: *i) throughput*, i.e. the number of results delivered per time unit; *ii) latency* (or response time), i.e. the time elapsed from the reception of a tuple triggering the operator internal processing logic and the delivering of the corresponding result.

To achieve the needed QoS, one could think to configure the operator in such a way as to sustain the peak load (e.g., the highest expected arrival rate) by using all the available resources at the maximum CPU frequency supported by the hardware. However, this solution may be very expensive both in distributed environments (number of machines turned on) and on single nodes (too high power consumption). The goal of any elastic support is to meet the application-dependent QoS specifications with high probability by keeping the operating cost within an affordable range. To this end, we target strategies able to modify the following configuration parameters of an elastic operator: *i)* the number of cores (replicas) used $n \in [1, \mathcal{N}]$; *ii) the routing function* $m : \mathcal{K} \rightarrow [1, n]$; *iii) the operating frequency* $f \in \mathcal{F}$ of the CPU(s).

2.3. SASO Properties

To evaluate our strategies we will refer to the well-known SASO properties [10, 2]:

- *(P1) stability*: a stable strategy avoids frequent modifications of the current operator configuration;
- *(P2) accuracy*: high accuracy means that the strategy minimizes the number of QoS violations;
- *(P3) settling time*: a strategy with a good settling time is able to reach the desired configuration quickly;
- *(P4) overshoot*: a strategy overshoots when it overestimates the configuration to meet the needed QoS (using more resources than needed).

These properties are strictly related and represent contrasting objectives. *Stability* is important because reconfigurations cause performance overhead (latency peaks, throughput drops) due to the protocol executed to change the operator configuration consistently with the computation semantics (this will be proved experimentally in Sect. 5). Therefore, it is important to choose a configuration that remains stable for a reasonable amount of time in the future. However, a very stable strategy that minimizes QoS violations (*accuracy*) implies a severe *overshooting* to avoid changing the configuration too frequently. Of course, overshooting increases resource/power consumption. Finally, the amplitude of a reconfiguration is a further parameter to control for two reasons: *i)* largest reconfigurations may be more costly; *ii)* if the controller takes large reconfigurations on average, it can find a stable configuration quickly without a large number of small changes (*settling time*) but it is more sensitive to prediction errors and to fluctuations in the workload. Our goal is to design adaptation strategies able to make trade-offs among these properties.

2.4. Assumptions

In this paper we will make a set of assumptions. First, we will suppose that all the replicas are executed on homogeneous cores (*A1*). Furthermore, we target single-node systems composed of several multicore CPUs (*A2*). Therefore, we will address distributed environments in our future work. Finally, we will assume that the underlying architecture is dedicated to the execution of one elastic parallel operator (*A3*). The coexistence of multiple operators or applications sharing the same resources is an interesting aspect that can integrated in our model-based approach. This will be investigated in our future research.

3. The Approach

We key idea of our approach is to apply the MPC method to design scaling strategies. Fig. 2 shows a generic MPC controller and its three main components:

- ***disturbance forecaster***: *disturbances* are exogenous events that cannot be controlled, e.g., the arrival rate and the processing time per input tuple. The controller must be able to measure and estimate their future values through forecasting tools (e.g., time-series analysis [11], Kalman filters [12]). We denote by $\mathbf{d}(\tau)$ the vector of disturbance variables measured during control step τ (a sampling interval) and by $\widetilde{\mathbf{d}}(\tau+1)$ the predicted vector for the next step;

- ***system model***: the MPC controller exploits a system model to compare alternative configurations. The model captures the relationship between *QoS variables* (e.g., service rate, latency, power) denoted by $\mathbf{q}(\tau)$ and the current configuration expressed as a set of *decision variables* $\mathbf{u}(\tau)$;

- ***optimizer***: at each control step the controller solves an optimization problem to obtain the 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 problem is constrained by the system model to predict the QoS, and by constraints on the admissible values of the decision variables.

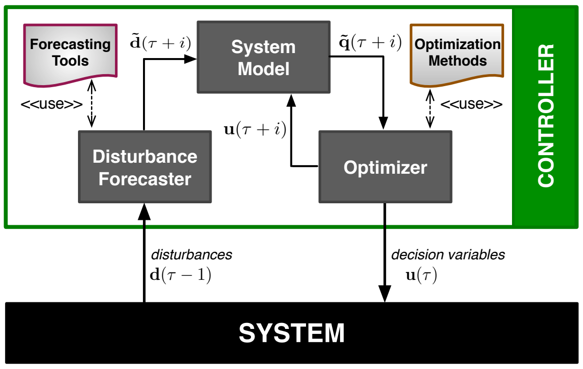


Figure 2: Internal logical structure of a MPC controller: disturbance forecaster, system model and optimizer components.

The fundamental principle of MPC is that only the first element $\mathbf{u}(\tau)$ of the optimal reconfiguration trajectory is used to steer the system to the next control step, and the whole 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 is shifted by one step each time (*receding horizon control*).

In the rest of this section we will describe the application of MPC to our problem of elastic DaSP operators.

3.1. Disturbances

At control step τ the updated measurements related to the last step $\tau-1$ are available and gathered by the controller. Tab. 1 summarizes this set of measurements.

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

Table 1: Basic monitored *disturbance* metrics collected by the splitter and the replica functionalities.

A *triggering tuple* is a tuple that triggers the operator internal processing logic (e.g., a new window must be processed). The replicas monitor the computation times per triggering tuple $\{\mathbf{c}_k\}_{k\in\mathcal{K}}$ and collect these punctual values by transmitting them to the controller at each step. To reduce the amount of measurements, each replica performs a sampling using a *monitoring interval* which is a submultiple of the control step. Times are collected in ticks, thus independently from the current CPU frequency.

The controller needs various additional metrics derived from the basic ones. They are summarized in Tab. 2.

3.2. Performance and Power Models

Tab. 3 shows the decision variables and the QoS variables which are the model inputs/outputs.

Throughput model. This model is aimed at giving an estimate of the average number of triggering tuples that the operator serves per time unit. We are interested in the inverse of the throughput, that is the operator *effective service time* denoted by $T_{\mathcal{S}}$, i.e. the average time interval between the generation of two consecutive results. From the arrays of measurements $\mathbf{c}_k$ for each $k \in \mathcal{K}$, the controller measures the average number of ticks required

Symbol	**Description**
T_k	Mean computation time per triggering tuple with key k.
$T^i_{\mathcal{R}}$	Mean computation time per triggering tuple (of any key) processed by replica i.
T	Mean computation time per triggering tuple (of any key) processed by any replica.
$T^i_{\mathcal{A}}$	Mean inter-arrival time (of any key) to replica i.
$\mathcal{P}_i$	Probability to transmit a triggering tuple (of any key) to the i-th replica.
$T^{id}_{\mathcal{S}}, \sigma_{\mathcal{S}}$	Mean and standard deviation of the ideal service time of the operator.
ρ_i	Utilization factor of the i-th replica.
ρ	Utilization factor of the operator.
$W_{\mathcal{Q}}$	Operator mean waiting time per triggering tuple (of any key).
c_a, c_s	Coefficients of variation for the operator inter-arrival and ideal service time.

Table 2: Metrics derived by the controller from the basic ones.

Symbol	**Description**
	Decision variables
$n(\tau)$	Number of replicas used.
$m^\tau\!:\!\mathcal{K} \rightarrow [1, n(\tau)]$	Routing function used by the splitter.
$f(\tau)$	The operating frequency (GHz) used by the operator.
	QoS variables
$T_{\mathcal{S}}(\tau)$	Effective service time of the operator. It is the inverse of the throughput.
$\mathcal{R}_{\mathcal{Q}}(\tau)$	Response time (*latency*) of the operator.
$\Phi(\tau)$	Power consumed by the operator (in watts).

Table 3: *Decision variables* selected by the controller and the *QoS variables* output of the models.

to process a triggering tuple with key k. We denote it by $\mathcal{C}_k(\tau-1)$. The mean computation time per triggering tuple of key $k \in \mathcal{K}$ can be derived as follows:

$$\widetilde{T}_k(\tau) = \frac{\widetilde{\mathcal{C}}_k(\tau)}{f(\tau)} \tag{1}$$

This expression assumes that the computation times are proportional to the CPU frequency, which is true for CPU-bound computations [13]. For memory-bound computations, in which the CPI (clock cycles per instruction) improves with lower frequencies, this estimation may become less accurate. In the following we will assume Expr. 1 a valid estimation by assessing its accuracy in Sect. 5.

The mean computation time per triggering tuple (of any key) processed by the i-th replica is:

$$\widetilde{T}^i_{\mathcal{R}}(\tau) = \widetilde{\mathcal{P}}_i(\tau)^{-1} \cdot \sum_{k|m^\tau(k)=i} \widetilde{p}_k(\tau) \cdot \widetilde{T}_k(\tau) \tag{2}$$

where $\widetilde{\mathcal{P}}_i(\tau) = \sum_{k|m^\tau(k)=i} \widetilde{p}_k(\tau)$. The formula is a weighted mean of the computation times of the keys routed to that replica.

In Expr. 1 and 2 we use the values of the key frequencies $\{p_k\}_{k\in\mathcal{K}}$ and the computation times $\{\mathcal{C}_k\}_{k\in\mathcal{K}}$ for the next step. Since their current values cannot be measured until the next control step, they need to be forecasted:

- by using the result of predictive history-based filters;
- in some cases it can be sufficient to use the last measured values as the next predicted ones, i.e. $\widetilde{p}_k(\tau) = p_k(\tau-1)$ and $\widetilde{\mathcal{C}}_k(\tau) = \mathcal{C}_k(\tau-1)$ for any $k \in \mathcal{K}$.

The mean inter-arrival time of triggering tuples to the i-th replica is given by:

$$\widetilde{T}^i_{\mathcal{A}}(\tau) = \frac{\widetilde{T}_{\mathcal{A}}(\tau)}{\sum_{k|m^\tau(k)=i} \widetilde{p}_k(\tau)}$$

This follows from the observation that the i-th replica receives only a fraction of the input tuples transmitted by the splitter, i.e. the ones whose key attribute is mapped onto that replica by the current routing function. In the above formula the inter-arrival time value for the current step $\widetilde{T}_{\mathcal{A}}(\tau)$ must be predicted using forecasting tools in order to track the future workload.

The *utilization factor* of the i-th replica is:

$$\widetilde{\rho}_i(\tau) = \frac{\widetilde{T}^i_{\mathcal{R}}(\tau)}{\widetilde{T}^i_{\mathcal{A}}(\tau)}$$

If it is greater than one, that replica is a bottleneck. If no replica is a bottleneck, the splitter is able to route tuples to the replicas without blocking on average. Otherwise, if a replica is a bottleneck, its input queue grows up to reaching its maximum capacity. At this point the back-pressure throttles the splitter which is periodically blocked from sending new tuples to the replicas. The *ideal service* *time* of the operator (the one in isolation) accounts for the slowest replica, the one with the highest utilization factor:

$$\widetilde{T}_{\mathcal{S}}^{id}(\tau) = \widetilde{T}_{\mathcal{R}}^{b}(\tau) \cdot \sum_{k|m^{\tau}(k)=b} \widetilde{p}_k(\tau) \tag{3}$$

$$\textit{Such that:}$$
$$b \in \operatorname*{arg\,max}_{i\in[1,\ldots,n(\tau)]} \widetilde{\rho}_i(\tau)$$

and the effective service time is given by the maximum between the ideal one and the inter-arrival time:

$$\widetilde{T}_{\mathcal{S}}(\tau) = \max\left\{\widetilde{T}_{\mathcal{A}}(\tau),\, \widetilde{T}_{\mathcal{S}}^{id}(\tau)\right\} \tag{4}$$

We observe that to optimize the throughput it is not strictly necessary to balance the workload, but it is sufficient that all the replicas have utilization factor less than one. Under the assumption that the load is evenly balanced among the replicas, the ideal service time formula can be simplified as follows:

$$\widetilde{T}_{\mathcal{S}}^{id}(\tau) = \frac{\sum_{k\in\mathcal{K}} \widetilde{p}_k(\tau)\widetilde{T}_k(\tau)}{n(\tau)} = \frac{\widetilde{T}(\tau)}{n(\tau)} \tag{5}$$

Latency model. Inspired by the approach studied in Ref. [14], we model the latency (response time) by using 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:

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

$W_{\mathcal{Q}}$ is the *mean waiting time* before starting a service, and T is the mean computation time per triggering tuple.

Our idea is to model the operator as a $G/G/1$ queueing system with inter-arrival times and service times having general statistical distributions. An approximation of the mean waiting time is given by Kingman's formula [15]:

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

where $\widetilde{\rho}(\tau) = \widetilde{T}_{\mathcal{S}}^{id}(\tau)/\widetilde{T}_{\mathcal{A}}(\tau)$ and the coefficients of variation are $c_a = \sigma_{\mathcal{A}}/T_{\mathcal{A}}$ and $c_s = \sigma_{\mathcal{S}}/T_{\mathcal{S}}^{id}$.

We choose this model for its generality, since it does not need restrict assumptions on the type of the arrival and service stochastic processes. Simpler formulas of the waiting time for other queueing systems like $M/M/1$, $M/D/1$ and $M/G/1$ exist and can be used in our strategies by assuming that the transmission rate of the stream source can be modeled as a Poisson process. However, the case of Expr. 7 is more general and challenging because several information (e.g., coefficients of variation) must be efficiently monitored by the runtime to apply it. In this paper we will use Expr. 7 by making the following simplifications:

- we model the operator has a single queueing system with ideal service time equal to the one of a replica divided by the number of replicas used $n(\tau)$ (as stated in Expr. 5). This roughly approximates the service time of the operator provided that we are always able to evenly balance the load among the replicas;
- although the mean inter-arrival time for the next step $\widetilde{T}_{\mathcal{A}}(\tau)$ is forecasted using statistical filters, the estimated 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 ideal service time is equal to $\widetilde{c}_s(\tau) = c_s(\tau-1)$. So doing, we suppose that c_s is unaffected by changes in the parallelism degree.

To increase the model precision we use a feedback mechanism in order to fit Kingman's approximation to the last measurements. This mechanism is defined as follows:

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

The parameter r is a correction factor defined as the ratio between the measured 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.

Power model. We are not interested in determining the exact amount of power consumed. It is sufficient a proportional estimation such that we can compare different operator configurations. We focus on the dynamic power dissipation originated from the activities in the CPU, which follows the underlying formula [16, 13, 17]:

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

where the power 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. In the model C_{eff} represents the *effective capacitance* [18], a technological constant that depends on the hardware characteristics of the CPU.

3.3. Optimization Process

The decision variables take their values from a set of finite options. To simplify our problem, we take out the routing function from the decision variables. Consequently, we assume that at each step the controller is able to find a routing function that balances the load. In this way the controller uses Expr. 5 to estimate the ideal service time instead of the more general Expr. 3. This assumption is realistic in cases where the ratio between the most frequent and the least frequent key is acceptably small.

The MPC controller is designed with a *rebalancer* component in charge of computing a new routing function that balances the workload among the replicas. This component is triggered in two cases: *i)* each time it detects that the load difference between the most and the least loaded replicas is higher than a threshold; *ii)* any time the MPC controller changes the number of replicas. The internal structure of the MPC controller is sketched in Fig. 3.

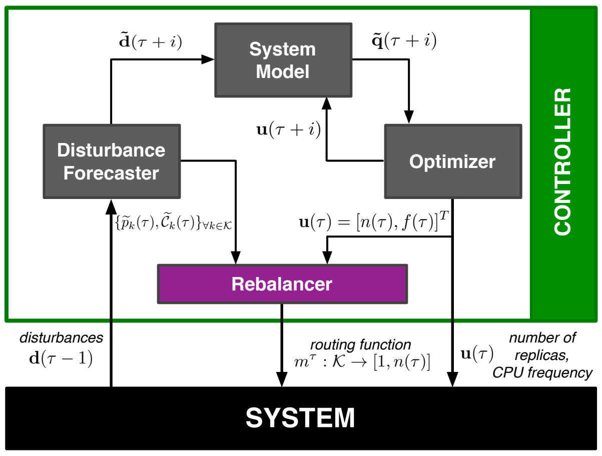

$\tilde{\mathbf{d}}(\tau+i)$
System Model
$\tilde{\mathbf{q}}(\tau+i)$
$\mathbf{u}(\tau+i)$
Disturbance Forecaster
Optimizer
CONTROLLER
$\{\widetilde{p}_k(\tau), \widetilde{\mathcal{C}}_k(\tau)\}_{\forall k \in \mathcal{K}}$
$\mathbf{u}(\tau) = [n(\tau), f(\tau)]^T$
Rebalancer
disturbances $\mathbf{d}(\tau-1)$
routing function $m^\tau : \mathcal{K} \to [1, n(\tau)]$
$\mathbf{u}(\tau)$
number of replicas, CPU frequency
SYSTEM

Figure 3: Internal logical structure of the MPC controller with the *rebalancer* component for generating the routing function.

3.3.1. Problem statement

The MPC controller solves at each step a minimization problem whose objective function $\mathcal{J}$ is expressed as the sum of a cost L over all the steps of the prediction horizon, i.e. $\min \mathcal{J}(\tau) = \sum_{i=0}^{h-1} L(\mathbf{q}(\tau+i), \mathbf{u}(\tau+i))$. A general form for L is the following:

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

In the following we will describe in detail the three components of the objective function.

QoS cost. In this work we use two QoS cost definitions according to the target QoS policy of the system:

- *throughput-based cost*: the goal in this formulation is to make the operator able to sustain the input rate. We model this objective as follows:

$$Q_{cost}\big(\widetilde{\mathbf{q}}(\tau+i)\big) = \alpha\, \widetilde{T}_{\mathcal{S}}(\tau+i) \tag{11}$$

  That is a linear cost proportional to the effective service time, where $\alpha > 0$ is a unitary price per time instant. To minimize the QoS cost the controller chooses one of the configurations with minimum service time (i.e. the predicted inter-arrival time);

- *latency-based cost*: in latency-sensitive applications the response time needs to be bounded to some maximum thresholds [19]. We model this requirement with a cost function defined as follows:

$$Q_{cost}\big(\widetilde{\mathbf{q}}(\tau+i)\big) = \alpha \, \exp\left(\frac{\widetilde{R}_{\mathcal{Q}}(\tau+i)}{\delta}\right) \tag{12}$$

  where $\alpha > 0$ is a positive cost factor. The cost lies in the interval $(\alpha, e\,\alpha]$ for latency values within the interval $(0, \delta]$, where $\delta > 0$ is the desired threshold. Such kind of cost heavily penalizes configurations with a latency greater than the threshold.

Resource cost. The resource cost is defined as a cost proportional to the number of used replicas or to the overall power consumed. This corresponds to the two following cost definitions:

$$\begin{aligned} R_{cost}\big(\mathbf{u}(\tau+i)\big) &= \beta\, n(\tau+i) && \textit{per-core cost} && (13) \\ &= \beta\, \widetilde{\Phi}(\tau+i) && \textit{power cost} && (14) \end{aligned}$$

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

Switching cost. We use the following switching cost definition to represent an abstract cost for changing the current configuration:

$$S^{w}_{cost}\big(\mathbf{\Delta_u}(\tau+i)\big) = \gamma \left(\|\mathbf{\Delta_u}(\tau+i)\|_2\right)^2 \tag{15}$$

where $\gamma > 0$ is a unitary price factor, and $\mathbf{\Delta_u}(\tau)$ is the difference between the decision vectors used at two consecutive control steps. Quadratic switching cost functions are common in the control theory literature [7]. Any norm can be used in this definition. In this paper we use the Euclidean norm.

This switching cost term has two important effects on reconfigurations:

- it is proportional to the reconfiguration amplitude, hence it penalizes larger reconfigurations. Large reconfigurations likely require migrating more keys to keep the load balanced, and this may be source of further QoS violations (this will be studied in Sect. 5.2.2);

- it allows the controller to be more conservative in releasing/acquiring resources. The effect is that the use of the switching cost allows smoothing the reconfiguration sequence in case of frequent fluctuations of the arrival rate, where resources are continuously acquired and released in control steps close in time.

A final consideration should be made on the nature of the parameters α, β and γ. They are defined as the ratio between a dimensionless weight (*priority*) and a *scale factor*, used to normalize the values of the cost terms using reasonable estimations of their maximum bounds obtained through preliminary experiments without controller.

3.3.2. Search space reduction

At the worst case the optimization phase needs to explore the combinatorial set of all the feasible combinations of the decision variables. The number of possible trajectories is $\mathcal{O}(\Omega^h)$ where $\Omega = \mathcal{N} \times |\mathcal{F}|$, thus the optimization has an exponential increase in worst-case complexity with an increasing number of reconfiguration options and longer prediction horizons. For this reason, the computational overhead of the controller could become a major concern, as the strategy needs to occupy a small (negligible) fraction of the control step interval. Therefore, methods to reduce the computational overhead are mandatory for the real-time execution of our approach.

The MPC optimization is essentially a search problem over a tree structure called *evolution tree* [20], whose height corresponds to the horizon length h and the arity to the total number of configuration options Ω, as shown in Fig. 4. An approach to reduce the explored search space consists in Branch & Bound methods (B&B). In this paper we will use the following procedure:

- we assign to each explored node i of the tree a variable C_i which represents the cost spent to reach that node from the root. The cost of the root is zero;
- for each node we have Ω possible branches. The subtree rooted at node i is explored if and only if $C_i < C_{opt}$, where C_{opt} is the minimum cost of all the root-to-leaf paths currently explored during the search process. If i is a leaf, we further set $C_{opt} = C_i$.

This procedure can be applied if *the cost function $\mathcal{J}$ is monotonically increasing with the step of the horizon.* According to Expr. 10, $L > 0$ for each step, thus the cost function $\mathcal{J}$ satisfies this property.

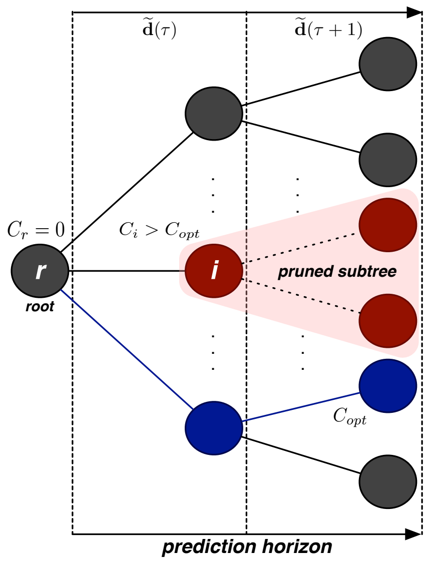


Figure 4: Evolution tree ($h = 2$) and B&B procedure to reduce the search space explored by the MPC optimization phase.

The procedure does not affect the final choice, which is the same of the exhaustive search. However, the space reduction depends on the size of the pruned subtrees and we have no guarantee of its efficacy in general.

In conclusion, B&B techniques mitigate the problem but do not solve it at all. For large configuration spaces the overhead could be unacceptable even using B&B. Possible solutions can be evolutionary algorithms, AI methods or heuristics [21]. The employment of such techniques is one of our future research directions, especially when we will move our methodology to distributed environments.

4. Runtime System

We have implemented our elastic strategies in the `FastFlow`[1] framework [22] for stream processing applications on multicores. In this section we briefly present our runtime environment and how to perform reconfigurations efficiently on shared-memory architectures.

In the implementation the splitter, merger and the replicas are executed by threads with fixed affinity on the cores of the underlying architecture. According to the FastFlow cooperation model [22], threads interact by exchanging memory pointers to shared data through `pop`() and `push`() primitives on lock-free queues. The splitter is interfaced with the input stream through a TCP/IP POSIX socket. The controller is executed by a dedicated control thread.

4.1. Reconfiguration Mechanisms

In the case of an increase in the number of replicas, the controller instantiates the new replica threads, and creates the FastFlow queues used to interconnect them with the splitter and merger and the controller itself. The controller sends special *control messages* through dedicated queues to the splitter and the merger in order to notify them of the new replicas. Then, the splitter starts the migration protocol described in the sequel. Symmetric actions are taken in the case of a removal of a subset of replicas.

In case of an increase or a decrease in the number of replicas, the controller computes a new routing function which is notified to the splitter. As studied in our past work [8], on shared-memory machines the best solution is to keep the workload as balanced as possible among the replicas. Accordingly, many keys can be moved during a reconfiguration. On distributed-memory architectures like clusters, where state migration may need to transfer the state between machines, different heuristics [23, 3, 2] can be used to accept a slight load unbalance among replicas by moving only a small subset of the keys.

Finally, variations in the current CPU frequency do not affect the structure of the parallel implementation and can be performed transparently. They are performed by the controller using the `C++ MAMMUT` library[2] (MAchine Micro Management UTilities), which allows the controller to change the frequency of the cores by writing on proper `sysfs` files of the Linux OS.

4.2. State Migration

The state migration protocol is critical. We identify four fundamental properties of a reconfiguration protocol,

[1]`http://mc-fastflow.sourceforge.net`
[2]The Mammut library is open source and freely available at `https://github.com/DanieleDeSensi/Mammut`

which represent general design principles to perform reconfigurations with low latency:

- *(R1) gracefulness*: the splitter and the replicas should avoid discarding input tuples, processing tuples with the same key out-of-order, and should prevent the generation of duplicated inputs/results;
- *(R2) fluidness*: the splitter should never wait for the migration completion before re-starting to distribute new incoming tuples to the replicas;
- *(R3) non-intrusiveness*: the migration should involve only the replicas exchanging parts of their state. The other replicas must be able to process the input tuples without interferences;
- *(R4) fluentness*: while a replica involved in the migration is waiting to acquire the state of an incoming key, it should be able to process all the input tuples with other keys for which the state is ready to be used.

In the next section we will use these properties to qualitatively evaluate the main existing solutions. Then, we will present our approach.

4.2.1. Qualitative comparison

The approach in Ref. [2] processes the tuples in their arrival order without generating duplicate inputs and results *(R1)*. However, the migration activity needs synchronization barriers among the splitter and the replicas. Therefore, this support is neither able to achieve *(R2)*, i.e. the splitter is blocked waiting for the migration completion, nor properties *(R3)* and *(R4)*, because all the replicas are blocked during the whole reconfiguration process. Analogous is the case of the approach described in Ref. [5].

Substantial enhancements have been developed in Ref. [24]. The migration is made less intrusive, as it involves only the replicas that need to exchange state partitions *(R3)*. Furthermore, the replicas that wait for the acquisition of some state partitions can process the tuples whose state is already available *(R4)*, and the splitter does not block until the migration has finished *(R2)*. However, this mechanism adds additional complexity because during the reconfiguration the splitter transmits tuples of the migrated keys to both the old and the new replicas, and the merger filters duplicated results. Therefore, *(R1)* is not achieved. Similarly, in the approach described in Ref. [4] the splitter (or any upstream operator) has to deal with the case of missing tuples not served during the migration, which must be properly re-generated from the last checkpoint. This phase affects the routing of new tuples in the splitter, by adding a delay that depends on the number of items to re-generate *(R2)*. In contrast, in our approach (see next section) the splitter transmits at most one migration message per replica, which represents a negligible delay in the distribution activity.

Ref. [25] targets streaming applications in the MapReduce framework. They have developed an asynchronous checkpointing technique to efficiently migrate the state partitions. As in other solutions, this approach allows the state transfer to be executed asynchronously while the replicas are working. A new replica acquires the updated state partitions of the incoming keys using the last checkpoint, and the splitter temporarily distributes duplicated tuples to both the old and the new replica (as in Ref. [4]) until the state has been acquired successfully. Since duplicated tuples are generated during the reconfiguration phase, property *(R1)* is not met. The same drawback characterizes the solution described in Ref. [3].

Finally, the technique described in Ref. [23] is the most similar to our approach. Besides the achievement of properties *(R3)* and *(R4)* common to many other solutions, this approach gives a central role to the splitter, which is in charge of buffering all the tuples of the moved keys in a buffer. Such tuples will be delivered to the replicas as soon as the migration is complete by quiesces the routing of new fresh input tuples (not involved in the migration) to the other replicas in the meanwhile. Therefore, although no duplicated tuple/result is generated in this solution *(R1)*, property *(R2)* is not met. Tab. 4 summarizes the qualitative comparison among existing solutions. In the next section we will present our migration protocol targeting very low-latency applications.

Work	R1	R2	R3	R4
[2, 5]	Yes	No	No	No
[24, 3, 25]	No	Yes	Yes	Yes
[4]	No	No	Yes	Yes
[23]	Yes	No	Yes	Yes
Our	Yes	Yes	Yes	Yes

Table 4: Qualitative comparison between migration protocols.

4.2.2. Migration protocol

After a reconfiguration decision, the controller transmits a *reconfiguration message* to the splitter containing the new routing table m^τ. The splitter always receives data non-deterministically from the input stream (new tuples) and from the controller (reconfiguration messages), the latter with higher priority. Once received a reconfiguration message, the splitter recognizes the keys that must be migrated and transmits to the involved replicas a sequence of *migration messages*:

- `move_out(k)` is sent to the replica r_i that held the state of key $k \in \mathcal{K}$ before the reconfiguration but will not hold them after it, 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. $m^{\tau-1}(k) \neq j \wedge m^\tau(k) = j$.

All the keys $k \in \mathcal{K}$ s.t. $m^{\tau-1}(k) = i \wedge m^\tau(k) = i$ are not involved in the migration and will be processed by the replicas without interferences *(R3)*. To reduce the number of migration messages, all the messages directed to the same replica are packed into one single message with all the keys that must be moved in/out to/from the destination replica. It is worth noting that the splitter does not need to wait for the completion of the migration activities by the replicas and can always route new tuples to them *(R2)*.

The splitter starts immediately to route the new input tuples with the routing function m^{τ} *(R1)*. Let us suppose that, as a result of the reconfiguration chosen by the controller ①, a key $k \in \mathcal{K}$ must be migrated from replica r_i to r_j as depicted in Fig. 5. 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. In fact, the replica receives the `move_out(k)` message through a `pop`() from the same FIFO queue used for retrieving tuples distributed by the splitter. Therefore, r_i can safely save the state of that key (denoted by s_k) to a *backing store* (see Fig. 5) used to collect the migrated state partitions and to synchronize the pairs of replicas involved in the migration.

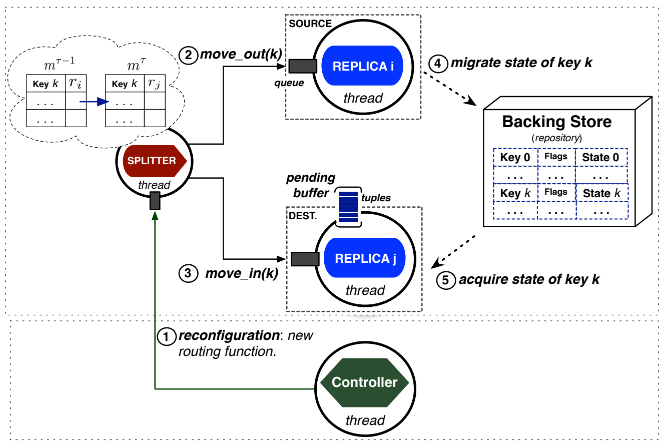


Figure 5: Example of state migration between replica r_i and replica r_j for key k.

The replica r_j, which receives the `move_in(k)` message ③, may receive new incoming tuples for that key before the state is ready. Only when the replica r_i has properly saved the state s_k to the repository ④, it can be acquired by r_j ⑤. During this phase replica r_j is not blocked but accepts and processes new tuples *(R4)*. All the tuples with key k are enqueued in a *pending buffer* private of that replica until the state s_k is available. The availability of s_k in the backing store is periodically checked at each reception of a new tuple. When the state becomes available, it is acquired by r_j 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.

The main enhancement with respect to Ref. [23] is that the buffer of the moving keys received during the reconfiguration is not centralized in the splitter, causing a delay in its activity after the end of the migration, but it is decentralized in the replicas themselves (only the ones involved in the migration). Therefore, the splitter is always able to route tuples to the replicas without interruptions.

When replicas are executed on multiple distributed nodes, the state migration requires copying the data structures to/from the repository. Instead, in our case the backing store consists in a shared memory area in which replicas exchange memory references to the data structures. This avoids the copy overhead, and the synchronization to check the state availability can be efficiently implemented using spin-loops on boolean flags.

5. Experiments

In this section we evaluate our control strategies on a kernel of a data stream processing application operating in the high-frequency trading domain (HFT).

The code is compiled with `gcc 4.8.1` and the `−O3` optimization flag. The architecture is a dual-CPU Intel Sandy Bridge with 16 hyperthreaded 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.

5.1. Application

HFT computations ingest huge volume of data at a great velocity with strict QoS requirements. The application kernel is inspired by the work in Ref. [26], see Fig. 6.

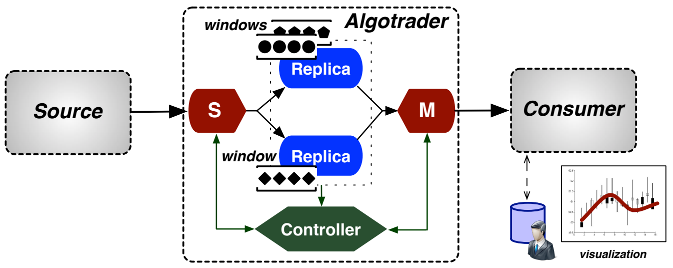


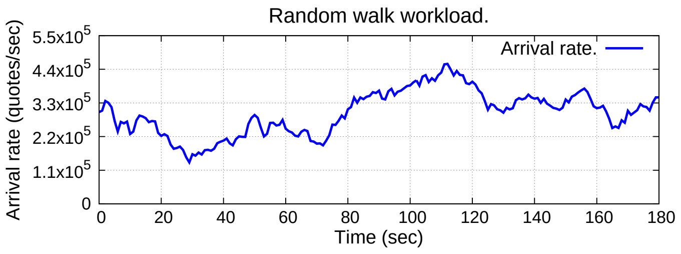
Figure 6: Kernel of a high-frequency trading application.

The *source* operator delivers a stream of *quotes* (buy and sell proposals) represented by a record of attributes, i.e. the proposed price, volume and the stock symbol (64 bytes in total). The *algotrader* operator processes the quotes grouped by the stock symbol. A count-based window of size $|\mathcal{W}| = 1,000$ is maintained for each group. After receiving $\delta = 25$ new tuples of the same symbol, the computation processes the buffered quotes. The logic estimates the future price by performing two phases:

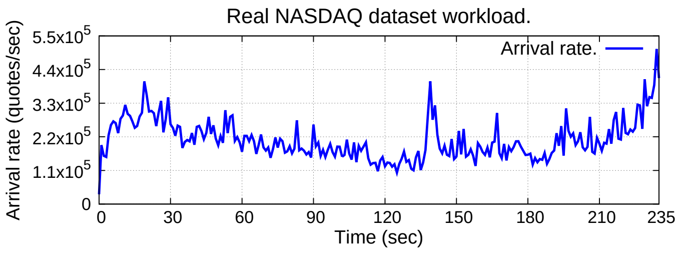
- *aggregation*: the quotes with a timestamp within the same *resolution interval* (1 ms) are transformed into a single tuple by averaging the values of the attributes;
- *regression*: the quotes (one per resolution interval) are used as input of the Levenberg-Marquardt regression algorithm that produces a polynomial fitting the aggregated quotes. We use the C++ library `lmfit` [27].

The *consumer* operator processes the results by producing a representation in the form of candlestick charts. The algotrader and the consumer are executed on the same machine while the source is allocated on a different host.

We use a *synthetic* and a *real dataset* workload (see Fig. 7). In the last we use a trading day (30 Oct 2014) of the NASDAQ market[3] with 2, 836 traded stock symbols. The peak rate is near to 60, 000 quotes per second. To stress the computation, we accelerate it by 100 times. In the synthetic workload the arrival rate follows a random walk model, and the key frequency distribution is fixed and equal to a random time-instant of the NASDAQ dataset. The execution of the synthetic workload and the real dataset lasts 180 and 235 seconds respectively, the latter equal to about 6 hours and half of a whole trading day.



(a) Random walk.



(b) Real dataset.

Figure 7: Arrival rate: synthetic workload and the real throttled (100×) dataset of a NASDAQ trading day.

5.2. Evaluation of the Mechanisms

In the first set of experiments we analyze: *i)* the overhead of our elastic support; *ii)* a comparison of different migration protocols; *iii)* computational complexity aspects of our control strategies.

5.2.1. Overhead of the elastic support

In this experiment we measure the maximum input rate that the algotrader withstands with the highest CPU frequency. The input tuples are generated with a fast constant rate while the 2, 836 keys are uniformly distributed. Fig. 8 compares the elastic implementation, with the controller functionality and all the monitoring activities performed with a sampling of 1 second, and the implementation without the elastic support.

[3]The dataset can be downloaded at the website: `http://www.nyxdata.com`

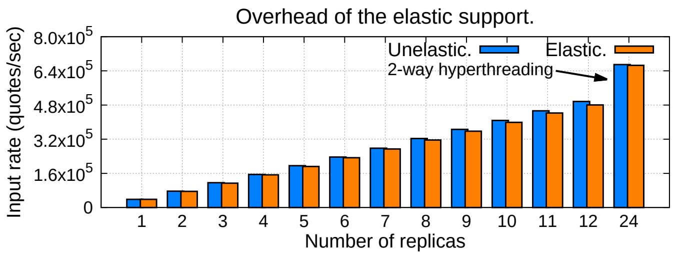


Figure 8: Comparison between the elastic and the unelastic implementations.

The overhead is of $3 - 4\%$ on average. The maximum number of replicas without hyperthreading is 12, as four cores are dedicated to the splitter, merger, controller and consumer threads. We also report the case of 24 replicas in which we map two replicas per core by exploiting hyperthreading. The *scalability*, i.e. the ratio between the rate with 12 replica and with one replica, is 12.91 and 12.56 for the unelastic and the elastic case. This slight hyperscalability is due to better temporal locality. In fact, owing to key partitioning, with more replicas it is more likely to find the needed window structure in one of the private caches of a core (temporal locality). Furthermore, we observe that with hyperthreading the operator sustains higher rates. However, we do not use this feature in the evaluation because its gain is not always easily predictable. We will investigate the use and modeling of hyperthreading in our strategies in our future work.

As a further experiment we study the maximum rate by increasing the number of keys, see Fig. 9. In addition to the previous case, we configure four experiments with 5K, 10K, 25K and 100K uniformly distributed keys. We show for each case the maximum rate with one replica and with 12 replicas, and we report over each bar the value of the scalability. As we can observe, the maximum rate decreases with more keys, although the scalability remains roughly similar. The main reason is that the higher the number of keys the greater the processing time per tuple, because less temporal locality can be exploited by the replicas and more cache misses are on average generated during the processing on each tuple.

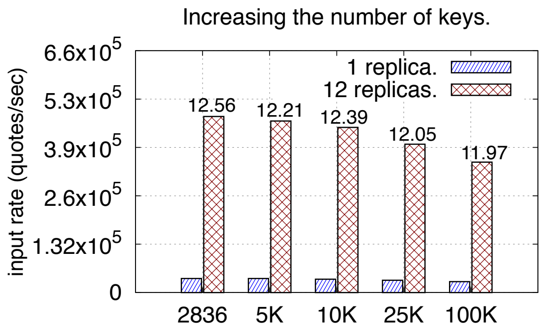


Figure 9: Maximum input rate with different numbers of keys.

5.2.2. *Analysis of migration protocols*

In this section we compare our migration protocol (denoted by MP) with two alternatives that reproduce some of the features of the protocols described in Sect. 4.2.1:

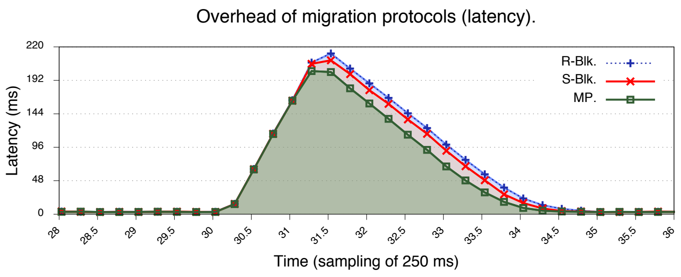
- a protocol (R-BLK) that achieves all the properties except *(R4)*. The protocol blocks the replicas involved in the migration, which are not able to process the input tuples until the state of all the incoming keys has been acquired;
- a protocol (S-BLK) that achieves all the properties except *(R2)*. The protocol uses a centralized pending buffer for the tuples belonging to the moving keys, which are buffered by the splitter. Using a centralized buffer implies that the splitter is involved in dequeuing all the tuples from the buffer after the state has been transferred. This activity delays the scheduling of new input tuples because the splitter does not receive new tuples from the input stream during the dequeuing phase. This approach is inspired to the work presented in Ref. [23].

Other more intrusive protocols produce worse results and are not considered for the sake of brevity.

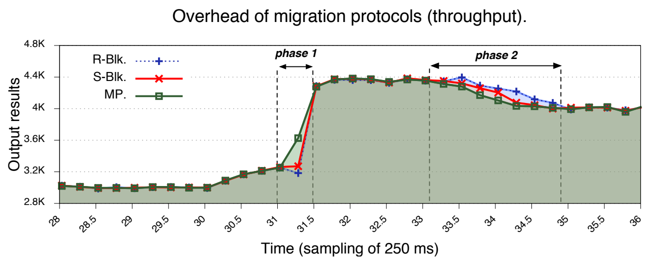
We study a scenario in which the operator is not a bottleneck and its workload is balanced until timestamp 30. Then, we force the input rate to abruptly change from 3×10^5 tuples/sec to 4×10^5 tuples/sec. The strategy triggers a reconfiguration at timestamp 31 by changing the number of replicas from 6 to 8 with the same CPU frequency. To perfectly balance the workload, the reconfiguration moves about 80% of the keys. A low-latency migration protocol has two goals: the tuples belonging to non-migrated keys that are received during the migration phase must be processed with low latency; furthermore, after the migration completion the protocol must be able to reach the new steady state quickly.

Fig. 10a shows the latency peaks. During the migration MP is always able to process tuples not belonging to the migrated keys. The R-BLK protocol produces higher latency peaks because almost all the replicas are blocked during the migration. The S-BLK achieves lower latency peaks than R-BLK. However, after the migration it delays the splitter from the routing of new tuples of any key, and the latency decreases more slowly than in MP.

Fig. 10b reports the number of results produced per monitoring interval. In the first phase the protocol moves the state partitions of the migrated keys. During this time period our protocol produces more results than the other two solutions. In the second phase all the tuples buffered during the migration are finally computed. In R-BLK this set of tuples comprises also tuples of non-migrated keys, which are computed later and with greater latency than in MP. S-BLK represents an intermediate solution that still delays the splitter when all the buffered tuples can be processed. In MP these tuples have been already distributed



(a) Latency peaks observed during the migration.



(b) Throughput measured during the migration.

Figure 10: Impact of the migration protocol: latency and number of results of the algotrader operator.

and buffered by the corresponding replicas, and they are computed in advance with lower latency.

This set of experiments shows that reconfigurations do not come from free, but they produce latency peaks and throughput drops due to the complex protocol needed to handle the reconfiguration consistently.

5.2.3. *Dealing with control complexity*

The MPC controller may explore a potentially huge number of states that grows exponentially with the horizon length. This can rapidly make the computational burden excessive for the real-time adaptation. Tab. 5 shows the theoretical number of states that the control strategy should explore with an exhaustive research and the ones explored with the B&B solution described in Sect. 3.3.2.

	States	**Explored**	**%**	**Time**
$h = 1$	108	108	100	45.77 usec
$h = 2$	11,664	2,537	21.75	608 usec
$h = 3$	1,259,712	72,055	5.72	17 msec

Table 5: Number of explored states by the MPC controller.

Ideally, the execution of the MPC procedure should cover a small fraction of the control step. According to the results of the table, the B&B solution is capable of reducing the number of explored states. The reduction is of several orders of magnitude and makes it possible to complete the online optimization in few milliseconds with the longest length of the horizon used in this paper. However, we can note that the number of explored states still grows rapidly. Therefore, other solutions must be adopted in cases with more reconfiguration options and longer horizons (e.g., evolutionary algorithms).

5.3. Evaluation of Elastic Strategies

We study different MPC strategies listed in Tab. 6. Each strategy is evaluated without switching cost ($\gamma = 0$) or with the switching cost term and different lengths $h \geq 1$ of the horizon. Horizons longer than one step are meaningful only with the switching cost enabled. Therefore, $h = 1$ is implicit in any NoSw strategy. The `*-Node` strategies change the number of replicas using the maximum CPU frequency, while the `*-Power` strategies change also the CPU frequency.

QoS cost	Resource cost	Name	alpha
throughput-based (Expr. 11)	per node (Expr. 13)	`Th-Node`	rw: 200 real: 200
throughput-based (Expr. 11)	power cost (Expr. 14)	`Th-Power`	rw: 200 real: 300
latency-based (Expr. 12)	per node (Expr. 13)	`Lat-Node`	rw: 2 real: 3
latency-based (Expr. 12)	power cost (Expr. 14)	`Lat-Power`	rw: 2 real: 4

Table 6: MPC strategies studied in the experiments.

The parameters α, β and γ require a careful tuning in order to give the desired weight to the components of the cost functions and to normalize properly their values. We use $\beta = 0.5$ and $\gamma = 0.4$ for all the strategies, while we need a different value of α for each workload scenario (see Tab. 6). This choice of the parameters gives more priority to the QoS cost, while the resource cost and the switching cost have a lower priority. We model in this way an important case in which the controller tries to reduce QoS violations by using minimal resources.

All the strategies perform statistical predictions of measured disturbances. We assume that:

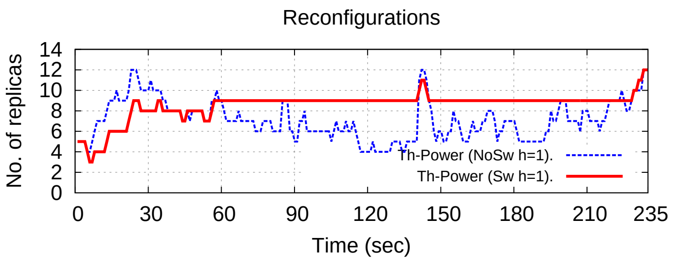
- the arrival rate is predicted according to a Holt Winters filter [28] able to capture trend and cyclic non-stationarities of the time-series;
- the frequencies and the computation times per key are estimated using the last measured values.

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

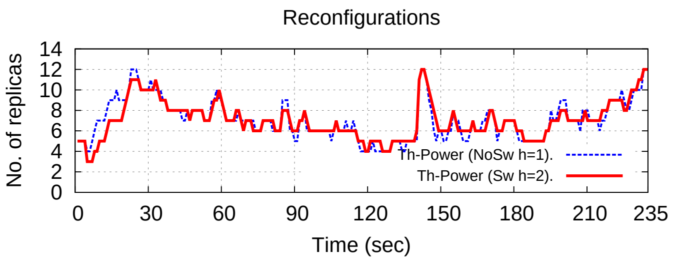
5.3.1. Reconfigurations

We study the effect of the switching cost on the number of reconfigurations. Fig. 11 shows the replicas used by the `Th-Power` strategy with the real dataset workload.

As we can observe, *the switching cost acts as a brake that slows the acquisition/release of resources*. In particular:



(a) `Th-Power` NoSw ($h = 1$) vs. Sw ($h = 1$).



(b) `Th-Power` NoSw ($h = 1$) vs. Sw ($h = 2$).

Figure 11: Number of used replicas per control step (1 sec). `Th-Power` strategy with and without the switching cost.

- during phases in which the arrival rate is expected to increase (*increasing trends*), the strategy without switching cost acquires resources at each step. With switching cost resources are acquired more slowly;
- the opposite behavior characterizes *decreasing trends* of the arrival rate, where with the switching cost the resources are released more slowly by the controller.

This is evident in Fig. 11a. The effect of the brake is very intensive with a short horizon of one step, and as a result from step 60 to 220 the controller uses more resources that needed, because it has no convenience to release them.

The horizon length is used to *mitigate the effect of the brake*. With a longer horizon of $h = 2$ (Fig. 11b) the controller *i)* anticipates the acquisition of resources that will be needed in future steps, and *ii)* anticipates the release of resources that will be no longer needed in the future steps. Qualitatively the reconfiguration sequence with a longer horizon approximates the sequence of choices taken without the switching cost (blue dashed line), however by smoothing several reconfigurations that can be avoided.

Fig. **??** summarizes the results with the other strategies. More reconfigurations are performed in the real workload, due to a higher variability of the arrival rate. Furthermore, more reconfigurations are performed with the strategies `Th-Power` and `Lat-Power` with respect to `Th-Node` and `Lat-Node`, because they use more reconfiguration options (CPU frequency and the number of replicas).

These results allow us to clarify an important property of our strategies, stated as follows:

Result 1. *The switching cost allows the strategy to reduce the number and frequency of reconfigurations (P1). This is partially offset by increasing the horizon length.*

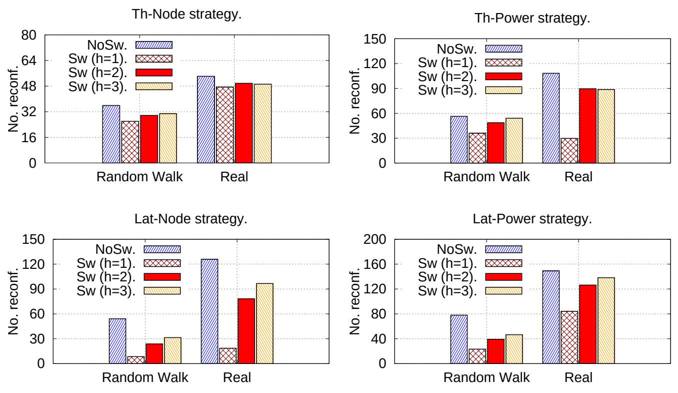


Figure 12: Number of reconfigurations per strategy: random walk and real workload.

5.3.2. *Resource and power consumption*

For `*-Node` strategies we measure the number of replicas used, while for `*-Power` we consider the power consumption in watts collected through the RAPL (Running Average Power Limit) interface [29]. We measure the core power counters. Additional $25 - 30$ watts per second must be added to obtain the per-socket power consumption.

According to assumption *(A1)*, the two chips of our multicore always use the same frequency. Fig. **??** shows the watts consumed by the strategies without switching cost.

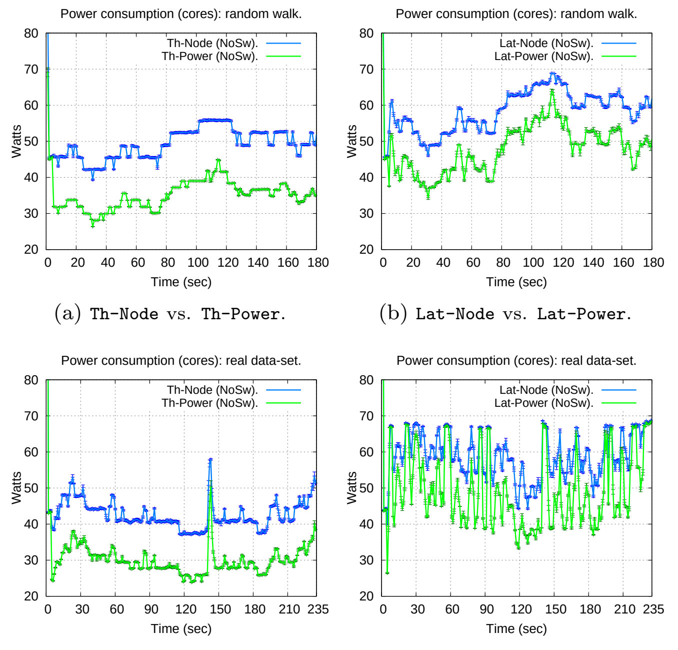


Figure 13: Power consumption (watts) of the non-switching cost strategies: random walk and real dataset workload.

The watts consumed with the `*-Power` strategies are always below the consumption without frequency scaling. The `Th-Power` strategy saves $13 \div 14$ watts per second on average compared with `Th-Node`, while `Lat-Power` saves $10 \div 11$ watts than `Lat-Node`. The effect of the switching cost with different horizon lengths is shown in Fig. **??**. As we can observe, the consumption with short horizons is higher because the used configuration is often oversized. The reason is that the controller is more conservative in releasing already owned resources than in acquiring new resources (this is a side effect of using a higher priority for the QoS cost term in the optimization). By using longer horizons this effect is partially mitigated and lower overshooting can be achieved. In conclusion we can state the following result:

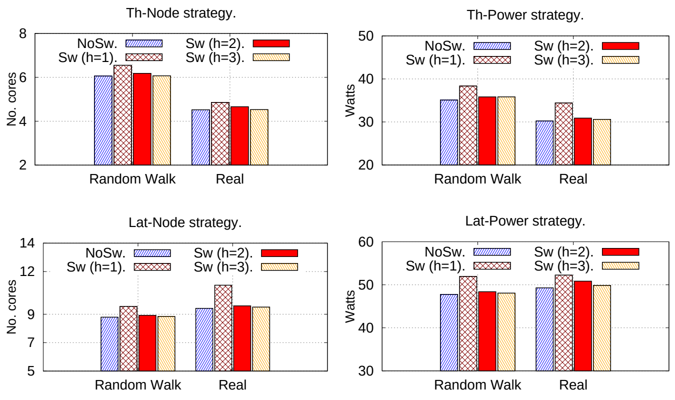


Figure 14: Resources/power consumed per strategy: random walk and real workload.

Result 2. *The switching cost causes overshoot (P4). This can be mitigated by using longer horizon lengths.*

We can see that the `Th-*` strategies are less resource demanding that `Lat-*`. In fact, throughput optimization requires to find the minimum configuration achieving a utilization factor less than one. Instead, the `Lat-*` strategies need to find a configuration that properly minimizes the utilization factor such that the waiting time (Expr. 6) is small enough to meet the latency constraint.

Finally, to assess the importance of the elastic support on multicores we have evaluated the power consumption of the cores with a static *peak-load configuration* in which the operator is configured to use the maximum number of replicas (12) at the maximum frequency (2 Ghz) for all the steps of the execution. The average consumption is of 68.25 watts per steps. The power saving of the elastic support is significant, i.e. 30.7% with the `Lat-Power` strategy (NoSw) and 48.6% with `Th-Power` (NoSw), and obtained with a limited increase in QoS violations ($10 - 20\%$).

5.3.3. *Respecting the QoS constraints*

The strategies have important effects in the accuracy *(P2)* achieved by our elastic support. A QoS violation is a deviation from the desired behavior defined as follows:

- for throughput-based strategies we measure the ratio between the number of results produced per control step N_{out} and the number of triggering tuples N_{in} received. We detect a QoS violation if the ratio is lower than a specified threshold θ_{th};

- for latency-based strategies we detect a QoS violation each time the average latency measured during a control step is higher than a threshold θ_{lat}.

Fig. **??** shows the QoS violations under the random walk workload with the `Th-Power` strategy. We detect a violation when the blue line crosses the red region ($\theta_{th} = 0.95$). The strategy without switching cost adopts the minimal configuration to avoid being a bottleneck at each step. If the arrival rate predictions are underestimated, the operator may likely be a bottleneck and the ratio N_{out}/N_{in} assumes values lower than θ_{th}. The controller reacts by changing the configuration in the future steps, and the enqueued tuples can be processed when more resources are allocated by producing peaks greater than 1 of the ratio. This is the reason for the zig-zag pattern in the figure.

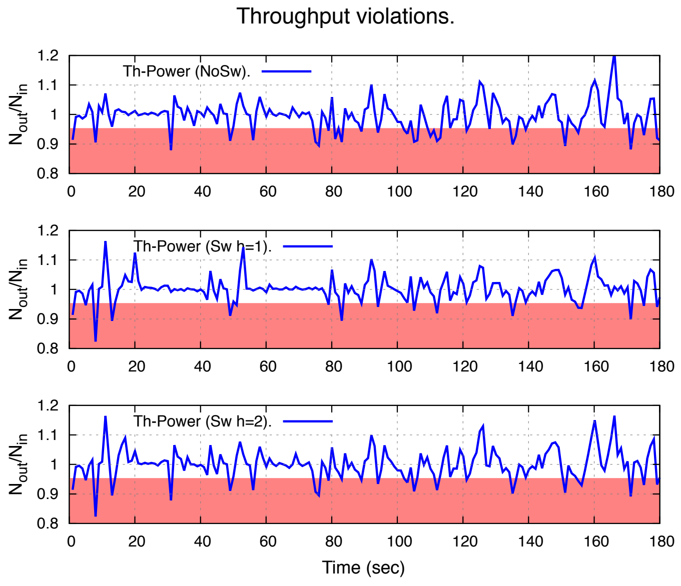


Figure 15: Number of throughput violations: strategy `Th-Power` (NoSw $h = 1$ and Sw $h = 1, 2$) and random walk.

The best accuracy is obtained by the strategy with switching cost and horizon $h = 1$. This is an expected result, as this strategy uses more resources than necessary by tolerating workload underestimations. With longer horizons the accuracy worsens (26 violations with $h = 2$) but resource consumption improves, consequence of a lower overshooting. Therefore, in our approach *longer horizons are mainly useful to reduce resource/power consumption without increasing too much the number of QoS violations* (which are minimum with a very over-provisioning strategy like Sw $h = 1$). Fig. **??** shows a summary of the QoS violations achieved by all the strategies under the synthetic and the real dataset workload scenarios. The behavior can be summarized by the following property:

Result 3. *The switching cost allows the MPC strategy to reach better accuracy (P2). This positive effect is partially offset by increasing the horizon length.*

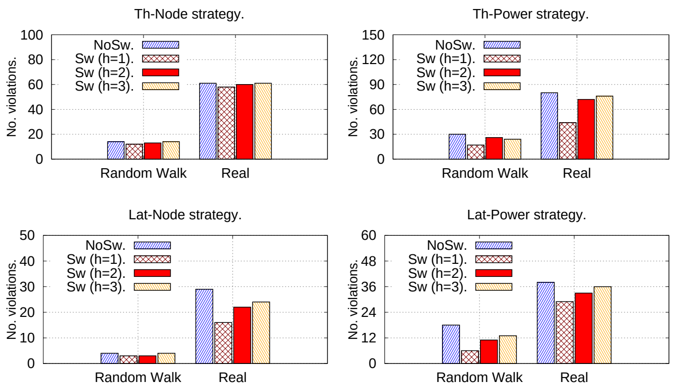


Figure 16: Number of QoS violations per strategy: random walk and real workload.

5.3.4. Settling time

When the workload changes suddenly an effective strategy should be able to reach rapidly the new configuration that meets the QoS requirements. If the strategy takes small reconfigurations (e.g., fews replicas are added/removed each time), this negatively impacts the settling time property. Fig. **??** shows the average *reconfiguration amplitude*. It is the average Euclidean distance between the vector $\mathbf{u}(\tau)$ and the vector $\mathbf{u}(\tau - 1)$ for each τ. The admissible frequency values have been normalized to obtain integers from 1 to 9.

As shown in Fig. **??**, the strategy with switching cost and $h = 1$ performs smaller reconfigurations. The highest amplitude is achieved by the strategy without the switching cost. Therefore, we can conclude that:

Result 4. *The switching cost reduces the average reconfiguration amplitude. Better settling time (P3) can be achieved by using longer prediction horizons.*

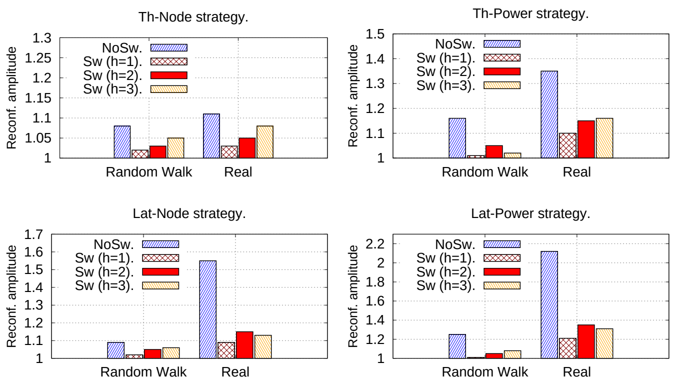


Figure 17: Reconf. amplitude of the strategies.

5.4. Comparison with Similar Approaches

We compare our work with a reactive strategy based on policy rules and the heuristic presented in Ref. [2]. The reactive strategy is based on *event-condition-action* rules widely adopted in Autonomic Computing [30]. The strategy increases/decreases the number of replicas if the operator utilization factor is over/under a maximum/minimum threshold (θ_{max} and θ_{min}).

We have also implemented the algorithm in Ref. [2] developed for the SPL framework [31]. The strategy monitors whether the operator is capable of sustaining the actual stream speed and evaluates a corresponding congestion index. A congestion is detected if the index is over a threshold. The strategy is throughput oriented and monitors the number of input tuples served by the operator. It changes the number of replicas based on the current congestion index and the recent history of past actions. If a past action did not improve throughput, the algorithm avoids executing it again. To adapt to fluctuating workload, the authors introduce specific mechanisms to forget the recent history if the congestion index or the throughput change significantly. The algorithm uses a *sensitivity* parameter to determine when a significant change happens. Further details can be found in Ref. [2].

Tab. **??** shows the results for the real dataset scenario. Since the rule-based strategy and the heuristic one target throughput optimization without frequency scaling, we compare them with `Th-Node`. We use a horizon of 2 steps that, as we have seen, achieves the best SASO trade-off. For the comparison we use a control step of 4 seconds because the SPL-strategy is very unstable with too frequent steps. This is an important shortcoming that makes this strategy unable to track the workload with a fine-grained sampling. The best values for the congestion threshold and sensitivity parameters are 0.1 and 0.9.

	No. reconf.	**QoS viol.**	**Ampl.**	**No. replicas**
Rule-based*	39.17	62	1.07	4.63
Rule-based**	29	59	1.06	4.58
SPL-strategy	40.18	58	1	4.63
Th-Node (4s)	11	56	1	4.51
Th-Node (2s)	24.77	54	1.04	4.50

Table 7: Comparison with existing works. * $\theta_{max} = 0.9$ and $\theta_{max} = 0.8$. ** $\theta_{max} = 0.95$ and $\theta_{max} = 0.8$.

The results show that our approach is the winner. Fewer reconfigurations are performed (*stability*) with fewer violations (*accuracy*). Our approach uses a smaller number of replicas on average (*overshoot*) with a comparable amplitude (*settling time*). This is a confirm of the effectiveness of our predictive model-based approach.

6. Related Works

Stream Processing Engines (SPEs) are available both as academia prototypes [3, 24, 4], open-source solutions [32, 33] and industrial products [31, 34]. In their early days SPEs managed dynamic situations either by over-provisioning resources or by means of load shedding [1]. The first solution is not cost-effective, while the second one consists in discarding a fraction of the input stream to alleviate the stream pressure.

Elasticity is a recent feature of SPEs. Most of the existing works propose reactive strategies. In Refs. [3, 4, 5] the authors use a set of threshold-based rules on the actual CPU utilization by adding or removing computing resources accordingly. Other works use more complex metrics. In Ref. [14] the mean and standard deviation of the service time and the inter-arrival time are used to reactively enforce latency constraints. In Ref. [2] the strategy measures a congestion index and the throughput achieved with the current number of replicas. To the best of our knowledge, this is the only work in addition to our that targets the SASO properties. Our approach follows a different vision as we propose a model-based predictive approach instead of a heuristic-based reactive one.

Although predictive strategies have been applied to the control of data centers and clouds [21], they are essentially new in DaSP. As far as we know, the work in Ref. [6] is the only one before this paper that tries to apply a predictive approach in SPEs. It leverages the knowledge of the future to plan a smart resource allocation. The approach has been evaluated using oracles that give exact predictions. Some experiments take into account possible prediction errors, but they do not use real forecasting tools.

All the previous works except [14] are not optimized for low latency. In contrast our strategy exhibits a high degree of flexibility. Based on the mathematical formulation of the MPC problem, we are able to address both throughput and latency constraints. In Ref. [5] the authors study how to minimize latency spikes during the state migration. We have studied this problem in this paper, but we extend this vision by making the strategy fully latency aware: the MPC-based strategy is able to compare different configurations in terms of their expected latency.

All the previous works take into account only the number of used nodes. Instead, our strategies address power consumption on DVFS-enabled CPUs. Few works target power consumption on stream processing, e.g., the work in Ref. [35] provides a power-aware scheduler for streaming applications. However, it does not propose any elastic support to resource scaling.

7. Conclusions and Future Work

In this paper we studied a predictive approach to elastic partitioned-stateful stream operators on multicores. Our approach is based on MPC and is able to control power consumption by accommodating throughput and latency requirements. We evaluated our strategies in a high-frequency trading application by showing the capability to achieve good trade-offs among the SASO properties.

In the future we plan to extend our work in several directions. The current development effort in FastFlow is targeting distributed-memory architectures. Therefore, we plan to extend our work on clusters. Furthermore, our future aim is to integrate the MPC strategy for single partitioned-stateful operators in a complete context, in which the strategies of different elastic operators or applications (executed on the same machine) need to coordinate to find an agreement in their scaling decisions.

Acknowledgments

This work has been partially supported by the EU H2020 project RePhrase (EC-RIA, H2020, ICT-2014-1).

References

[1] H. Andrade, B. Gedik, D. Turaga, Fundamentals of Stream Processing, Cambridge University Press, 2014, cambridge Books.

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

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

[4] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, Integrating scale out and fault tolerance in stream processing using operator state management, in: Proc. of the 2013 ACM SIGMOD Int. Conference on Management of Data, SIGMOD '13, ACM, New York, NY, USA, 2013, pp. 725–736.

[5] T. Heinze, Z. Jerzak, G. Hackenbroich, 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, ACM, New York, NY, USA, 2014, pp. 13–22.

[6] A. Kumbhare, Y. Simmhan, 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, 2014, pp. 344–353.

[7] E. F. Camacho, C. Bordons (Eds.), Model predictive control, Springer-Verlag, Berlin Heidelberg, 2007.

[8] T. De Matteis, G. Mencagli, Keep calm and react with foresight: Strategies for low- latency and energy-efficient elastic data stream processing, in: Proceedings of the 21th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, ACM, New York, NY, USA, 2016.

[9] B. Gedik, Partitioning functions for stateful data parallelism in stream processing, The VLDB Journal 23 (4) (2014) 517–539.

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

[11] N. R. Herbst, N. Huber, S. Kounev, 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, ACM, New York, NY, USA, 2013, pp. 187–198.

[12] R. E. Kalman, A new approach to linear filtering and prediction problems, Transactions of the ASME–Journal of Basic Engineering 82 (Series D) (1960) 35–45.

[13] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, R. Rajkumar, Critical power slope: Understanding the runtime effects of frequency scaling, in: Proceedings of the 16th International Conference on Supercomputing, ICS '02, ACM, New York, NY, USA, 2002, pp. 35–44.

[14] B. Lohrmann, P. Janacik, O. Kao, Elastic stream processing with latency guarantees, in: Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on, 2015, pp. 399–410.

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

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

[17] J. Li, J. Martinez, Dynamic power-performance adaptation of parallel computation on chip multiprocessors, in: High-Performance Computer Architecture, 2006. The Twelfth International Symposium on, 2006, pp. 77–87.

[18] A. Chandrakasan, R. Brodersen, Minimizing power consumption in digital cmos circuits, Proc. of the IEEE 83 (4) (1995) 498–523.

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

[20] G. Mencagli, M. Vanneschi, Towards a systematic approach to the dynamic adaptation of structured parallel computations using model predictive control, Cluster Computing 17 (4) (2014) 1443–1463.

[21] S. Abdelwahed, J. Bai, R. Su, N. Kandasamy, On the application of predictive control techniques for adaptive performance management of computing systems, Network and Service Management, IEEE Transactions on 6 (4) (2009) 212–225.

[22] M. Danelutto, M. Torquati, Structured parallel programming with "core" fastflow, in: V. Zsók, Z. Horváth, L. Csató (Eds.), Central European Functional Programming School, Vol. 8606 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 29–75.

[23] M. Shah, J. Hellerstein, S. Chandrasekaran, M. Franklin, Flux: an adaptive partitioning operator for continuous query systems, in: Data Engineering, 2003. Proceedings. 19th International Conference on, 2003, pp. 25–36.

[24] Y. Wu, K.-L. Tan, Chronostream: Elastic stateful stream computation in the cloud, in: Data Engineering (ICDE), 2015 IEEE 31st International Conference on, 2015, pp. 723–734.

[25] A. Kumbhare, M. Frincu, Y. Simmhan, V. K. Prasanna, Fault-tolerant and elastic streaming mapreduce with decentralized coordination, in: Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on, 2015, pp. 328–338.

[26] H. Andrade, B. Gedik, K. L. Wu, P. S. Yu, Processing high data rate streams in system s, J. Parallel Distrib. Comput. 71 (2) (2011) 145–156.

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

[28] R. Fried, A. George, Exponential and holt-winters smoothing, in: M. Lovric (Ed.), International Encyclopedia of Statistical Science, Springer Berlin Heidelberg, 2014, pp. 488–490.

[29] M. Hähnel, B. Döbel, M. Völp, H. Härtig, Measuring energy consumption for short code paths using rapl, SIGMETRICS Perform. Eval. Rev. 40 (3) (2012) 13–17.

[30] M. C. Huebscher, J. A. McCann, A survey of autonomic computing&mdash;degrees, models, and applications, ACM Comput. Surv. 40 (3) (2008) 7:1–7:28.

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

[32] Apache storm, https://storm.apache.org.

[33] Apache spark streaming, https://spark.apache.org/streaming.

[34] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, Millwheel: Fault-tolerant stream processing at internet scale, Proc. VLDB Endow. 6 (11) (2013) 1033–1044.

[35] D. Sun, G. Zhang, S. Yang, W. Zheng, S. U. Khan, K. Li, Re-stream: Real-time and energy-efficient resource scheduling in big data stream computing environments, Information Sciences 319 (2015) 92 – 112, energy Efficient Data, Services and Memory Management in Big Data Information Systems.