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

Towards Power-Aware Data Pipelining on Multicores

Marco Aldinucci · Marco Danelutto · Daniele De Sensi · Gabriele Mencagli · Massimo Torquati

Received: date / Accepted: date

Abstract Power consumption management has become a major concern in software development. Continuous streaming computations are usually composed by 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 for both performance and power consumption. In this paper, we describe the design of an adaptive concurrency control algorithm for implementing power-efficient communications on shared memory multicores. The algorithm provides the throughput offered by a `nonblocking` implementation and the power efficiency of a `blocking` protocol. We demonstrate that our algorithm reduces the power consumption of data streaming computations without decreasing their throughput.

1 Introduction

In the realm of parallel and distributed computing, `throughput` and `latency` have been traditionally considered the primary metrics to evaluate computing systems. However, in recent years, power consumption has gained more and more significance up to the point it reached the same importance of traditional metrics [10]. As a consequence, system optimisation is played against multiple concerns: performance and power consumption, at least.

The optimisation of the performance-power trade-off has been mainly pursued by means of *dynamic* reconfigurations of the system at different abstrac-

Marco Danelutto, Daniele De Sensi, Gabriele Mencagli, Massimo Torquati
University of Pisa, Italy
Computer Science Department
E-mail: {marcod, desensi, mencagli, torquati}@di.unipi.it

Marco Aldinucci
University of Turin, Italy
Computer Science Department
E-mail: aldinuc@di.unito.it

tion levels, from hardware level up to synchronisation mechanisms to algorithms. Their principal goal is reducing power consumption with limited impact on performance. Examples are Dynamic Voltage and Frequency Scaling (DVFS), approximate computing [13], energy-efficient data structures [16] and adaptive locking [24].

A recent work has demonstrated that a simple and effective energy saving technique consists in optimising the synchronisation mechanisms [11]. In a shared-memory system, a standard approach to synchronise producer/consumer interactions between pairs of threads uses a concurrent FIFO queue that supports `push` and `pop` operations in an atomic and efficient way. This approach is particularly relevant in Data Streaming computations where the application is modelled as a graph of modules communicating through channels implemented as concurrent queues [4].

In the lock-based concurrent queues, if the thread that currently holds the lock is delayed then all other threads attempting to access the data structure are delayed too. This phenomenon is called `blocking`. In blocking algorithms the waiting of a thread, is often implemented by suspending it, i.e., by putting the thread to *sleep*. In this case, the suspended thread is moved in a waiting queue and the hardware context is released to the OS. Suspended threads do not directly consume power. However, suspension and restart mechanisms impair application performance due to many factors such as waiting time in the ready queue, context switch, compulsory cache miss or core migration [11].

Concurrent queues can also be implemented in a efficient and scalable way by using `nonblocking` algorithms. The term `nonblocking` refers to all progress conditions requiring that the failure or indefinite delay of a thread cannot prevent other threads from making progress [19]. By definition, these algorithms cannot use locks, therefore `nonblocking` algorithms use spinning for implementing the waiting of a thread.

While `nonblocking` algorithms are mainly chosen for their progress guarantees (e.g., *wait-* and *lock-freedom*), they are also employed for their higher throughput and lower latency [19] [16]. Unfortunately, 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), which keeps the CPU core active doing nothing useful. A common approach for reducing this shortcoming is to delay retries with micro-sleep according to a fixed or variable duration. The very same technique is used to reduce contention in spinlock algorithms by way of *exponential backoff* [1]. Methodologically, the key issue is that these techniques always require some tuning which is application-dependent, platform-dependent and time-consuming. Using wrong tuning parameters would impair the reactivity of the technique, due to a wrong micro-sleep duration.

In continuous streaming computations, the optimal concurrency control mechanisms may change during time, due to variable arrival rates and fluctuations in the workload. In such scenario, selecting the proper concurrency control strategy is a critical aspect. Let us consider a network streaming com-

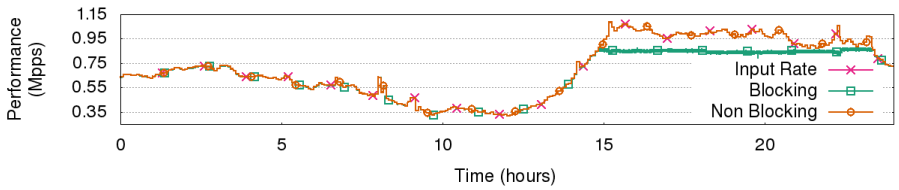

Fig. 1 Motivating example: throughput of a streaming application over time under variable input rate with blocking and nonblocking concurrency control mode.

putation where each received data packet needs to be analysed. The typical performance scenario over time is sketched in Fig. 1.

The application has been run two times on the same input dataset and by sending data at a rate equal to the one of a real network. The first time, by statically setting a `blocking` concurrency mode and the second time with a `nonblocking` concurrency mode. Interestingly, between 15 and 23, only the `nonblocking` protocol is able to sustain the input arrival rate, on the other hand, from 0 to 15 the `blocking` protocol is capable to sustain the input bandwidth by consuming less power. In fact, the power consumption is proportional to the sustained bandwidth for the `blocking` mode, while it is a constant amount (about 100 Watts), regardless of the input arrival rate, for the `nonblocking` mode. Indeed, for low input rate (e.g., from 0 to 14), each thread is waiting to receive new data from its input `nonblocking` queue, spending a significant amount of time in a busy-waiting loop, increasing the overall power consumption of the entire system. On the other hand, during high input rate phases, the system needs to be very reactive to sustain the increased input bandwidth and the maximum throughput can be achieved only if a low-latency `nonblocking` algorithm for accessing the communication queues is used.

`Blocking` *vs* `nonblocking` approach has always been considered mutually exclusive alternatives. The contribution of this work is to break this dichotomy, by proposing a new implementation strategy for parallel pipelines used in continuous stream processing computations. We combine the power efficiency of `blocking` implementations with the higher throughput of `nonblocking` approaches. This strategy will then be adapted to manage applications structured as arbitrary graphs. The main objective is to reduce the power consumption of the overall streaming network, whose channels are implemented via lock-free message queues, without compromising the maximum achievable throughput.

The outline of this paper is the following. Sect. 2 provides background concepts. Sect. 3 describes the design of the algorithm. Sect. 4 shows an evaluation using two real-world applications. Sect. 5 describes some related works and eventually, Sect. 6 provides the conclusions and outlines possible future works.

2 Background

In this section we provide the background needed to clearly identify the motivations that led to the implementation of a new communication protocol between threads of a parallel streaming network on multicores.

Stream parallelism. It is a well-known programming model supporting the parallel execution of a stream of data elements by using a series of *sequential* or *parallel* modules [22]. Modules (often called *filters*, *agents* or *operators*) compute in parallel over subsequent or independent data tasks and communicate data via *channels*. There are many applications in which the input streams are primitive, because they are generated by external sources (e.g. HW sensors, networks, etc.) or I/O devices. However, there are cases in which streams are not primitive, but it is possible that they can be generated directly within the program [2].

Streaming computations can be modelled as a graph where each module is a node, and each edge is a communication channel. Data is introduced in the application by one or more *source* nodes and the result are sent out through one or more *sink* nodes. Despite being a common approach, writing a correct, efficient and portable program is particularly difficult and error prone. Fortunately, there are notable examples of parallel patterns targeting stream parallelism, such as *pipeline* and *task-farm*, that allow to abstract the stream parallel computation [18].

Of particular interest are stateful streaming computations for which maintaining the entire stream history is unfeasible. In these cases, a common approach is to use a *sliding-window* buffer, to store only the most recent data [4]. When the window is full, an arbitrary complex function is executed over the elements in the window. However, if the window is not full the user function can not be triggered, and the data is simply stored into the window buffer just paying a low amount of CPU cycles. Since for these elements the computational part is almost negligible with respect to the cost of computing the user function on the entire window buffer, the latency of the communication protocol between two distinct nodes of the streaming application may have a significant impact on the overall application throughput.

Programming model for streaming. Over the years, several programming models and tools have been proposed to simplify the development of streaming applications and to obtain the desired QoS level [23] [17] [7].

In this study, we used the FastFlow parallel programming framework [1], a C++11 template library that allows the programmer to build fast networks of streaming nodes. Each node may have multiple inputs and multiple output channels. Two distinct nodes are connected by a channel that is implemented using either a bounded or unbounded lock-free FIFO queue [3]. FastFlow promotes the use of customisable parallel patterns such as `task-farm`, `pipeline`,

[1] FastFlow website: `http://mc-fastflow.sourceforge.net`

`map/reduce` and `feedback loop` [18]. In the FastFlow model, parallel patterns can be easily combined and nested, while the single network node can be customized for low-level data routing and scheduling.

Adaptivity in data stream processing. In continuous streaming applications, where data is ingested by physical sensors or by software platforms, providing a given QoS in presence of variable arrival rates and sudden workload changes is a critical aspect. The parallel components used to build the streaming application, have to provide auto-tuning capabilities or at least suitable interfaces and mechanisms to the programmer for managing dynamic changes over time, for example to change the number of threads used by the application [8] [9] or the *Concurrency Control* mode used in the synchronisations [24].

Adaptivity is a fundamental feature of parallel components at any abstraction level, though it is often not considered in the design of parallel frameworks. In order to enable adaptivity, the parallel framework has to provide a proper set of dynamic reconfiguration mechanisms and features that can be used by an autonomous manager/controller of the parallel component to fulfil specific application requirements. Examples of such mechanisms are: a) changing the number of threads (*Dynamic Concurrency Throttling*); b) changing the concurrency mechanisms used in the synchronization primitives (*Concurrency Control*); c) changing the mapping of the threads on the cores (*Threads Packing*); d) dynamically scale the clock frequency of the cores (*DVFS*).

Concurrency control, power implications. Current CPUs can use different hardware mechanisms in order to reduce their power consumption. If the core utilisation lower than 100%, the operating system could decide to lower its clock frequency (by changing the so called P-State). However, while doing busy waiting on a `nonblocking` message queue, the core is 100% utilised, and the clock frequency will not be decreased. In addition to that, if the core is idle for a long enough interval, the power control of the CPU can start shutting down some components of the core (entering in the so called C-States), thus further reducing its power consumption. For these reasons, `blocking` queues are more power efficient than `nonblocking` queues.

However, suspending a thread is costly in terms of performance [11] and should be done only when the loss in performance is compliant with the required QoS.

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 optimise 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 [12,3]. 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 [7] as the reference parallel framework for implementing pipelines. In FastFlow pipelines are implemented by composing multiple logical *nodes*, where each of them has one input and one output message queue (nodes can in general have multiple input or multiple output independent message queues). Nodes are de-facto implemented by a POSIX thread whose default concurrency control mode is `nonblocking` for accessing both input and output queues.

We extended the FastFlow concurrency mode by associating to each queue 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);
 if ok then
  if (CCM==blocking and
      C_waiting) then
     signal C_cond_in;
 else
  if (CCM==blocking and
     Q.isFull) then
    wait_on P_cond_out;
  goto 1;
 return success;
```

Algorithm 2: CCPop

```
1:ok=Q.pop(data)
 if ok then
  if (CCM==blocking and
      P_waiting) then
     signal P_cond_out;
 else
  if (CCM==blocking and
     Q.isEmpty) then
    wait_on C_cond_in;
  goto 1;
 return success;
```

To `push` a message into the output queue the runtime first pushes the data pointer into the lock-free SPSC queue (`Q`); if it succeeds, depending on the current concurrency mode of the node (let us call it `CCM`), two different actions are taken. In case of `nonblocking` mode the operation has been successfully completed without any further step. 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. In that case, C will be awakened by an explicit `signal` on its input condition variable (`C_cond_in`). If the `push` on the lock-free queue fails and `CCM=nonblocking`, the push operation is executed again until it will complete with success. If `CCM=blocking`, then the runtime checks if the queue is full and, in that case, the thread is put to sleep on

its output condition variable (`P_cond_out`). If it is not full, the operation is restarted from the beginning (this is a spurious condition that may happen with lock-free data structure).

Let us now consider the `pop` operation. The runtime pops a new data pointer from the lock-free queue. If the operation succeeds and `CCM=nonblocking`, the operation has been successfully completed. 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 (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. If the `pop` fails and `CCM=nonblocking` the operation will be repeated until it completes with success. If `CCM=blocking` then the runtime checks if the queue is empty and puts the thread to sleep on its input condition variable, otherwise the operation is restarted from the beginning.

By atomically 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.

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. 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 withouth any extra synchronisation.

Let us now consider the two possible cases for the dynamic switching of the concurrency control mode: i) from `blocking` to `nonblocking`; ii) from `nonblocking` to `blocking`.

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 both the latencies of `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 sleeping 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 optimise 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. In both cases, since 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.

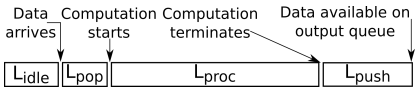

Fig. 2 Different latencies in node operations.

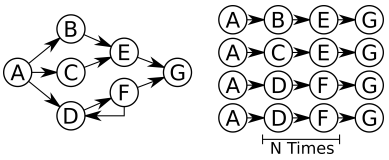

Fig. 3 A data stream processing graph and all its possible paths.

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 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` latencies 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:

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

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

3.3 Generic Streaming Graph

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 subpaths multiple times (*N Times* in Fig. 3) in the pipeline. However, it is sufficient to optimise each node of the path just once, thus the algorithm will still optimise the throughput of the application even if we do not consider these duplicate nodes. For example, two bottom paths in Fig. 3 will actually be the same path for the purpose of the algorithm.

3.3.1 Message Queues' Memory Utilisation.

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 utilisation 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 utilisation. 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 describe the results obtained validating the algorithm proposed in Sec. 3.

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

To measure the power consumption, we used the MAMMUT[2] library, that on this specific architecture relies on RAPL counters. In these experiments we required the algorithm to optimise performance if each +1% increase in the throughput will lead to a power consumption not greater than 1%. The algorithm operates on a monitoring interval of 1 second and by inhibiting the evaluation of a node for 10 seconds in case of a rollback.

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. On the target architecture we obtained the following average values: $L^b_{push} = 11usec$, $L^{nb}_{push} = 0.9usec$, $L^b_{pop} = 0.1usec$ and $L^{nb}_{pop} = 0.01usec$.

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

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

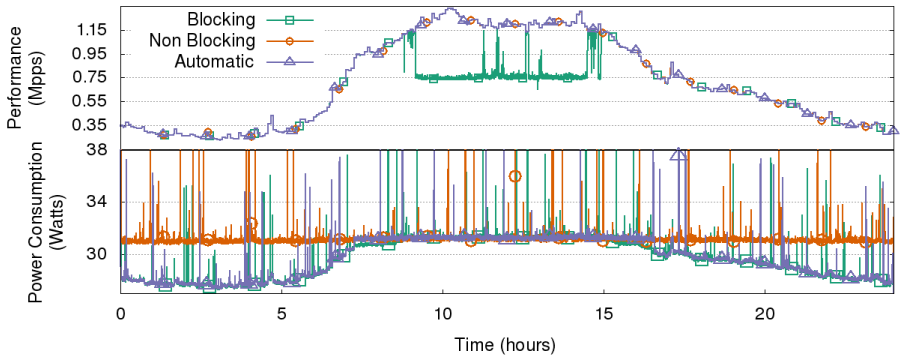

Fig. 4 Comparison of performance and power consumption of `blocking`, `nonblocking` and `Automatic` queues on protocol identification application.

4.0.1 Protocol Identification Application.

The first application we use for validating our algorithm is a network monitoring application [6].

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 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 for which the protocol has been already identified, no additional processing is performed.

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 [4].

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

We ran the application for 24 hours and with the results shown in Fig. 4. 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 some queues to `nonblocking`, thus improving the performance to be able to sustain the input bandwidth. For

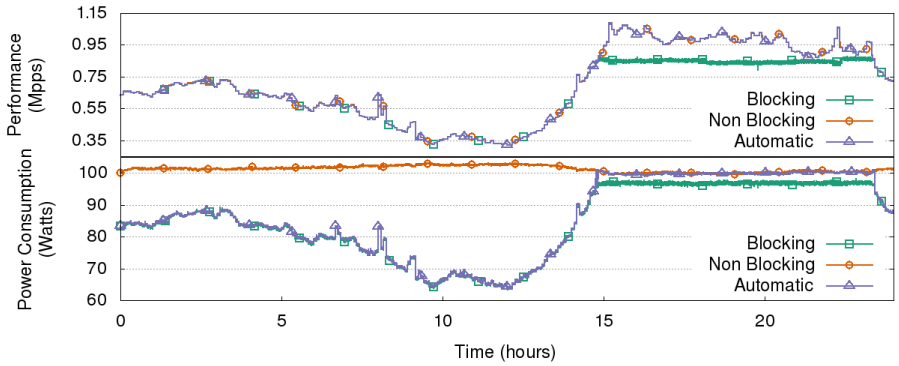

Fig. 5 Comparison of performance and power consumption of `blocking`, `nonblocking` and `Automatic` message queues on threats detection application.

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.

4.0.2 Malware Detection Application.

This application is the one briefly introduced in Sect. 1, and it is deeply described in [6].

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.

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

In this experiment, the application 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 characterise 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.

[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/`.

The results of our test are sketched in Fig. 5, showing that the `Automatic` policy is able to achieve the maximum performance while having the optimal power consumption. 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.

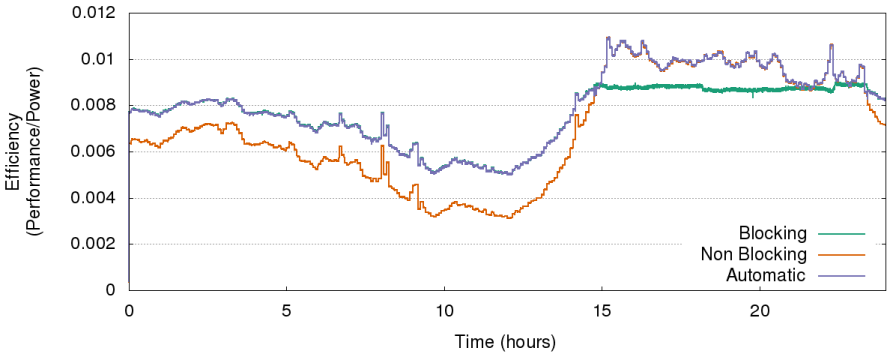

Fig. 6 Comparison of efficiency of blocking, non-blocking and automatic message queues on threats detection application.

In Fig. 6 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 characterised by the highest efficiency between those of the other two techniques.

5 Related Work

Concurrent programming requires primitives such as locks to synchronise threads of execution. An alternative to locking is *optimistic concurrency* [14] where accesses to shared data can proceed concurrently. In case of conflicting accesses, these are dynamically detected and recovered (typically by rolling back the state).

A specialised form of optimistic concurrency is `nonblocking` concurrency used to implement efficient concurrent data structures. Nonblocking algorithm implementations aim to overcome the various problems associated with the use of locks. Various `nonblocking` progress conditions such as *wait-freedom*, *lock-freedom*, and *obstruction-freedom* have been deeply studied and proposed in the literature [19].

Transactional memory (TM) [15], provides a generic mechanism for optimistic concurrency. TM can be used to designate arbitrary regions of code making them appear to execute atomically. Speculative Lock Elision [20] and

Transactional Lock Removal [21] are two techniques proposed for optimistically executing program's lock regions using transactional memory.

Quite a few authors have previously combined optimistic and pessimistic (lock-based) concurrency control mechanisms in the context of Database Systems. Authors in [25] combine 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 find that by using either choice exclusively for the entire application is often suboptimal. Moreover, they formalize a theory for correctly composing different concurrency control protocols into a single program.

Adaptive runtime techniques developed for selecting between TM and locking for every transaction are described in [24].

In [16] the authors analysed 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 be more performing.

The trade-off of the *spin-then-sleep* technique has been studied in [5] where it is shown that simply spinning or sleeping is suboptimal 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 spinlock algorithms, e.g., *exponential backoff* [1]. Instead of continuously retry in checking the given condition, the thread is put to sleep between two retries for an amount of time that increase exponentially until a maximum value. 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. Moreover, finding good values for the minimum and the maximum sleeping time is not straightforward and may depend on the target application.

Differently from previous work, our approach does not try to optimise locks. The proposed algorithm uses POSIX mutexes and condition variables only to implement thread sleeping in a portable way. Threads are suspended only if, for the given input rate, there is no throughput degradation with respect to the `nonblocking` lock-free protocol.

6 Conclusions and Future Work

In this work, we described an algorithm for the automatic selection of optimal concurrency control mode for message queues implementing data streaming channels. We have validated our proposal over two real-world data streaming applications, showing that our solution can adapt the concurrency mode

according to the input data rate of the application, keeping up with the maximum throughput without wasting power. As a future work, we will consider the optimisation 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.

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

References

1. Agarwal, A., Cherian, M.: Adaptive backoff synchronization techniques. SIGARCH Comput. Archit. News **17**(3), 396–406 (1989)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with fastflow. In: E. Jeannot, R. Namyst, J. Roman (eds.) Proc. of 17th Intl. Euro-Par 2011 Parallel Processing, *LNCS*, vol. 6853, pp. 170–181 (2011)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Proc. of 18th Euro-Par Conf., *LNCS*, vol. 7484, pp. 662–673. Springer (2012)
4. Andrade, H., Gedik, B., Turaga, D.: Fundamentals of Stream Processing. Cambridge University Press (2014). Cambridge Books
5. Boguslavsky, L., Harzallah, K., Kreinen, A., Sevcik, K., Vainshtein, A.: Optimal strategies for spinning and blocking. JPDC **21**(2), 246 – 254 (1994)
6. Danelutto, M., Deri, L., De Sensi, D., Torquati, M.: Deep packet inspection on commodity hardware using FastFlow. In: Proc. of 15th Inter. Parallel Computing Conference (ParCo), vol. 25, pp. 92 – 99. IOS Press (2013)
7. Danelutto, M., Torquati, M.: Structured parallel programming with "core" FastFlow. In: Central European Functional Programming School, *LNCS*, vol. 8606, pp. 29–75. Springer (2015)
8. De Matteis, T., Mencagli, G.: Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In: Proc. of the 21st ACM SIGPLAN PPoPP Symposium, pp. 13:1–13:12 (2016)
9. De Sensi, D., Torquati, M., Danelutto, M.: A reconfiguration algorithm for power-aware parallel applicatios. ACM TACO **13**(4), 43:1–43:25 (2016)
10. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. SIGARCH Comput. Archit. News **39**(3), 365–376 (2011)
11. Falsafi, B., Guerraoui, R., Picorel, J., Trigonakis, V.: Unlocking energy. In: USENIX ATC 16), pp. 393–406. USENIX Association, Denver, CO (2016)
12. Giacomoni, J., Moseley, T., Vachharajani, M.: Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In: Proc. of the 13th ACM SIGPLAN PPoPP Symposium, pp. 43–52. ACM (2008)
13. Han, J., Orshansky, M.: Approximate computing: An emerging paradigm for energy-efficient design. In: 2013 18th IEEE ETS Symposium, pp. 1–6 (2013)
14. Herlihy, M.: Apologizing versus asking permission: Optimistic concurrency control for abstract data types. ACM Trans. Database Syst. **15**(1), 96–124 (1990)
15. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proc. of the 20th Annual Intern. Symposium on Computer Architecture (ISCA), pp. 289–300. ACM, New York, NY, USA (1993)
16. Hunt, N., Sandhu, P.S., Ceze, L.: Characterizing the performance and energy efficiency of lock-free data structures. In: 2011 15th Workshop on Interaction between Compilers and Computer Architectures, pp. 63–70 (2011)
17. Leibiusky, J., Eisbruch, G., Simonassi, D.: Getting Started with Storm. O'Reilly Media, Inc. (2012)

18. Mattson, T., Sanders, B., Massingill, B.: Patterns for parallel programming. Addison-Wesley Professional (2004)
19. Moir, M., Shavit, N.: Concurrent data structures. In: Handbook Of Data Structures And Applications, Computer and Information Science Series, chap. 47. Chapman & Hall/CRC (2004)
20. Rajwar, R., Goodman, J.R.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proc. of the 34th ACM/IEEE MICRO Symposium, pp. 294–305. IEEE Computer Society, Washington, DC, USA (2001)
21. Rajwar, R., Goodman, J.R.: Transactional lock-free execution of lock-based programs. SIGOPS Oper. Syst. Rev. **36**(5), 5–17 (2002)
22. Stephens, R.: A survey of stream processing. Acta Informatica **34**(7), 491–541 (1997)
23. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications, pp. 179–196. Springer (2002)
24. Usui, T., Behrends, R., Evans, J., Smaragdakis, Y.: Adaptive locks: Combining transactions and locks for efficient concurrency. In: Proc. of the 2009 18th PACT Conf., pp. 3–14. IEEE Computer Society, Washington, DC, USA (2009)
25. Ziv, O., Aiken, A., Golan-Gueta, G., Ramalingam, G., Sagiv, M.: Composing concurrency control. In: Proc. of the 36th ACM SIGPLAN PLDI Conference, pp. 240–249. ACM, New York, NY, USA (2015)