Power-Aware Pipelining with Automatic Concurrency Control

Received XX Month 2017; | Revised YY Month 2017; | Accepted ZZ Month 2017

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

Power-Aware Pipelining with Automatic Concurrency Control

Massimo Torquati[1] | Daniele De Sensi[1] | Gabriele Mencagli[1] | Marco Aldinucci[2] | Marco Danelutto[1]

[1]Department of Computer Science, University of Pisa, Italy
[2]Department of Computer Science, University of Turin, Italy

Correspondence
Massimo Torquati, Department of Computer Science, University of Pisa, Largo B. Pontecorvo 3, I-56127 Pisa, Italy. Email: torquati@di.unipi.it

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

Summary

Continuous streaming computations are usually composed of different modules, exchanging data through shared message queues. The selection of the algorithm used to access such queues (i.e. the *concurrency control*) is a critical aspect both for performance and power consumption. In this paper we describe the design of automatic concurrency control algorithm for implementing power-efficient communications on shared-memory multicores. The algorithm automatically switches between `nonblocking` and `blocking` concurrency protocols, getting the best from the two worlds, i.e. obtaining the same throughput offered by the `nonblocking` implementation and the same power efficiency of the `blocking` concurrency protocol. We demonstrate the effectiveness of our approach using two micro-benchmarks and two real streaming applications.

KEYWORDS:
Data Streams, Data Pipelining, Blocking, Concurrency Control, Power Saving, Multicores

1 | INTRODUCTION

In the realm of parallel and distributed computing, *throughput* and *latency* have been traditionally considered the primary metrics to evaluate the performance of computing systems (1). However, in recent years *power consumption* has gained more and more significance up to the point it reached the same importance of traditional metrics (2). As a consequence, system optimization is played against multiple concerns, performance and power consumption at least, to dynamically find acceptable trade-offs during the processing of real-world workloads.

The optimization of the performance/power trade-off has been mainly pursued by means of *dynamic reconfigurations* of the system at different abstraction levels, from the hardware level up to the run-time system and the application levels by employing suitable synchronization mechanisms and algorithms. Their principal goal is to reduce power consumption with limited impact on performance. Examples are Dynamic Voltage and Frequency Scaling (DVFS) (3), approximate computing (4), energy-efficient data structures (5) and adaptive locking (6).

A recent work has demonstrated that a simple and effective power saving technique consists in optimizing the synchronization mechanisms (7). In a shared-memory system, a standard approach to synchronize producer/consumer interactions between pairs of threads relies on a concurrent FIFO queue that supports `push` and `pop` operations in an atomic and efficient way. This approach is particularly relevant in the *Data Stream Processing* paradigm (8), where applications are modeled as arbitrarily complex graphs of modules/operators exchanging unbounded sequences of data items through channels implemented by concurrent queues.

A naive implementation consists in protecting the access to a concurrent queue by using *mutex* mechanisms. If the thread that currently holds the mutex is delayed, all the other threads attempting to access the data structure are delayed too. Furthermore, acquiring the mutex implies *passive waiting*, that is the suspension of all the threads waiting for the mutex acquisition. The suspended threads are moved in a waiting queue and their core/hardware contexts are released to the OS. In addition, passive waiting can be used as the basic mechanism to handle synchronization events during the usage of the queue, like when the producer tries to push a data item in a full-size queue or when the consumer attempts to pop from an

empty queue. *Suspended threads do not directly consume power*. However, suspension and mainly restart mechanisms may impair application performance due to many factors such as the waiting time in the ready queue, context switch overhead, compulsory cache misses or core migration (7). In general, a concurrent algorithm that may force the calling thread to be blocked waiting that a given condition holds is defined as `blocking`.

Concurrent queues can be implemented in a efficient and scalable way by using `nonblocking` algorithms. Although many definitions exist and have been utilized in the literature, in this paper we consider a concurrent algorithm as `nonblocking` when thread suspension cannot be caused by synchronizations related to the data structure usage (of course a thread can still be de-scheduled by a time-sharing scheduler or due to preemption). Therefore, a solution that replaces passive waiting phases with a busy-waiting *spin-loop* (e.g., by replacing mutexes with spin-locks) are accepted as `nonblocking` according to this definition. One of the drawbacks of non-blocking algorithms is that the busy-waiting phase consumes CPU cycles and power without doing useful task, and the approach may impair throughput when there are more threads than available hardware contexts/cores.

An interesting and widely studied class of `nonblocking` algorithms are the ones classified as *lock-free*. This term refers to the fact that the failure or the suspension of a thread in any arbitrary point during its execution cannot prevent at least one thread in the system to make progress (9). As an example, a spin-lock based queue is not lock-free because if a thread fails after the lock acquisition none of the other threads will be able to correctly complete its operation on the queue. In a lock-free concurrent queue the producer and the consumer threads are able to push and pop elements concurrently by working on different positions of the queue. This does not necessarily mean that threads do not have to take into account their mutual interference. For instance, a thread may start a push/pop operation and, if the queue state changes due the access by another thread, this must be detected and the operation restarted until it is correctly executed. Such actions (occurring in while-loops) may consume CPU cycles and power although the degree of concurrency inside the queue's code is maximized.

While lock-free algorithms are mainly chosen for their progress guarantees, they are also employed for their higher throughput and lower latency (9) (5). Furthermore, by avoiding the threads to be de-scheduled in the synchronization phases, they contribute to reduce the so-called *OS noise* which may be a source of scalability problems in many High Performance Computing applications (10). Unfortunately, as said, the `nonblocking` approach is not power-efficient due to the busy-waiting loop executed when a given operation cannot be immediately concluded (e.g., CAS retry loop). Although several approaches have been proposed to reduce the power consumption during busy-waiting loops, for example using pause, memory barriers or monitor/mwait instructions (7), none of them revealed completely successful on the off-the-shelf commodity multicores and busy-waiting is *de-facto* considered not power efficient.

A way to improve power saving is to delay the busy-waiting phases with short phases of passive waiting obtained by executing micro-sleep system calls. This technique is called *backoff* and has been widely adopted in the implementation of spin-locks (11). Theoretically, if the micro-sleep phases are perfectly regulated each time by an oracle, the achievable performance/power trade-off can be optimal (same performance of the pure busy-waiting approach with almost the same power saving of passive waiting). However, this is unlikely in real situations and the use of wrong values could have dramatic effects on the reactivity of the system or on its power consumption. Furthermore, the tuning phase can hardly be automatized and the solutions are in general not portable since the sleeping intervals should be regulated for each machine and application, and it might be even impossible to use sufficiently fine-grained sleeps in some OSs [1].

Instead of optimizing the backoff technique, this paper proposes a different approach that breaks the dichotomy between `blocking` *vs* `nonblocking` techniques that have been often used as mutual exclusive alternatives in the past. Our solution is designed for *data stream processing* scenarios, where thread synchronization happens on the use of lock-free concurrent queues. To exemplify the problem, we show in Fig. 1 the analysis we performed on a network streaming application where a continuous flow of data packets is analyzed in real-time. We tested the application with two different input rates: the first is of 350K packets per second while the second has about 1M packets per second. In both cases, we collected the throughput and power consumption measures obtained by two configurations of the application, the first using a `blocking` concurrency mode, the second uses a `nonblocking` mode.

Interestingly, with the slowest input rate the two versions achieve almost the same throughput, whereas the power consumption is significantly in favor of the `blocking` concurrency mode. This result is obtained because in that case the application is fast enough to process the packets at their arrival speed, and it often idle waiting for new packets. The `blocking` version optimizes power consumption in those phases by suspending the application threads. Furthermore, the wake-up overhead is negligible compared with the packet inter-arrival time.

In the case of the high input rate the pressure to the system is intensive and on average there are always packets to consume from the network. Hence, the idle time is minimized and the two versions consume comparable power. The extra-overhead of the `blocking` version (e.g., for waking up suspended threads) impairs throughput because the inter-arrival time interval is short and a slight improvement of the thread processing speed immediately delivers better throughput.

The solution that we propose is to automatically switch between the two concurrency modes according to the actual properties of the incoming workload. Each queue supports both the concurrency modes that can be switched transparently to the user. This strategy is used to adapt graphs of streaming operators working on single-producer single-consumer (SPSC) queues. Although the switching decision requires some tuning, we

[1] For example, on Windows OSs is difficult to put one thread to sleep for less than one millisecond.

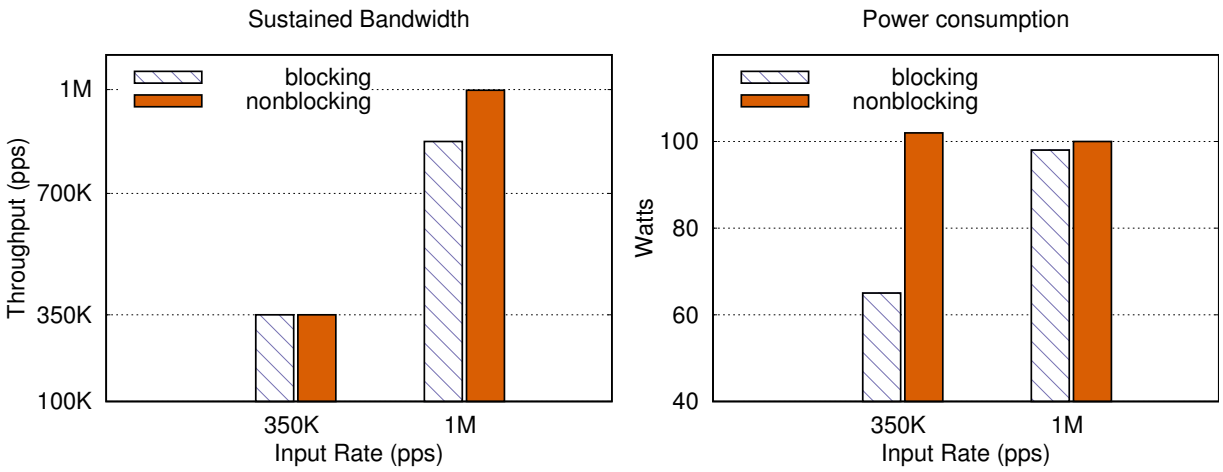

FIGURE 1 Motivating example: throughput and power consumption of the `blocking` *vs* `nonblocking` concurrency control modes in a network application with different input rates.

design our algorithm so that the tuning is done only once per architecture by executing some micro-benchmarks, being independent from specific workloads and applications. To confirm the effectiveness of our approach, we validate the algorithm using two real streaming applications.

The outline of this paper is the following. Sect. 2 provides background concepts useful for understanding the rest of the reading. Sect. 3 describes the design of our algorithm and its application to the lock-free queues adopted by the FastFlow parallel programming framework. Sect. 4 shows an evaluation using synthetic benchmarks and two real-world applications. Sect. 5 describes some related works while Sect. 6 provides the conclusion of this work and outlines possible future research directions.

2 | BACKGROUND

In this section we will describe the background concepts that are useful to understand the contribution of this paper. We will discuss the main features of data stream processing both in terms of its computing paradigms and programming models. Then, we will focus on the mechanisms developed in past research work and adopted in current frameworks to balance performance with power consumption. This description will contextualize precisely which frameworks and tools will benefit from our power-aware data pipelining mechanism.

2.1 | Data Stream Processing

Data stream processing frameworks support the development of applications that continuously process unbounded flows of input elements to be processed on-the-fly. Applications are modeled as data-flow graphs of operators that compute in parallel over subsequent and independent data items, and communicate partial results via data streams (12). The stream processing approach is worth for a wide class of applications, not limited to cases where the input stream is primitive, e.g., when it is generated by external sources like sensors, networks or I/O devices, but also when it is generated by reading from an in-memory large data structure (13).

In most of the existing frameworks like Apache Storm (14), StreamIt (15), FastFlow (16) and IBM InfoSphere Streams (17), operators can be replicated to increase performance when hotspots are discovered. Besides the default distribution policies, the programmer may be eventually in charge of programming how input items are dispatched to the distinct replicas of an operator, and of building the output stream by respecting the input ordering. In that vein, some libraries promote the use of predefined patterns to instantiate parallel versions of operators, like *pipeline* and *farm* (18). This approach has the merit to reduce the programming effort and to help in writing correct and deadlock-free applications, although the expressiveness in writing arbitrarily complex graphs can be slightly hampered.

Operators represent the basic building block of graph topologies, which perform pre-built or user-defined transformations of the input data streams into output data streams. *Stateless* operators apply the processing logic on each input item independently of the others. Instead, in *stateful* operators the output of the computation on each input item depends on the value of the item itself and also on the history of the past items seen by the operator. Of particular interest are *windowed operators* that maintain the most recent items received so far (e.g., the last $n > 0$ received items or all the items received in the last $n > 0$ time units) and apply a user-defined function on each window data group (19). Except the case when the window results can be computed incrementally, i.e. the result is updated each time a new item of that window is available, in common window-based

computations the user-function is triggered only when the window is full. Hence, no significant computation cost is paid when non-triggering items are received, because they need just to be stored into a proper buffer thus paying few clock cycles for that. Since the triggering of the user function is infrequent than the arrival of new items, windowed operators may often have a low computation time per item on average which reflects in a pressing need of low communication overhead for data exchange between operators in order to achieve high throughput.

2.2 | Run-time Adaptation Mechanisms

The input workload of data stream processing applications is often highly variable, both for causes related to the user behavior, different application phases, network outages and based on the intrinsic variability of the computation time of user-defined functions, applied over individual input items or window data groups. Over the years, several run-time mechanisms have been developed to adapt the application structure in terms of operator deployment and their replication degree based on the monitored workload.

One of the most common reconfiguration mechanism is *concurrency throttling* (called *operator fission* in stream processing), which consists in dynamically changing the concurrency level used by an operator in order to accommodate the incoming workload by meeting the application performance constraints. Most of the existing research papers (20, 21) on this topic have focused on performance, because stream processing applications are demanding in terms of service-quality requirements (like in automatic financial trading systems). Because of the downtime potentially caused by changes in the concurrency level (e.g., for state management activities (22, 23)), advanced strategies have been designed to make reconfigurations only when strictly necessary, avoiding oscillating decisions that may seriously impair performance instead of improving it (3, 24). Even more intrusive are *structural changes* (25), where computationally lightweight operators can be fused together in order to reduce communication latencies. In modern stream processing systems no support exists for performing such reconfigurations on-the-fly. Indeed, the application needs to be stopped and resumed with the new structure, thus hampering the possibility to use structural changes frequently in case of high-speed streams.

Workload consolidation (26) (also known as *Thread Packing*) have been proposed as an alternative to concurrency throttling. In fact, instead of changing the number of threads implementing an operator, they try to pack threads on the same core by reducing the reconfiguration time with comparable power efficiency (27). By packing as much as possible the running processes of a stream processing system over the available resources, it is possible to turn off under-utilized computing nodes, reducing the power consumption of the system. Another direction instead, is to exploit the DVFS (Dynamic Voltage and Frequency Scaling) support provided in most of the existing multicore architectures (3, 28). Instead of scaling out/in the number of threads, the runtime increases/decreases the working frequency of the CPU in order to save power/increase throughput based on the current workload. Other techniques are based on adaptive scheduling strategies (29, 30), which decide how to execute operator replicas over the cluster nodes to achieve trade-offs between resource consumption and performance.

As far as we know, there are few papers that address the balance between throughput and power efficiency at the level of concurrency mechanisms. The problem can be tackled on two fronts: one is oriented to efficient data structures supporting specific operators; the other is to provide general-purpose synchronization mechanisms that work at a low level, i.e. at the level of the basic cooperation mechanisms used in the runtime of stream processing systems. The paper presented in (31) proposes ScaleGate, a concurrent data object enabling low-latency operations for merging multiple input streams into an outbound stream ordered by timestamps. The data structure allows a set of readers to consume ready tuples based on their timestamp by guaranteeing deterministic processing. The same idea has been recently extended to support multiway aggregation operators in (32), showing high-throughput and low-latency levels with respect to traditional implementation techniques.

None of the previous papers address the problem of optimizing the performance/power trade-off. Although synchronization mechanisms play a crucial role in case of fine-grained operators fed by high speed streams, the application workload is far to be stable in stream processing systems, and the use of performance-friendly or power-friendly mechanisms can be regulated automatically based on the workload. Once the features of the incoming workload are recognized, a synchronization mechanism equipped with *automatic concurrency control* is able to choose one of the different working modes that fit better with the current workload scenario, i.e. choosing a highly reactive mode in case of high speed streams, where a small optimization of the operator's processing speed might reveal of great importance for throughput, or using less aggressive concurrency modes to save power without impairing performance when the arrival rate allows the application to be often idle waiting new data items to process.

The switching logic between different concurrency modes is the hardest part to design. The logic must have a minimal overhead in the standard processing flow since it is executed continuously. Furthermore, the application of the switching actions must be immediate on the application dataflow graph in order to make the mechanism able to rapidly receive reliable feedbacks from the system's execution.

3 | DESIGN AND IMPLEMENTATION

In this section we describe the design of a shared-memory message queue that can be used both in `blocking` and `nonblocking` concurrency control modes for a generic data streaming computation. Then, we describe how to use it to optimize the power consumption and performance of a generic streaming graph.

3.1 | Base Mechanisms

An effective way for implementing pipeline parallelism between two threads on multicores is to use a lock-free Single-Producer Single-Consumer (SPSC) FIFO queue (33, 34). As discussed in Sec. 2 this approach is not power-efficient if a thread is performing busy-waiting because the underlying hardware context remains active so that the OS cannot set the core in a low-power state.

In case of variable arrival rates, the *producer* (P) or the *consumer* (C) threads might spent some time in a busy-waiting loop because the message queue is full or empty. Rather than spinning, to reduce power consumption we want to put the threads to sleep waking them up as soon as they can make useful work. The only portable way for doing this is to uses POSIX `mutexes` and `condition variables` (or equivalent C++1x features).

We consider FastFlow (16) as the reference parallel framework. We extended the FastFlow concurrency mode by associating to each FastFlow channel a POSIX mutex and a condition variable and by changing the `push` and `pop` operations as described in the following pseudo-code.

Algorithm 1: CCPush

```
1:ok=Q.push(data);
2: if ok then
3:  if (CCM==blocking and
4:       C_waiting) then
5:       signal C_cond_in;
6: else
7:   if (CCM==blocking and
8:      Q.isFull) then
9:     wait_on P_cond_out;
10:   goto 1;
11: return success;
```

Algorithm 2: CCPop

```
1:ok=Q.pop(data)
2: if ok then
3:   if (CCM==blocking and
4:       P_waiting) then
5:       signal P_cond_out;
6: else
7:   if (CCM==blocking and
8:      Q.isEmpty) then
9:     wait_on C_cond_in;
10:   goto 1;
11: return success;
```

Let us describe the Algorithm 1. To `push` a message into the output queue the runtime first pushes the data pointer into the lock-free SPSC queue `Q` (line 1); if it succeeds, depending on the current concurrency mode of the node (called `CCM` in the algorithms), two different actions are taken. In case of `nonblocking` mode the operation has been successfully completed without any further step (returning success at line 11). If `CCM=blocking`, the runtime checks if the consumer node (C) has to be woken up because it has been previously put to sleep waiting for a new message (testing `C_waiting` at line 4). In that case, C will be awakened by an explicit `signal` on its input condition variable `C_cond_in` (line 5) and the operation returns with success (line 11). If the `push` on the lock-free queue fails and `CCM=nonblocking`, the push operation is executed again until it will complete with success (line 10). If `CCM=blocking`, then the runtime checks if the queue is full (line 8) and, in that case, the thread is put to sleep on its output condition variable `P_cond_out` (line 9). If it is not full (this is a spurious condition that may happen with lock-free data structure), or if the thread is woken up by a signal, the operation is restarted from the beginning (line 10).

Let us now consider the `pop` operation described in the Algorithm 2. The runtime pops a new data pointer from the lock-free queue (line 1). If the operation succeeds and `CCM=nonblocking`, the operation has been successfully completed (line 3 and line 11). If `CCM=blocking` then the runtime checks if the producer (P) has to be woken up because it is waiting for a new free slot in the queue (line 3 and 4). This case can happen only if the input queue has a bounded size. In that case, P receives a signal on its output condition variable and the operation completes with success (line 5 and 11). If the `pop` fails and `CCM=nonblocking` the operation will be repeated until it completes with success (line 10). If `CCM=blocking` then the runtime checks if the queue is empty (line 8) and puts the thread to sleep on its input condition variable (line 9), otherwise the operation is restarted from the beginning (line 10).

By switching the `CMM` variable between `blocking` and `nonblocking` concurrency mode, it is possible to control the throughput and the power consumption of the nodes using the queue. The next section will explain this mechanism more in detail.

3.2 | Power-Aware Data Pipelining

To describe the algorithm, we first consider a streaming network structured as a pipeline, a connected graph where each node has at most one input queue and one output queue. Then, in Sec. 3.3 we extend the algorithm to generic streaming networks.

To simplify the exposition, from now on we consider that all message queues are *unbounded* in size. Consequently, each node would never need to do busy-waiting or to suspend itself when doing a `push` operation. This may cause an uncontrolled growth of memory usage and we will discuss this aspect in Sect. 3.3.

As described in Sec. 3.1, by changing the `CCM` variable it is possible to change the concurrency mode of the producer and consumer thread. But, who decides if it is worth to switch from `blocking` to `nonblocking` and vice-versa for a given pair of nodes?

Our implementation, considers a manager thread that is in charge of making decisions for the entire streaming application. At configurable time intervals, by collecting monitoring information about the current performance and power consumption of the entire application, the manager decides which message queue should operate in `blocking` or `nonblocking` concurrency control mode by directly notifying the producer and consumer threads. Each concurrent activity will execute the following operations in a loop:

1. Reads an element from its input queue by executing a `pop`. The average latency of this operation is L^b_{pop} for `blocking` queues and L^{nb}_{pop} for `nonblocking` queues. If no data is present in the queue (the `pop` fails), the node waits for new data to arrive by suspending itself or by doing busy waiting. Let us denote this average waiting time with L_{idle};
2. Executes some processing (with a latency L_{proc}) on the data element;
3. Sends the computed result(s) on its output queue through a `push`. This operation has an average latency of L^b_{push} in case of `blocking` queues and L^{nb}_{push} in case of `nonblocking` queues. Since the output queue is unbounded in size, this operation will always succeed.

The breakdown of a single loop iteration of a node is sketched in Fig. 2 . Timing values (e.g., L_{idle}, L_{proc}) are stored by the single node in its internal variables that can be accessed (read only) by the manager without any extra synchronization.

In the rest of this section we will describe the two possible cases for the dynamic switching of the concurrency control mode: *i)* from `blocking` to `nonblocking`; *ii)* from `nonblocking` to `blocking`. It should be noted that the manager must manipulate the `CCM` variable in mutual exclusion (a mutex is also needed in the POSIX condition variables API) and it has to wake up through a signal a thread possibly waiting on the queue to avoid critical cases when the consumer is blocked while the producer is already in the `nonblocking` mode.

3.2.1 | From blocking to nonblocking

Suppose that when the application starts, it uses all the message queues in `blocking` mode. To improve the throughput of the application, we have to improve the throughput of its slowest node, i.e., the one with the highest latency. We call this node S. If it has $L_{idle} > 0$, despite being the slowest node in the application, it is still fast enough to process the incoming data, so there is no need to improve the throughput of the application at all. Otherwise, we can improve the throughput of S by reducing the latency of both the `pop` and `push` operations. Let us start with the `pop` operation. Switching the input queue to `nonblocking` mode would have no impact on the power consumption since $L_{idle} = 0$ and S will not do busy wait. Now let us consider the `push` operation. We could switch the output queue of S to `nonblocking` mode and reduce L_{push} as well. Let us call T the successor of S. Since S is slower than T, $L_{idle}(T)$ is greater than zero. If the message queue is in `blocking` mode, while idling T is suspended on the condition variable. However, after switching to `nonblocking` mode, T will start doing busy-waiting, thus increasing the power consumption of the application. To determine if the increase in power consumption is worth the increase in performance, we decided to let the application user (or the system developer) to set some preferences for the application, by specifying maximum allowed increase in power consumption for each 1% increase in performance. Similarly, the user might just set a maximum power consumption of the system letting the runtime system to optimize the performance with the given power budget (this is also known as *Power Capping*).

For evaluating the outcome of the decision, we adopt a *rollback-based* approach. When a potential performance improvement for the output queue is detected, the algorithm switches the queue from `blocking` to `nonblocking`. Then, the performance and the power consumption are monitored for the next time interval. If the results of the switching does not comply with the user requirements, the decision is reverted back, otherwise is kept. Since in both cases we improved S by switching its input queue, the slowest node might now be a different one. If this is the case, the algorithm is executed on the new slowest node, otherwise it terminates. To avoid too many rollback operations, if a node was involved in a rollback operation, it is marked and it is not evaluated again for a time interval that can be specified by the user.

3.2.2 | From nonblocking to blocking

Due to workload fluctuations, the system could start receiving less data per unit of time. In such a case, the message queues will become empty and some nodes will start doing busy-wait on their input queues. By switching a queue to `blocking` mode, the nodes using the queue would suspend

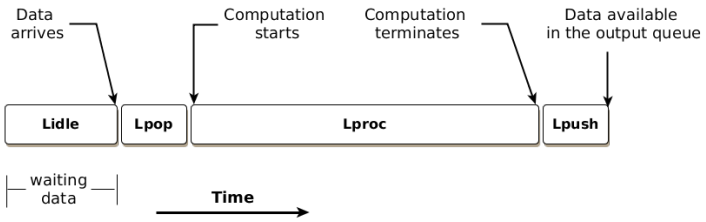

FIGURE 2 Different kind of latencies in the node operations.

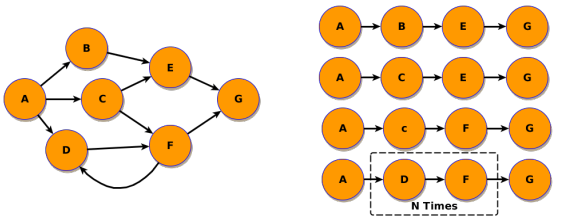

FIGURE 3 A data stream processing graph and all its possible routing paths.

on the condition variable instead of doing busy-wait. However we would also increase the latency of `push` and `pop` operations. To ensure that this switch does not decrease the throughput of the nodes, it is sufficient to ensure that the increase in the `push` and `pop` latency is "absorbed" by the idle latency, i.e., even if these operations will last longer, the nodes will still have enough time before receiving the next data element, thus not reducing their performance. To do so, it is sufficient to find the pairs of nodes P (producer), C (consumer) such that the following condition is true:

$$L_{idle}(C) > L^{b}_{pop}(C) - L^{nb}_{pop}(C) \textbf{ and } L_{idle}(P) > L^{b}_{push}(P) - L^{nb}_{push}(P)$$

If this condition is satisfied, we will have $L_{idle} > 0$ for both nodes after switching to `blocking`, thus not reducing their throughput.

3.3 | Generic Streaming Graphs

Here we discuss how to apply to a generic connected graph, the algorithm previously described for the pipeline graph.

We may observe that a data element, flowing from a *source* to a *sink* of the streaming network, will cross different processing nodes and different message queues. Since each node may have multiple output channels towards different nodes, the path followed by an input element depends both on the scheduling policies adopted by these multi-output nodes and by the data element itself. However, all the possible paths are statically known (see Fig. 3).

Since each of these paths is actually a pipeline, we can apply the algorithm described for the pipeline separately on each path. When computing all the possible paths we remove the backward edges, i.e. those forming a loop in the graph. Indeed, a loop simply replicates a sub-paths multiple times (*N Times* in Fig. 3) in the pipeline. However, it is sufficient to optimize each node of the path just once, thus the algorithm will still optimize the throughput of the application even if we do not consider these duplicate nodes. For example, two bottom paths in Fig. 3 (right) will actually be the same path for the purpose of the algorithm.

3.3.1 | Message queues' memory utilization

In the algorithm, we considered all the message queues used by the application to be unbounded in size. However, if the application receives data at a faster rate than the one it has been designed for, data would accumulate in the message queues, leading to an uncontrolled growth of memory utilization and to catastrophic effects on the application.

The application, however, has been designed to sustain a given maximum input data rate or not to exceed a certain memory utilization. Accordingly, the *source* nodes would throttle itself by not injecting data into the application at a rate faster than the one the application was designed for. In this way, the total amount of memory of the system is kept under a maximum value.

4 | EXPERIMENTS

In this Section we first analyze the results obtained considering two simple benchmarks aimed at measuring latency and throughput in different configurations, then we describe the results obtained validating the algorithm proposed in Sec. 3 for two real streaming applications: Network Protocol Identification and Malware Detection. Finally we characterize the applicability of the proposed algorithm by manually tuning the service time of the Malware Detection application and by running an image filtering application using OpenCV and thread over-provisioning.

All experiments were conducted on an Intel workstation with 2 Xeon E5-2695 @2.40GHz CPUs, each with 12 2-way hyper-threaded cores, running with Linux x86_64. We did not use hyper-threading and we ran all the experiments by selecting the maximum CPU clock frequency available through the `performance` scaling governor.

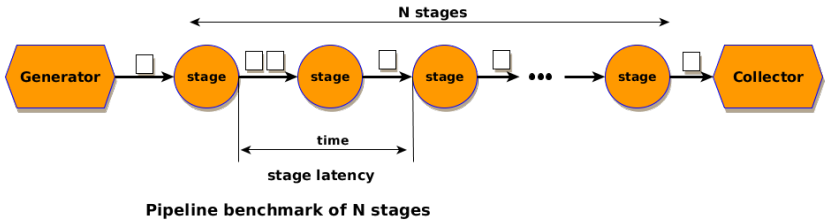

FIGURE 4 Pipeline benchmark of N stages aimed at measuring the *average latency* of each single stage, i.e. the average time needed to a packet to move from the previous stage to the next one.

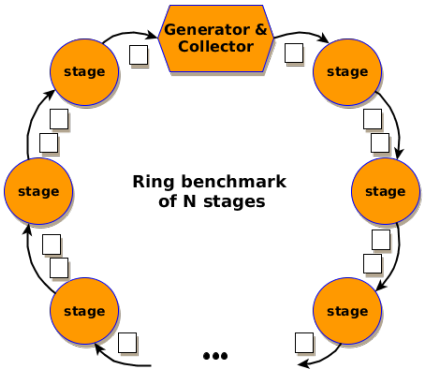

FIGURE 5 Ring benchmark of N stages aimed at measuring the maximum throughput sustained by the system varying the number of stages.

Benchmarks and applications have been implemented using the FastFlow parallel framework version 2.1.3. FastFlow is an open-source, structured parallel programming framework supporting highly efficient stream parallel computation on heterogeneous multi-core platforms (16). It is realized as a C++11 header-only template library that allows the programmer to simplify the development of parallel applications modeled as directed graphs of processing nodes. FastFlow provides a set of ready-to-use, parametric algorithmic skeletons modeling the most common parallelism exploitation patterns, which may be freely nested to model arbitrarily complex graph topologies.

To measure the power consumption, we used the MAMMUT[2] library (35), that on the multi-core system used for the tests relies on RAPL counters. The results reported in each plot are the average values of five repetitions, unless otherwise stated.

4.1 | Synthetic benchmarks

The first set of experiments considers two benchmarks: i) *pipeline benchmark* (Fig. 4), a linear chain of N stages where the first stage (the *Generator*) injects into the chain a continuous stream of packets at a pre-defined constant rate for a given amount of time (30 seconds in our tests); a *Collector* stage gathers all packets injected in the pipeline; ii) *ring benchmark* (Fig. 5), a ring of N stages where the master stage called *Generator & Collector* continuously sends packets to the first stage of the ring and receives back packets from the last stage of the ring. The number of packets injected at the maximum speed into the ring is fixed and equal to M. The last packet is a special packet (called *End-Of-Stream*) that allows to stop all stages and measure the execution time (called $TotalExecutionTime$).

The objective of the benchmarks is twofold: for the *pipeline benchmark* to measure the average latency for the single stage, that is the average time needed by a packet to cross a pipeline stage (i.e. the average time between the `push` of the packet by the previous stage and a `pop` operation by the next stage for receiving the packet); for the *ring benchmark* to measures the overall throughput of the system varying the number of stages of the ring. In the *ring benchmark* each stage spends $1,000$ CPU cycles before sending the packet to the next stage, instead in the *pipeline benchmark* each stage forwards the packet as soon as possible. The latency in the *pipeline benchmark* is computed by starting a timer for each packet before sending it into the chain and stopping the timer when the packet is received by the *Collector* stage. The *Collector* computes a moving average with an overlapping constant size of 10 values and eventually produce in output the average value divided by the $(N+1)$ (i.e the number of channels of the pipeline). The overall throughput in the *ring benchmark* is computed considering the total amount of messages exchanged during the benchmark execution time and precisely $M*(N+1)/TotalExecutionTime$ where M and N are the number of messages and the number of stages, respectively.

We studied three different configurations: *1)* `nonblocking` concurrency mode where all threads continuously keep polling their input/output queues with a minimal active backoff of few hundred CPU cycles; *2)* `blocking` concurrency mode where all threads are immediately put to sleep waiting for a wake-up signal if their input/output queues are empty/full; *3)* `backoff` where all threads uses different retry policy strategies alternating active polling and micro sleeping periods. For the `backoff` configuration, we consider two distinct cases: *a) backoff-2l* having two levels of retry policies: *aggressive* and *moderate*; *b) backoff-3l* having three distinct levels of retry policy: *aggressive*, *moderate* and *relaxed*. In the *backoff-2l* the *aggressive* policy performs 256 active polling tentative then it switches to *moderate* strategy where, between two distinct retries, the polling thread sleeps for 1 millisecond. In the *backoff-3l* case, the *aggressive* policy performs 64 tentative of active polling then it switches to *moderate* where the thread performs 256 tentative and between two distinct retries it sleeps for 50 microseconds then, if the queue is still empty/full, it switches again moving

[2]http://danieledesensi.github.io/mammut/

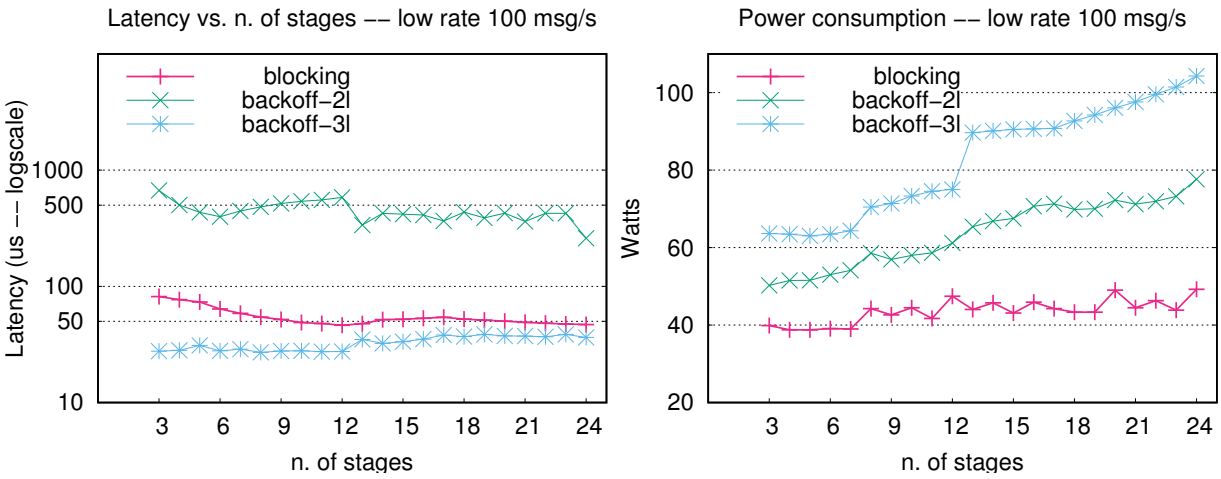

FIGURE 6 Latency and Power consumption of the pipeline benchmark in *blocking* and *backoff* configurations for "low" message rate (100msg/s).

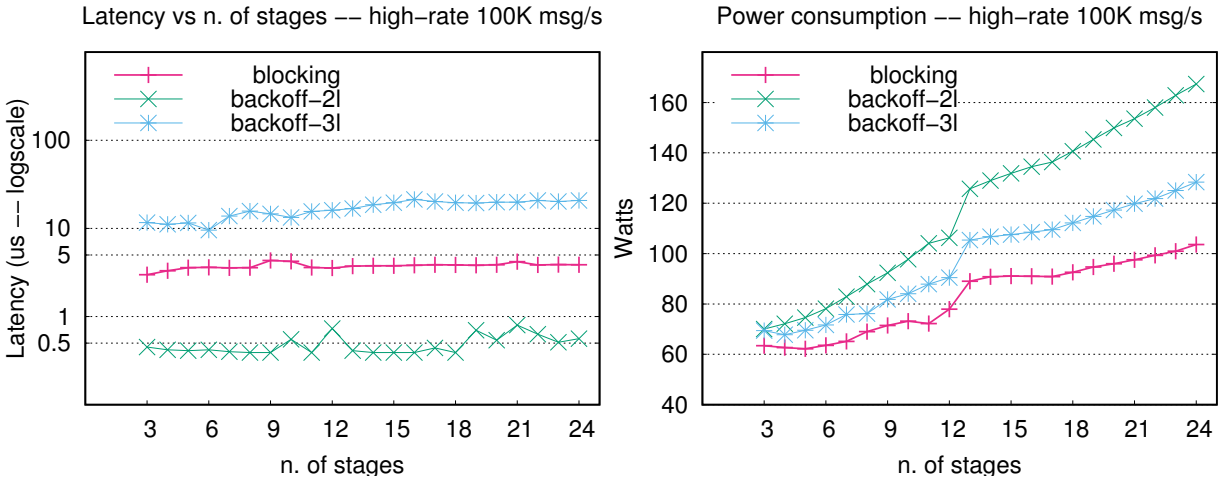

FIGURE 7 Latency and Power consumption of the pipeline benchmark in *blocking* and *backoff* configurations for "high" message rate ($100K$msg/s).

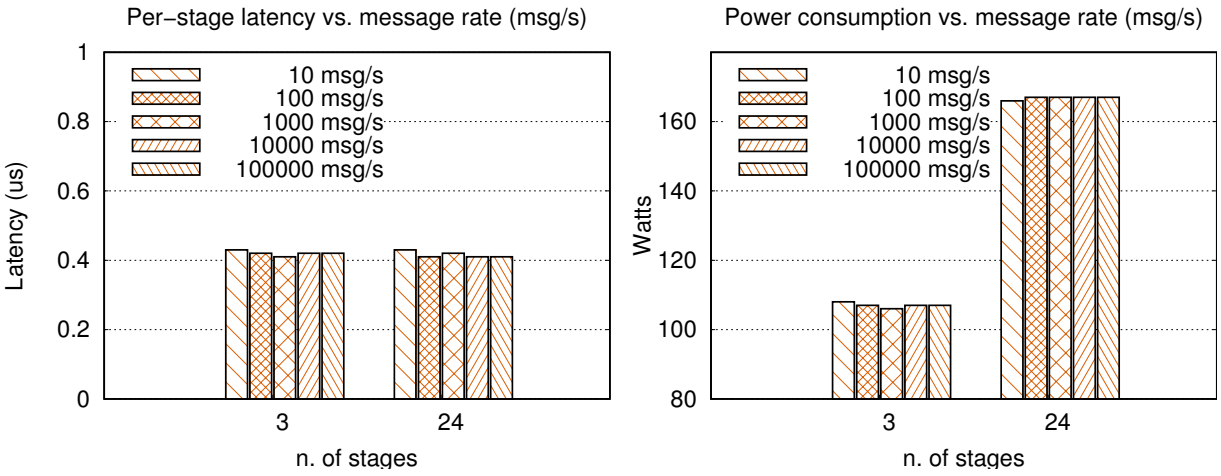

FIGURE 8 Latency and Power consumption varying the message rate for the pipeline benchmark in the `nonblocking` configuration.

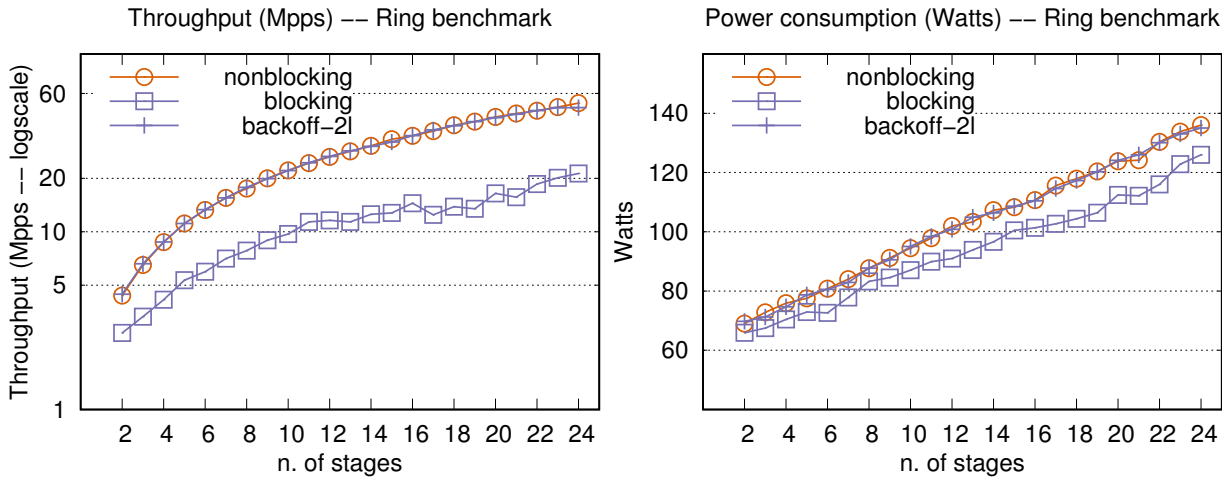

FIGURE 9 Throughput and Power consumption of the ring benchmark.

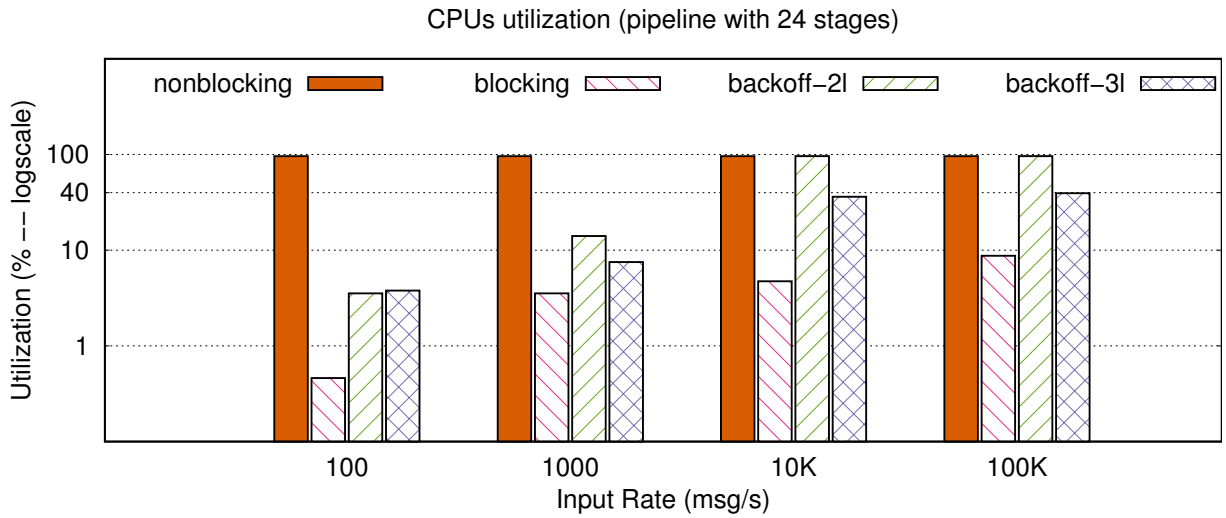

FIGURE 10 CPUs utilization (%) for the pipeline benchmark with 24 stages considering 100,1000 and $100K$ msg/s.

to the *relaxed* mode where the threads sleeps for 1 millisecond between two distinct retries. The *backoff-2l* is characterized by a more aggressive behavior where the thread spins or sleep, whereas the *backoff-3l* more gradually moves from spinning to sleeping over time. We select these specific values for sleeps and retries since they provide a good trade-off between power consumption and performance.

Fig. 6 shows the results of the *pipeline benchmark* in the `blocking` and `backoff` configurations for low message rate (10msg/s). For such low rate the best average latency is obtained using the *backoff-3l* configuration which has a *moderate* policy that performs small sleeps (50 microseconds) whereas the *backoff-2l* has a longer sleeping period (1 millisecond). It is worth noting that increasing the number of stages the `blocking` and *backoff-3l* case have closer average latencies. Concerning power consumption, as expected, the `blocking` concurrency mode is the most power efficient consuming less than 50 Watts in all tests.

The results obtained with higher input message rage ($100K$ msgs/s) are reported in Fig. 7 . In this test, a packet is produced each 10 microseconds. The best latency is obtained by the *backoff-2l* configuration while the *backoff-3l* is the worst configuration. This can be explained considering that the *backoff-2l* has a higher number of retry attempts in the *aggressive* strategy. In fact, it performs 256 retries of active polling instead of 64 retries of the *backoff-3l*. Therefore, with the given rate, in the *backoff-3l* the spinning thread will enter the retry loop of the *moderate* strategy which between two consecutive retries the thread is put to sleep for at least 50 microseconds which is a time higher than the actual rate. Conversely, the *backoff-2l* is the most power-hungry strategy consuming in the 24 stages configuration 167 Watts.

In Fig. 8 we show the average latency and the power consumption for the *pipeline benchmark* using the `nonblocking` configuration, varying the message rate between 10msg/s to $100K$msg/s. We considered two configurations for the pipeline: 3 and 24 stages. As can be seen, the latency is almost constant (about 400 nanoseconds) for the two cases regardless the input message rate and the number of stages of the pipeline chain. The

same applies for the power consumption that is constant regardless the input rate. The difference between the case with 3 and 24 pipeline stages is due to number of CPUs used in the two configurations, in fact each CPU of the platform considered for the tests hosts 12 cores. Therefore, in the pipeline configuration with 3 stages only the cores of the first CPU are used, allowing the OS to put the second CPU in low-power mode.

Concerning the throughput, in Fig. 9 is shown the results obtained by the `nonblocking`, `blocking` and *backoff-2l* configurations. As expected, the `blocking` concurrency mode offers the lowest overall system throughput due to the higher overhead of the concurrency mechanisms, whereas the `nonblocking` and the *backoff-2l* configurations have exactly the same system throughput reaching a maximum value of about 55 Mpps (millions-packets-per-second).

Finally, in Fig. 10 is reported the average CPU utilization of the *pipeline benchmark* for the case with 24 stages. As can be noted, the `nonblocking` configuration uses almost 100% of the CPU cycles available regardless the input rate, the `blocking` configuration uses less than 1% of CPU cycles for low message rate (100 msg/s) and about 10% CPU cycles for high message rate. The *backoff* configurations consumes much less of the `nonblocking` concurrency mode for low message rate while for high message rate the `backoff` concurrency mode consumes about 95% and 40% CPUs cycles in the *backoff-2l* and *backoff-3l* configurations, respectively.

In summary, on the base of the results of the two benchmarks, we can conclude that the `nonblocking` concurrency model is the most efficient and stable one when considering latency reduction as the most important metric. On the other hand, it is also the most power expensive both in terms of CPUs cycles and in terms of power consumed. The `backoff` and `blocking` concurrency models offer different performance results on the base of the given input rate. Moreover, the tuning of the backoff's sleeping time and number of tentative for each strategy, is not an easy task and need to be regulated on the base of the rate. The `backoff` strategy is a good compromise if the most important optimization metric is the throughput, in fact it can offer the same performance of the `nonblocking` strategy at high rate and to offer almost the same power consumption of the `blocking` strategy for low input rate. When latency is not the primary metric to optimize, the `blocking` concurrency model offers the most efficient and effective solution providing a good balance between absolute performance and power consumption.

Given the above considerations, we concentrate our study on the `blocking` and `nonblocking` concurrency control strategies that represent the two best solution for low and high data rate respectively.

4.2 | Applications

In this section we consider two streaming applications: the *Protocol Identification* and *Malware Detection*. In the experiments we used the automatic algorithm proposed in Sec. 3 to optimize the performance/power consumption ration of the applications considered by setting the following requirement: for each +1% increase in the throughput we are willing to pay an increase of power consumption not greater than 1%.

Since the algorithm activates once every second, and since it takes just few milliseconds to decide which queues must be switched, the overhead of the algorithm is less than 1% both in terms of performance and power consumption.

To compute L^{b}_{push}, L^{nb}_{push}, L^{b}_{pop} and L^{nb}_{pop} needed to decide when to switch from `nonblocking` to `blocking` mode, we run a micro-benchmark composed by a producer-consumer pair of nodes (i.e. a simple 2-stage FastFlow pipeline), considering the average latency over 200 thousands messages exchanged when the queue between the producer and the consumer is empty, which represents a worst case scenario for the latency. On the target architecture we obtained the following average values: $L^{b}_{push} = 27usec$, $L^{nb}_{push} = 0.4usec$, $L^{b}_{pop} = 0.8usec$ and $L^{nb}_{pop} = 0.01usec$.

These values include the cost of the `push` and `pop` operations plus the cost introduced by the FastFlow runtime for the message handling.

4.2.1 | Protocol identification application

The first application we use for validating our algorithm is a network monitoring application (36).

This application is implemented as a three stage pipeline. The source node receives the network packets and assigns to each of them a key, such that packets belonging to the same "*application flow*" have the same key. The packets are then forwarded to the second stage of the pipeline through the message queue. For each packet, the second stage stores packet information into a hash table by using the key. This information is used to correlate packets belonging to the same "*application flow*" in order to detect the application protocol (e.g., HTTP). If the node receives a packet belonging to a flow whose protocol has been already identified, no additional processing is performed and the packet is simply forwarded to the third stage.

This behavior creates a situation where for each logical "*application flow*" we have a high latency on the first packets but then, after the protocol has been identified, the latency drops down to almost zero. This is a typical scenario in many data stream processing applications where the computation is triggered only upon receiving a given number of input tuples (8).

Eventually, the second stage will forward each packet to the third node, which injects the packets again into the network. To analyze the application in a realistic environment, we sent the packets to the application at variable rates, equal to those characterizing a modern *Internet Service Provider*. For this purpose, we used the dataset available at http://bit.ly/1RY7fEt.

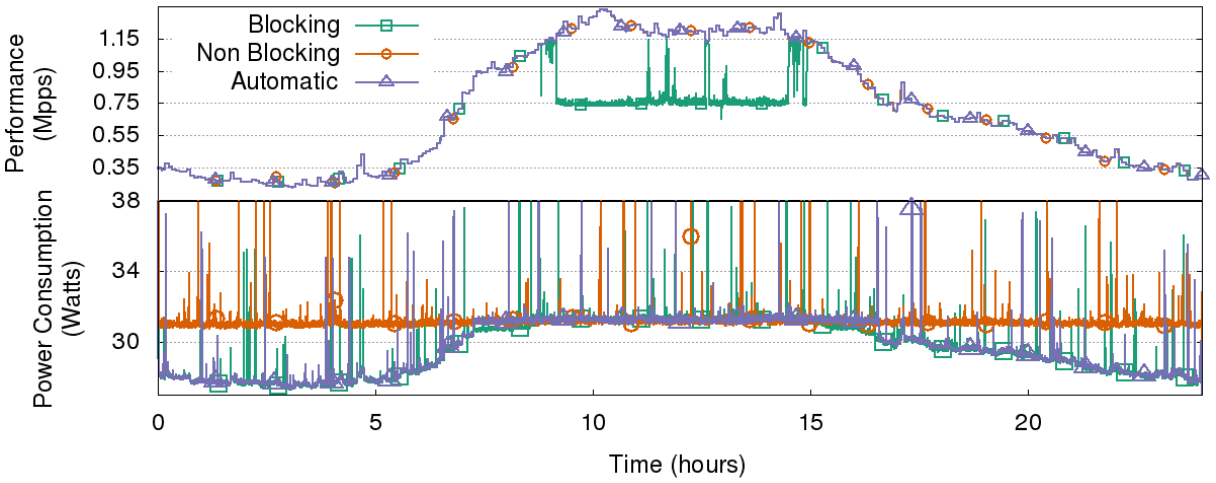

FIGURE 11 Performance and power consumption comparison of `blocking`, `nonblocking` and `automatic` concurrency control strategies for the Protocol identification application.

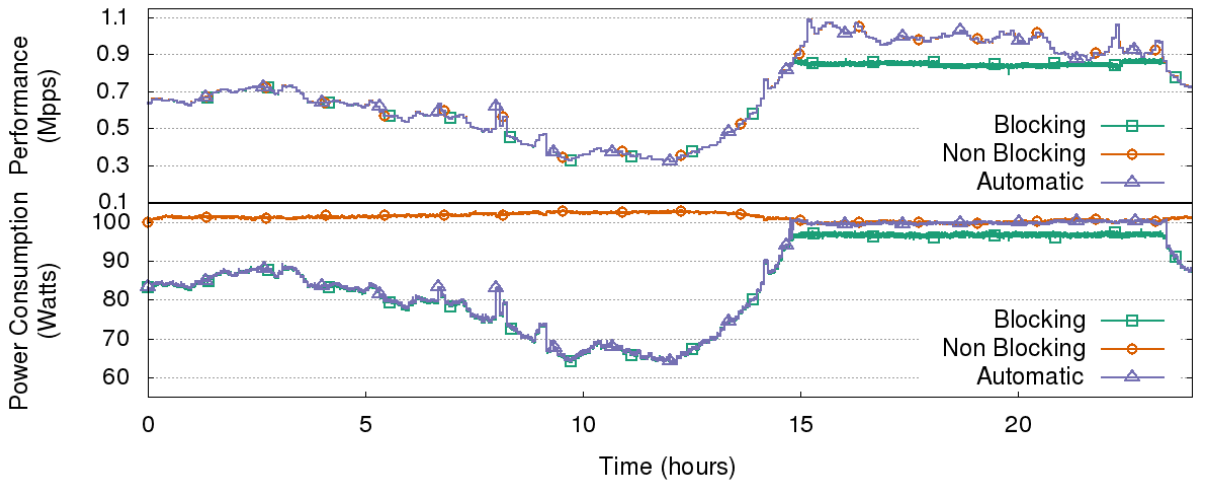

FIGURE 12 Performance and power consumption comparison of `blocking`, `nonblocking` and `automatic` concurrency control strategies for the Malware detection application.

We ran the application for 24 hours and the results are shown in Fig. 11 (the performance is measured using packets-per-second – pps). The spikes in the plot of the power consumption can be explained considering that the application uses only three threads pinned on three CPU's cores. During the application execution, the OS executes on the other CPU's cores others threads/processes (i.e. demons, services, etc.) which temporarily increase the power consumption of the CPU. The algorithm we are proposing (`automatic`) is able to provide, at every time, the best performance and the lowest power consumption among the three concurrency modes. When there is no need to improve the throughput (because there is not enough data to process), the algorithm switches the queues to `blocking` concurrency mode, thus reducing the power consumption. However, when the input arrival rate increases, it switches the queues to `nonblocking`, thus improving the performance to be able to sustain the input data rate. For this application, the `automatic` algorithm leads to a maximum performance improvement of 33.28% with respect to the `blocking` case and to a maximum power reduction of 11% with respect to the `nonblocking` case.

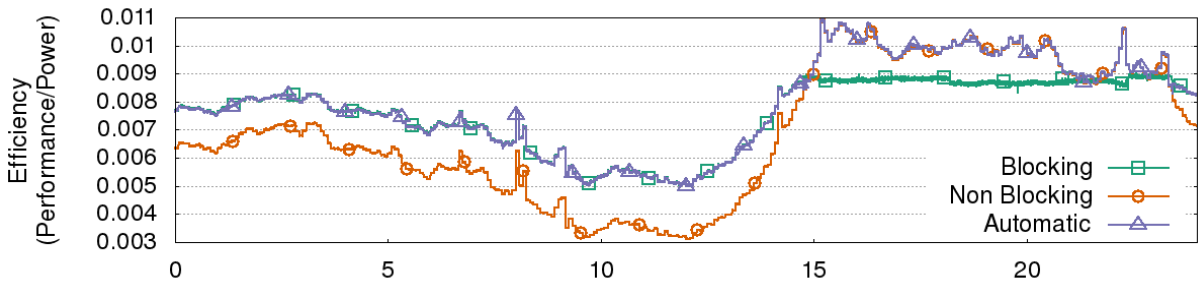

FIGURE 13 Comparison of efficiency of `blocking`, `nonblocking` and `automatic` concurrency control strategies on the Malware detection application.

4.2.2 | Malware detection application

This application is described in (36). Logically, the Malware Detection Application is structured as a 3-stage pipeline where the middle stage computes the most expensive part and can be conveniently replicated a number of times. In FastFlow, this network can be easily and efficiently implemented by using a single task-farm pattern with a custom scheduling policy.

Each worker of the task-farm, after having identified the protocol, searches for a predefined set of "signatures" (representing malware binaries) inside each HTTP packet. The packets are scheduled to a specific worker according to the value of the key computed by the first logical stage of the pipeline that is implemented by the task-farm emitter.

In this experiment, the application graph is composed by 24 nodes (one for each core of the machine). As in the previous test, the arrival rate of the packets to the application is variable. In our test, we used the rate that characterize a modern *Internet Exchange Point* network[3]. For the malware detection part, we used a subset of the database used by the *ClamAV* antivirus[4], containing 2000 signatures.

The results of our test are sketched in Fig. 12 , showing that the `automatic` policy is able to achieve the maximum performance while having the optimal power consumption (i.e. the same values obtained by the `blocking` concurrency mode). Between 15 and 22 the `blocking` concurrency mode has a lower power consumption but it cannot sustain the same arrival rate of the `nonblocking` mode.

In Fig. 13 we show another interpretation of the result, by plotting the efficiency of the different concurrency control techniques, expressed as the ratio between the performance and the power consumption. As we can see from the plot, the `automatic` strategy is always characterized by the highest efficiency between those of the other two techniques.

4.2.3 | Corner Cases and Summary of Results

In this section, we study the applicability of our algorithm considering some corner case scenarios. Analyzing again the Malware Detection application described in the previous section, we studied its behavior when the database of signatures is more extensive than the one considered in the previous tests. This means that the generic worker executes more work for each input packet so its service time increases. By increasing the time spent computing the single input element (i.e. L_{proc} in Fig. 2), the relative impact of L_{pop} and L_{push} decreases, thus reducing the benefit of the `nonblocking` strategy over the `blocking` one. In a nutshell, for application characterized by a high L_{proc}, the `blocking` strategy is as much performing as the `nonblocking` one.

While in the previous section (results shown in Fig. 12), we considered a database of 2000 signatures, here we consider other two cases: a bigger database of 45000 signatures and a very large database of 90000 signatures. The results for the two cases tested are reported in Fig. 14 in the top and bottom plot, respectively.

Moving to a larger database of signature, the performance gap between `blocking` and `nonblocking` strategies get closer as shown in the top plot of Fig. 14 . As expected, the number of packets-per-seconds the system can sustain over time decreases because of the increased service time of each worker. When the largest database of signatures is considered, the two concurrency control policy provide almost the same performance (see the bottom plot of Fig. 14), whereas, in terms of the ratio between performance and power consumption, the `automatic` policy still provides the best efficiency.

To better quantify the relation between service time and maximum throughput of the `blocking` and `nonblocking` concurrency control strategies for the Malware Detection application, we reported the measured values in Table 1 . Particularly, the table shows the average worker service time

[3]https://stats.linx.net/, (IXMANCHESTER), 24 hours data between 02/01/2016 and 03/01/2016. We scaled it down by a 3x multiplicative factor to match the maximum performance achievable on our target architecture.

[4]https://www.clamav.net/.

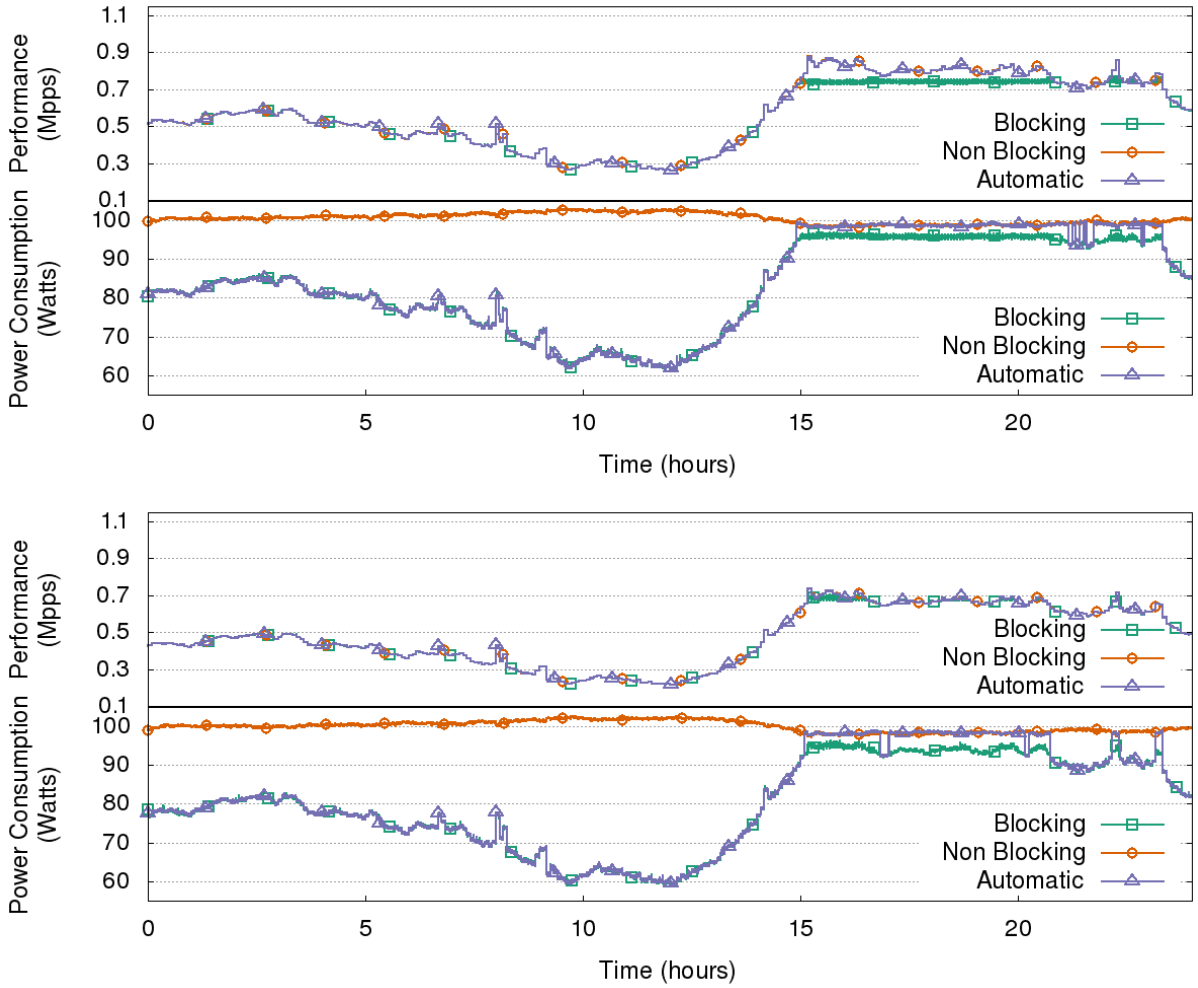

FIGURE 14 Performance and power consumption comparison of `blocking`, `nonblocking` and `automatic` concurrency control strategies for the Malware detection application when the database of signature is of 45000 (top plot) and 90000 (bottom plot) signatures, respectively.

(microseconds) for the entire execution and the maximum number of packets-per-second the system can sustain. The lower the worker service time, the more significant the performance gap between the two strategies. This confirms that the `nonblocking` strategy introduces lower overhead than the `blocking` one. From our tests, for service time higher than 26 microseconds, the `nonblocking` policy starts providing only marginal benefit if none at all. Therefore there is a clear threshold in the node's service time that delimits the point below which the `automatic` algorithm provides some benefits and it is worth to be used.

TABLE 1 Performance gap between `blocking` and `nonblocking` strategies considering different worker's service time for the Malware detection application.

Worker's service time (us)	Blocking vs Non Blocking throughput difference (%)
19	24.23%
23	15.3%
26	5.15%

TABLE 2 Impact of using thread over-provisioning with respect to the available number of cores for the video filtering application. The target machine has 24 cores and 48 hyperthreaded core contexts.

	Max Throughput (pps)	
num threads	Blocking	Non Blocking
24	646720	714291
48	708459	586084

Finally, we considered a new simple video filtering application that can be logically modeled as a three-stage pipeline. The peculiarity of this new application with respect to previous ones is that it always operates at maximum throughput. In fact, the input video is loaded into main memory by the first stage of the pipeline and then individual frames are sent to the second stage at maximum speed. Each frame can be computed in parallel by a set of workers and then the filtered frames are re-organized to preserve video frames order by the third stage, which also stores them in the output file. This application uses the OpenCV library [5] to manipulate frames and to apply an image filter (from RGB to Gray) to each frame of the video (the resolution of the video is 64×64). We used this video filtering application to study the behavior of the `blocking` and `nonblocking` strategies when thread over-provisioning is considered. The machine used for the test has 24 physical and 48 logical cores (featuring Intel's Hyper-Threading technology). The results are shown in Table 2 . While the `blocking` strategy benefits from an increased number of threads-per-physical-core, the `nonblocking` strategy decreases its performance when more than one thread-per-core is used. This is due to the higher resource contention introduced by the busy-waiting policy employed by the `nonblocking` strategy. Therefore, to avoid inconsistency, the `automatic` algorithm is automatically disabled when the number of threads is higher than the number of physical cores.

To conclude and summarize the results, the `nonblocking` strategy is the most performing one and the most power-hungry. Its power consumption depends on the number of cores used and not on the input data rate. The `blocking` strategy is the most power-efficient for low to medium input rates and when more threads than physical cores are used. However, its associated overhead does not always allow to reach the maximum throughput, particularly for very fine-grained computation. In our tests the limit has been identified considering node's service time in the range 25-30us. When applicable, the `automatic` strategy proposed in this work represents a good trade-off between absolute performance and power consumption. It allows exploiting both benefits of the `blocking` and `nonblocking` strategies leading to optimal values of the power/performance ratio.

5 | RELATED WORK

Concurrent programming requires primitives to synchronize the execution of concurrent threads accessing shared data structures. The standard approach is based on a basic *pessimistic* assumption that each time a thread tries to access a critical section this access may potentially be in conflict with other running threads that simultaneously do the same. This problem has been traditionally solved by using *locking* primitives that protect the critical section from concurrent accesses. In turn, the implementations of the lock data structures adopt hardware-level instructions to protect their use, like test-and-set instructions on x86-based machines. In addition, locking mechanisms can adopt busy waiting phases, as in spin-lock implementations, and also passive waiting as did by POSIX mutexes.

A different approach is based on an *optimistic* assumption that concurrent accesses to the share data structures are infrequent, and always paying the overhead of locking is something that can be avoided in the real executions. An approach to that is to exploit the so-called *Transactional Memories* (TM) (37, 38), which provide generic mechanisms for optimistic concurrency. TM can be used to designate arbitrary regions of code making them appear as executed atomically. If no concurrent access is executed during the transaction (a sequence of read/write operations on a memory area), the transaction can be committed and its effects will be permanent on the data structure. Otherwise, the transaction is aborted as it was never executed (i.e. by rolling back the state at the time instant before the transaction begun). As described in (39), TM has the potential of reducing power/energy consumption in real workloads, however hardware supports to TM are not present in every machine although the trend is to extend such support in most of the affordable computing platforms available nowadays. *Speculative Lock Elision* (40) and *Transactional Lock Removal* (41) are two techniques proposed for optimistically executing program's lock regions using transactional memories.

Another specialized form of optimistic concurrency is *lock-free concurrency*, used to implement efficient concurrent data structures. Lock-free algorithm implementations aim at overcoming the various problems associated with the use of locks. Various progress conditions such as *wait-freedom*, *lock-freedom*, and *obstruction-freedom* have been deeply studied and proposed in the literature (9). In (5) the authors analyzed some lock-free and lock-based concurrent data structures (i.e. FIFO queues, double-ended queues and sorted linked lists) by using hardware performance counters finding that lock-free algorithms tend to perform better in general that their lock-based counterparts. Another research direction of interest consists in using specific architectural features to improve the performance of locks. This idea has been followed in (42, 43), where the authors have proposed to use specific on-chip interconnection networks of some chip multiprocessors (i.e. Tilera ones and Netlogic/Broadcom chips) in order to exchange the lock ownership efficiently by exchanging messages over such networks. Although interesting, such solutions have a limited applicability since they target specialized architectures only.

[5] OpenCV library: http://opencv.org

Quite a few authors have previously combined optimistic and pessimistic concurrency control mechanisms in the context of Database Systems. Authors in (44) combined locking with Software TM (STM) in different parts of the program, obtaining better performance than just using a lock-based or an STM-based solution. They found that by using one of the two choices exclusively throughout the application execution is often sub-optimal. Moreover, they formalized a theory for correctly composing different concurrency control protocols into a single program. Adaptive run-time techniques developed for selecting between TM and locking for each transaction are described in (6). These papers represented an inspiration for our work. In fact the dichotomy between different concurrency modes can be breaken by designing an adaptation mechanism that: *i)* identifies the features of the current application phase (e.g., of its workload); *ii)* dynamically selects which concurrency mode to use and applies the chosen one without blocking the application; *iii)* iterates the points *i)* and *ii)* by continuously monitoring the application execution.

The trade-off of the *spin-then-sleep* technique has been studied in (45) where it is shown that simply spinning or sleeping is sub-optimal in many cases. Current implementation of mutexes in the Linux OS uses this technique. The mutex call spins for up to a few hundred cycles before employing a costly `futex` call for suspending the caller. Therefore, it can happen that the threads pay the cost of the futex call only to be immediately woken up, thus wasting both time and energy because the core where the thread is running is not immediately put in a low-power state.

A different approach for reducing power consumption of `nonblocking` algorithms is to use the same techniques used to reduce contention in spin locks, e.g., *linear* and *exponential backoff* (11). Instead of continuously retry in checking the given condition, the thread is put to sleep between two retries for an amount of time that increases linearly or exponentially until a maximum value. While from one side sleeping reduces power consumption because the core where the thread is running may enter a low-power state, this approach has the disadvantage that the threads may be not reactive enough since they might be backed off too far while there is some data ready to be processed. Finding good values for the minimum and the maximum sleeping time is not straightforward and may depend on the target application and on the given input rate for streaming applications. Adaptive backoff mechanisms have been developed for many years, and they target specific execution scenarios by typically performing a statistical analysis of the accesses and their duration. A recent example of this approach is described in (46), where Queueing Theory has been used to dynamically regulate the parameters of a general backoff strategy that can be applied to most of the lock-free data structures.

Although backoff strategies can achieve optimal trade-offs between performance and power consumption in theory, they are hard to be tuned. In our work we did not follow this approach but instead we proposed a switching scheme supported by proper mechanisms and strategies to dynamically choose the best concurrency mode that does not impair performance and power consumption too much. Our idea has been tuned and designed for data stream processing, which is an emerging topic in Big Data analytics.

6 | CONCLUSIONS AND FUTURE WORK

In this work we described an algorithm for the automatic selection of the optimal concurrency control mode for message queues implementing data streaming channels. We validated our proposal using two benchmarks and two real-world data streaming applications, showing that our solution properly adapts the concurrency mode according to the current input data rate of the application, keeping up with the maximum throughput without wasting power. As a future work, we will consider the optimization of the latency as well as the introduction of performance and power consumption prediction techniques to reduce the need of the rollback phase in the algorithm. Moreover, we would like to compare this concurrency control algorithm with other techniques like *concurrency throttling* and *DVFS*.

References

[1] Feitelson Dror G.. *Workload Modeling for Computer Systems Performance Evaluation*. New York, NY, USA: Cambridge University Press; 1st ed.2015.

[2] Esmaeilzadeh Hadi, Blem Emily, St. Amant Renee, Sankaralingam Karthikeyan, Burger Doug. Dark Silicon and the End of Multicore Scaling. *SIGARCH Comput. Archit. News.* 2011;39(3):365–376.

[3] De Matteis Tiziano, Mencagli Gabriele. Keep Calm and React with Foresight: Strategies for Low-Latency and Energy-Efficient Elastic Data Stream Processing. In: :13:1–13:12; 2016.

[4] Han J., Orshansky M.. Approximate computing: An emerging paradigm for energy-efficient design. In: :1-6; 2013.

[5] Hunt N., Sandhu P. S., Ceze L.. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures. In: :63-70; 2011.

[6] Usui Takayuki, Behrends Reimer, Evans Jacob, Smaragdakis Yannis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In: :3–14IEEE Computer Society; 2009; Washington, DC, USA.

[7] Falsafi Babak, Guerraoui Rachid, Picorel Javier, Trigonakis Vasileios. Unlocking Energy. In: :393–406USENIX Association; 2016; Denver, CO.

[8] Andrade Henrique, Gedik BuÄ§ra, Turaga Deepak. *Fundamentals of Stream Processing.* Cambridge University Press; 2014. Cambridge Books.

[9] Moir Mark, Shavit Nir. Concurrent Data Structures. In: Computer and Information Science Series. Chapman & Hall/CRC 2004.

[10] Morari A., Gioiosa R., Wisniewski R. W., Cazorla F. J., Valero M.. A Quantitative Analysis of OS Noise. In: :852-863; 2011.

[11] Agarwal A., Cherian M.. Adaptive Backoff Synchronization Techniques. *SIGARCH Comput. Archit. News.* 1989;17(3):396–406.

[12] Stephens Robert. A survey of stream processing. *Acta Informatica.* 1997;34(7):491–541.

[13] Aldinucci Marco, Danelutto Marco, Kilpatrick Peter, Meneghin Massimiliano, Torquati Massimo. Accelerating code on multi-cores with FastFlow. In: Jeannot E., Namyst R., Roman J., eds. *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, LNCS, vol. 6853: :170-181Springer; 2011.

[14] Leibiusky Jonathan, Eisbruch Gabriel, Simonassi Dario. *Getting Started with Storm.* O'Reilly Media, Inc.; 2012.

[15] Thies William, Karczmarek Michal, Amarasinghe Saman. StreamIt: A Language for Streaming Applications:179–196. Springer 2002.

[16] Danelutto Marco, Torquati Massimo. Structured Parallel Programming with "core" FastFlow. In: LNCS, vol. 8606: Springer 2015 (pp. 29-75).

[17] Biem Alain, Bouillet Eric, Feng Hanhua, et al. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In: SIGMOD '10:1093–1104ACM; 2010; New York, NY, USA.

[18] Mattson Timothy, Sanders Beverly, Massingill Berna. *Patterns for parallel programming.* Addison-Wesley Professional; 2004.

[19] De Matteis Tiziano, Mencagli Gabriele. Parallel Patterns for Window-based Stateful Operators on Data Streams: an Algorithmic Skeleton Approach. *Inter. Journal of Parallel Programming.* 2016;.

[20] Gedik Bugra, Schneider Scott, Hirzel Martin, Wu Kun-Lung. Elastic Scaling for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.*. 2014;25(6):1447–1463.

[21] Gulisano Vincenzo, Jimenez-Peris Ricardo, Patino-Martinez Marta, Soriente Claudio, Valduriez Patrick. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Trans. Parallel Distrib. Syst.*. 2012;23(12):2351–2365.

[22] Danelutto Marco, Kilpatrick Peter, Mencagli Gabriele, Torquati Massimo. State access patterns in stream parallel computations. *The International Journal of High Performance Computing Applications.* 0;0(0):1094342017694134.

[23] Liu X., Harwood A., Karunasekera S., Rubinstein B., Buyya R.. E-Storm: Replication-Based State Management in Distributed Stream Processing Systems. In: :571-580; 2017.

[24] Mencagli Gabriele, Vanneschi Marco, Vespa Emanuele. A Cooperative Predictive Control Approach to Improve the Reconfiguration Stability of Adaptive Distributed Parallel Applications. *ACM Trans. Auton. Adapt. Syst.*. 2014;9(1):2:1–2:27.

[25] Gedik B., Özsema H.G., Öztürk Ö.. Pipelined Fission for Stream Programs with Dynamic Selectivity and Partitioned State. *J. Parallel Distrib. Comput.*. 2016;96(C):106–120.

[26] Prekas George, Primorac Mia, Belay Adam, Kozyrakis Christos, Bugnion Edouard. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In: SoCC '15:342–355ACM; 2015; New York, NY, USA.

[27] Danelutto Marco, De Matteis Tiziano, De Sensi Daniele, Torquati Massimo. Evaluating Concurrency Throttling and Thread Packing on SMT Multicores. In: :219-223; 2017.

[28] De Sensi Daniele, Torquati Massimo, Danelutto Marco. A Reconfiguration Algorithm for Power-Aware Parallel Applicatios. *ACM TACO*. 2016;13(4):43:1–43:25.

[29] Carney Don, Çetintemel Uğur, Rasin Alex, Zdonik Stan, Cherniack Mitch, Stonebraker Mike. Operator Scheduling in a Data Stream Manager. In: VLDB '03:838–849VLDB Endowment; 2003.

[30] Mencagli Gabriele, Torquati Massimo, Danelutto Marco. Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Generation Computer Systems.* 2017;.

[31] Gulisano Vincenzo, Nikolakopoulos Yiannis, Papatriantafilou Marina, Tsigas Philippas. Data-Streaming and Concurrent Data-Object Co-design: Overview and Algorithmic Challenges:242–260. Cham: Springer International Publishing 2015.

[32] Cederman Daniel, Gulisano Vincenzo, Nikolakopoulos Yiannis, Papatriantafilou Marina, Tsigas Philippas. Brief Announcement: Concurrent Data Structures for Efficient Streaming Aggregation. In: SPAA '14:76–78ACM; 2014; New York, NY, USA.

[33] Giacomoni John, Moseley Tipp, Vachharajani Manish. FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue. In: :43–52ACM; 2008.

[34] Aldinucci Marco, Danelutto Marco, Kilpatrick Peter, Meneghin Massimiliano, Torquati Massimo. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In: LNCS, vol. 7484: :662-673Springer; 2012.

[35] De Sensi Daniele, Torquati Massimo, Danelutto Marco. Mammut: High-level management of system knobs and sensors. *SoftwareX*. 2017;6:150 - 154.

[36] Danelutto Marco, Deri Luca, De Sensi Daniele, Torquati Massimo. Deep Packet Inspection on Commodity Hardware using FastFlow. In: :92 – 99IOS Press; 2013.

[37] Herlihy Maurice, Moss J. Eliot B.. Transactional Memory: Architectural Support for Lock-free Data Structures. In: :289–300ACM; 1993; New York, NY, USA.

[38] Herlihy M.. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Trans. Database Syst.*. 1990;15(1):96–124.

[39] Rughetti D., Sanzo P. D., Pellegrini A.. Adaptive Transactional Memories: Performance and Energy Consumption Tradeoffs. In: :105-112; 2014.

[40] Rajwar Ravi, Goodman James R.. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In: :294–305IEEE Computer Society; 2001; Washington, DC, USA.

[41] Rajwar Ravi, Goodman James R.. Transactional Lock-free Execution of Lock-based Programs. *SIGOPS Oper. Syst. Rev.*. 2002;36(5):5–17.

[42] Buono D., Mencagli G.. Run-time mechanisms for fine-grained parallelism on network processors: The TILEPro64 experience. In: :55-64; 2014.

[43] De Matteis Tiziano, Luporini Fabio, Mencagli Gabriele, Vanneschi Marco. Evaluation of Architectural Supports for Fine-Grained Synchronization Mechanisms. In: Iasted; 2013; Innsbruck, Austria.

[44] Ziv Ofri, Aiken Alex, Golan-Gueta Guy, Ramalingam G., Sagiv Mooly. Composing Concurrency Control. In: :240–249ACM; 2015; New York, NY, USA.

[45] Boguslavsky L., Harzallah K., Kreinen A., Sevcik K., Vainshtein A.. Optimal Strategies for Spinning and Blocking. *JPDC*. 1994;21(2):246 - 254.

[46] Atalar Aras, Renaud-Goud Paul, Tsigas Philippas. How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses. In: Fatourou Panagiota, Jiménez Ernesto, Pedone Fernando, eds. *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, Leibniz International Proceedings in Informatics (LIPIcs), vol. 70: :23:1–23:17Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik; 2017; Dagstuhl, Germany.

AUTHOR BIOGRAPHY

Massimo Torquati is an Assistant Professor at the Computer Science Department of the University of Pisa in Italy. He has published more than 80 peer-reviewed papers in conference proceedings and journals, mostly in the fields of parallel and distributed programming and runtime systems for HPC. His current research interests are on pattern-based parallel programming models, high-performance data stream processing, concurrent lock-free data structures and autonomic management in parallel systems. Currently, he is the main developer of the FastFlow parallel programming framework.

Daniele De Sensi is a last year PhD candidate at the Computer Science Department of the University of Pisa, Italy. His doctoral work is focused on autonomic and power-aware runtime solutions for parallel applications. He designed algorithms to enforce power consumption and performance requirements on parallel applications through dynamic reconfigurations by exploiting online learning techniques. He is also interested in parallel programming models and network processing applications.

Gabriele Mencagli is an Assistant Professor at the Computer Science Department of the University of Pisa, Italy. He is co-author of more than 45 peer-reviewed papers appeared in international conferences, workshops and journals, and of one book. He is member of the Parallel Programming Models group at the same university. His research interests are in the area of parallel and distributed systems and data stream processing. He is also interested in studying interdisciplinary approaches for autonomic data stream processing applications and frameworks.

Marco Aldinucci Marco Aldinucci is an associate professor at Computer Science Department of the University of Torino since 2014. He is the author of over a hundred papers in international journals and conference proceeding. He has been participating in over 20 national and international research projects concerning parallel and autonomic computing. He is the recipient of the HPC Advisory Council University Award 2011, the NVidia Research award 2013, the IBM Faculty Award 2015. He is the P.I. of the parallel computing group alpha@UNITO, the director of the "HPC and smart data" laboratory at ICxT@UNITO innovation centre, and vice-president of the C3S@UNITO competency centre. His research is focused on parallel and distributed computing.

Marco Danelutto Marco Danelutto is a Full Professor at the Computer Science Department of the University of Pisa, Italy. His main research interests are in the field of parallel programming models, in particular in the area of parallel design patterns and algorithmic skeletons. He is author of more that 150 papers appearing in refereed international journals and conferences. He is the group leader of the Parallel Programming Model group at the same university, involved in a number of national and EU funded projects (CoreGRID, GRIDcomp, ParaPhrase, REPARA, RePhrase).