Reducing Message Latency and CPU Utilization in the CAF Actor Framework

Massimo Torquati*, Tullio Menga†, Tiziano De Matteis*, Daniele De Sensi* and Gabriele Mencagli*
*Department of Computer Science, University of Pisa
Largo B. Pontecorvo 3, I-56127, Pisa, Italy
Email: {torquati, dematteis, desensi, mencagli}@di.unipi.it
†ATS Advanced Technology Solutions S.p.A.
Via Montefeltro 6, 20156 Milano, Italy
Email: tullio.menga@atscom.it

***Abstract*—In this work, we consider the C++ Actor Framework (`CAF`), a recent proposal that revamped the interest in building concurrent and distributed applications using the actor programming model in C++. `CAF` has been optimized for high-throughput computing, whereas message latency between actors is greatly influenced by the message data rate: at low and moderate rates the latency is higher than at high data rates. To this end, we propose a modification of the polling strategies in the work-stealing `CAF` scheduler, which can reduce message latency at low and moderate data rates up to two orders of magnitude without compromising the overall throughput and message latency at maximum pressure. The technique proposed uses a lightweight event notification protocol that is general enough to be used used to optimize the runtime of other frameworks experiencing similar issues.**

Keywords: Actor model, CAF, multi-cores, message latency, work-stealing, polling strategies.

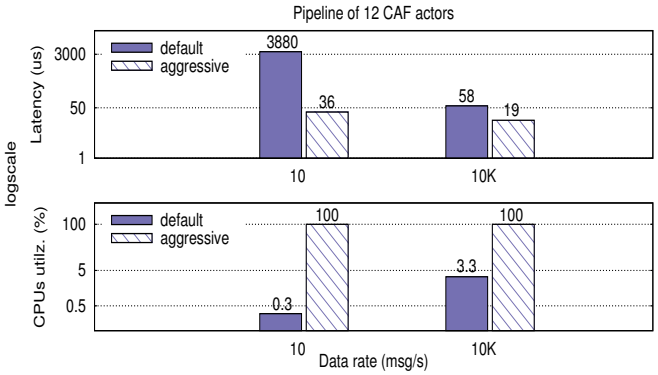

Fig. 1. Motivating example. Top:) Average message latency for a pipeline of 12 `CAF` actors for two constant message rate (10msg/s and 10Kmsg/s) and two `CAF` configurations. Bottom:) Corresponding CPUs' utilization.

I. Introduction

An ever-increasing number of applications requires high performance for serving concurrent tasks. Often, applications need to scale up instantaneously to satisfy high input demands as in modern cloud settings or, they need to meet stringent QoS requirements as in many real-time data streaming computations [1].

Throughput, *message latency* and more recently *power consumption* are considered the primary metrics to evaluate the performance of computing systems [2]. Designing a parallel programming framework that subsumes full scalability, low message latency, high-throughput while, at the same time, minimizes system power consumption is a challenging task that requires using many sophisticated algorithms at different abstraction level.

In this work, we consider the "C++ Actor Framework" `CAF` [3], which is a modern, full-fledged C++ development platform and a powerful runtime system that provides the user with the actor programming model.

The actor model of computation [4], [5] has recently gained significant attention because of its high level of abstraction that makes it appealing for concurrent applications in parallel as well as distributed computing. The model allows a complete separation of the software design from its deployment at runtime making it highly attractive for exploiting the potential parallelism of the modern multi/many-cores platforms.

The `CAF` runtime can offer high-throughput, reliability, scalability, and distribution transparency thanks to its careful design [6]. However, it has not been optimized for providing low end-to-end message latency, particularly in streaming computations where the input rate can be highly variable. To exemplify the problem we experienced, Fig. 1 shows the analysis we carried out on a simple synthetic streaming application composed of 12 `CAF` actors organized in a pipeline topology. The first actor injects messages at a constant rate; the other actors (but the last one) simply forward the message they receive in input to the next actor. We measured the average message latency to move a message from the first to the last actor. The test has been executed considering two different message rates: a low and a high data rate scenarios, i.e. 10msg/s and 10Kmsg/s, respectively. We also considered two configurations: the default `CAF` configuration, and a custom configuration where the work-stealing runtime has been configured to use an *aggressive* polling strategy that maximizes system reactivity. As shown in the figure, at low message rate the latency of the default configuration is two orders of magnitude higher than the latency at high message rate when considering the same version (3880 vs. 58

microseconds). Instead, if we consider the *aggressive* version, the latency is significantly lower but, as shown by the bottom plot, CPUs' utilization is 100% regardless the input data rates.

This simple test demonstrates that the CAF message latency depends on the input data rates. Moreover, when the message latency is tuned by improving system reactivity (e.g. by using an aggressive polling strategy in the runtime), CPUs' utilization and consequently the system power consumption significantly increase. Therefore, merely acting on the knobs currently provided by the framework, it is not enough to fully optimize the performance/power ratio on an application.

The solution we propose to overcome these issues and making the message latency independent of the data rate is to modify the polling strategies in the work-stealing CAF scheduler. To this end, we removed threads passive sleeping by introducing on-demand notifications implemented via lightweight event signaling. This way we were able to improve system reactivity without compromising CPU utilization, power consumption, and system throughput. The technique proposed is general enough to be employed in the optimization of the performance/power ratio of runtime systems experiencing similar issues.

The rest of this paper is organized as follows: Sect II provides the backgrounds and describes recent related works; Sect III discusses the proposed solution; Sect IV presents the experimental results and finally Sect V provides conclusions.

II. Background and Related Work

This section provides background concepts that are useful to understand the contribution of this work.

A. The Actor Model

The *actor model* is a well-known concurrent programming model first proposed by Hewitt et al. [4] in the context of Artificial Intelligence. Later, the actor model has been formalized by Agha [5], [7]. Actors are concurrent entities, which interact exclusively via asynchronous messages. They are uniquely identified by an opaque identifier so that they can be transparently addressed during send operations. By providing network-transparent messaging, the actor model offers a high-level of abstraction for designing applications targeting parallel and distributed systems.

Each actor buffers input messages in a mailbox and processes them sequentially in a single logical step thus avoiding non-determinism in actor execution. Upon receiving a message, an actor can: (1) send messages to other actors, (2) spawn new actors to distribute workload and (3) change its internal behavior to process subsequent input messages differently.

Such event-based computation model prevents blocking waits for specific messages, which helps avoiding deadlocks in complex programs. Moreover, since actors can only interact via message passing, there is no shared state between actors so that they never corrupt each other internal state (i.e., local variables) avoiding potential race conditions. The lack of shared state among actors, together with asynchronous messaging enables actor programs to potentially exploit the processing capabilities of the single multi-core machine. Also, since actors are not tied to the specific physical machine because of their opaque addressing, the runtime systems can distribute actors across multiple multi-core machines de-facto enabling strong scalability.

B. Actor-based Programming in CAF

The C++ Actor Framework (CAF) [3] allows the development of concurrent programs based on the actor model leveraging on modern C++ language. Differently from other well-known implementations of the actor model, such as Erlang [8] and Akka [9], which use virtual machine abstractions, CAF is entirely implemented in C++, and thus applications implemented in CAF are compiled directly into native machine code. This allows using the high-level programming model offered by actors without sacrificing performance introduced by virtualization layers.

CAF applications are built decomposing the computation in small independent work items that are spawned as actors and executed cooperatively by the CAF runtime. Actors are modeled as lightweight state machines that are mapped onto a pre-dimensioned set of runtime threads called *workers*. Instead of assigning dedicated threads to actors, the CAF runtime includes a scheduler that dynamically allocates ready actors to workers. Whenever a waiting actor receives a message, it changes its internal state to *ready* and the scheduler assigns the actor to one of the worker thread for its execution. As a result, the creation and destruction of actors is a lightweight operation.

Actors that use blocking system calls (e.g., I/O functions) can suspend runtime threads creating either imbalance in the threads workload or starvation. The CAF programmer can explicitly *detach* actors by using the detached spawn option, so that the actor lives in a dedicated thread of execution. A particular kind of detached actor is the *blocking actor*. Detached actors are not as lightweight as *event-based* actors.

In CAF, actors are created using the `spawn` function. It creates actors either from functions/lambdas or from classes and returns a network-transparent actor handle. Communication happens via explicit message passing using the `send` command. Messages are buffered in the mailbox of the receiver actor in arrival order before they are processed. The response to an input message can be implemented by defining *behaviors* (usually through C++ lambdas). Different behaviors are identified by handler function signature, for example using *atoms*, i.e. non-numerical constants with unambiguous type.

Fig. 2 presents a simple example showing some of the features of the CAF framework.

Two actors exchange an integer value that is decremented until it becomes zero. Lines 35 and 36 initialize the CAF framework. Actors `ActA` and `ActB` are spawned at line 38 and 39, respectively. The second actor is spawned as `detached` actor while *ActA* is an *event-based* actor. Each actor defines three behaviors (`init_a`, `send_a` and `stop_a`) (using C++11 lambdas starting from line 14). Line 11 tells the runtime to skip messages until a `init_a` message is received.

```
1  #include <caf/all.hpp>
2  using namespace caf;
3  // set atoms
4  using init_a = atom_constant<atom("init")>;
5  using send_a = atom_constant<atom("send")>;
6  using stop_a = atom_constant<atom("stop")>;
7
8  // actor implementation
9  behavior myActor(event_based_actor* self) {
10  // skip messages until we receive an init atom
11  self->set_default_handler(skip);
12  return {
13   // init
14   [=](init_a, actor next, int N) {
15    // restore default action to printing unhandled messages
16    self->set_default_handler(print_and_drop);
17    // start sending
18    if (N > 0)  self->send(self, send_a::value, N);
19    self->become(
20     // send
21     [=](send_a, int n) {
22      if (n == 0) self->send(self, stop_a::value);
23       else        self->send(next, send_a::value, n - 1);
24     },
25     // stop
26     [=](stop_a) {
27      self->send(next, stop_a::value);
28      self->quit();
29     }
30    );
31   }
32  };
33 }
34 int main() {
35   actor_system_config cfg;
36   actor_system sys{cfg};
37   // spawning players
38   auto actA = sys.spawn(myActor);
39   auto actB = sys.spawn<detached>(myActor);
40   anon_send(actA, init_a::value, actB, 100);
41   anon_send(actB, init_a::value, actA, 0);
42 }
```

Fig. 2. A simple CAF example of two actors exchanging an integer value. The value is decremented by each actor until it becomes zero.

Then, by using the send behavior (line 20), the two actors exchange the integer value n each time decrementing it until it becomes 0. Eventually, the stop behavior is fired, which terminates the current actor only after having sent the other peer the stop message.

C. CAF work-stealing scheduler

The CAF scheduler consists of a single coordinator (not necessarily an active thread) and a set of worker threads. The default CAF scheduling policy is the *work-stealing* inspired to the well-known scheduling algorithm proposed in [10]. The fundamental idea of this algorithm is to remove the bottleneck of a single global work items queue using, instead, a work items queue for each worker thread (see Fig. 3).

Workers obtain work items only from their queue until it becomes empty. Then, the worker acts as a thief, i.e., selects one of the other workers as a victim and tries to steal a work item from its queue. In CAF, a work item (or job) is an actor ready to be executed. Top-level actors and messages from detached threads are initially assigned to workers' queue using a round-robin policy. Jobs move between worker threads only as a result of a stealing operation. This strategy minimizes communication between threads and maximizes local reuse of data.

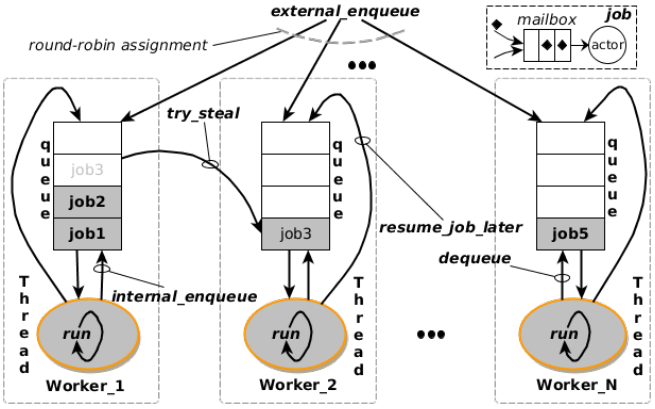

Fig. 3. Work-stealing implementation schema in CAF.

CAF uses a double-ended queue for its workers, which is synchronized with two spinlocks (one for the head and one for the tail). As sketched in Fig. 3, new "external" jobs are added to the back of the queue by using the method external_enqueue, while new work items generated by the worker itself are added to the front of the queue by using the method internal_enqueue. This approach is used to increase cache locality, since a new message that makes ready an actor to be executed is likely to be still in the cache. Workers execute the method run where the dequeue method is periodically called for obtaining jobs from the front of their queue and for stealing elements from the back of other workers' queue by using the method try_steal (called inside the method dequeue).

To regulate the number of times and the sleeping time between two distinct retries for obtaining a valid job, the method dequeue uses in sequence three polling strategies with different configurable values: 1) *aggressive*, 2) *moderate* and 3) *relaxed*. For each strategy, the worker tries to pop out a job from its local queue for a maximum number of *attempts* and, every *steal interval* attempts, it tries to steal a job from a victim worker. If all previous attempts fail, then the thread sleeps for a *sleep time* interval. Once the maximum number of attempts is reached, the worker moves to the next polling strategy. The *relaxed* polling strategy is a dead end, i.e. the worker will remain inside this strategy until it obtains a valid job to execute.

Considering the CAF default values reported in Table I, the *aggressive* strategy performs 100 attempts in total and 10 steal attempts with no sleep interval in between. The *moderate* strategy tries to steal for 100 times with 50 microseconds sleep interval between two steal attempts; the number of attempts for obtaining a job from the local queue is 500. Finally, the relaxed strategy runs indefinitely and the thread sleeps for 10 milliseconds between two attempts.

D. Pitfalls of work-stealing

Work-stealing is the default scheduling algorithm in several parallel frameworks such as Intel's Threading Building Blocks

TABLE I
DEFAULT VALUES FOR THE POLLING STRATEGIES OF THE WORK-STEALING SCHEDULER IN CAF.

Polling strategies	attemps	steal interval	sleep time (us)
aggressive	100	10	0
moderate	500	5	50
relaxed	1	1	10000

(TBB) [11], Cilk++ [12], X10 [13] and several OpenMP implementations.

One of the main downsides of the work-stealing algorithm is its termination phase or, more in general, the efficient detection of idle states [14]. This is because workers have local knowledge only, and when their queue runs out of work items, they do not know if in the other queues there are jobs to steal or, instead, if they could safely suspend themselves. Moreover, if there are very few jobs in the system, it is hard to decide if it is more convenient to keep trying to steal jobs (and for how many times) or instead if it is better to wait for a new job in the local worker queue or for enough jobs into the system.

These issues are typically faced by implementing smart polling strategies that postpone stealing retries using short sleeping intervals without blocking the thread indefinitely. In theory, if such sleeping delays are perfectly regulated it is possible to obtain the best performance without consuming extra CPU cycles and without adding further overheads. In practice, it is quite difficult to find optimal values for the backoff delays. Well-known techniques typically employed in spin-locks are based on exponential and linear backoffs [15]. The trade-off of the *spin-then-sleep* technique has been evaluated in [16] where it is shown that merely spinning or sleeping is sub-optimal in many real scenarios.

In addition, with highly variable input workload typical of streaming applications [1], different polling configurations are needed for the various phases of the applications making the tuning phase of the polling interval a difficult task. As we will see in Sect. IV, the use of wrong values, could affect application reactivity or significantly increase power consumption.

III. DESIGN

As described in Sect II, `CAF` uses three polling strategies with different retry intervals and sleeping delays. In streaming computations, if the arrival time is greater than the time spent for computing a task plus the time spent in the polling phases inside the *agrressive* and *moderate* strategies, the worker thread will fall to sleep in the *relaxed* policy which employs long sleeping time interval. In `CAF`, sleeping is implemented using the `std::this_thread::sleep_for` C++ command that suspends the current thread for the entire time value. If during the sleeping interval, a new job is scheduled in the thread's worker queue (calling the `external_enqueue` method), the thread will see this job only at the end of the sleeping interval. In the worst case scenario, this could be almost the entire time period, thus significantly increasing the total message latency.

To avoid this issues, it is sufficient to remove passive sleeping in the *relaxed* polling strategy, and to add explicit signaling on a per-thread *event* object. The worker thread will suspend itself waiting for an event that will be signaled by the `CAF` scheduler as soon as a new job has been pushed in the thread's worker queue. Since it is not possible to suspend the thread indefinitely until an event is received (this could lead to starvation), the maximum waiting time of the worker thread is regulated by a timeout associated with the event object.

```
1  class event_cv {
2    static constexpr int32_t RESET{257};
3    static constexpr int8_t  GREEN{0}, RED{1};
4  public:
5    void notify_one() {
6     // set the notify bit
7     event.b.notify.store(RED);
8     // if the peer is not waiting, return
9     if (!event.b.waiting.load()) return;
10    // prepare to wake-up the peer
11    event.b.waiting.store(GREEN);
12    syscall(SYS_futex,&event,FUTEX_WAKE_PRIVATE,RESET,
13            NULL,NULL,0);
14   }
15   void wait(struct timespec& timeout) {
16    while(event.b.notify.load()==GREEN) {
17     // prepare to go to sleep
18     event.b.waiting.store(RED);
19     int r=syscall(SYS_futex,&event,FUTEX_WAIT_PRIVATE,RED,
20                   &timeout,NULL,0);
21     event.b.waiting.store(GREEN);
22     // cheking if the timeout has expired
23     if (r==-1 && errno==ETIMEDOUT) return;
24    }
25    // reset the notify bit
26    event.b.notify.store(GREEN);
27   }
28 private:
29   union { // event data structure
30     int32_t                                          e;
31     struct { std::atomic<int8_t> waiting, notify;} b;
32   } event{GREEN};
33 };
```

Fig. 4. Event implementation using the `futex` system call in Linux.

An example of event object implemented exploiting the `futex` system call on Linux OS, is sketched in Fig. 4. No complex compare-exchange based loops are used, as all atomic operations required are embedded within the *futex* itself. An event can either be set (`RED` in the example) or unset (`GREEN`). If an event is unset, then the thread will wait until it is set. The waiting is implemented with a `futex` system call, which atomically checks if `event==RED` before putting the calling thread to sleep for at most the time interval set in the `timeout` (line 14 of Fig. 4). A similar implementation of event objects can be made by using the SIMD Extensions 3 instructions (SSE3) `monitor/mwait`, which introduce lower overhead than the `futex` system call. The *monitor* instruction defines an address range used to monitor write-back stores. The `mwait` instead is used to block the hardware context where the thread is running entering in an optimized power state waiting for a write-back store to the address range defined by the *monitor* call. Unfortunately, their applicability

is very limited because kernel-level privileges are required[1].

We decided to favor code portability with respect to absolute performance figures, therefore we implemented the event notification mechanism using only standard C++ mechanisms, i.e. C++11 *std::mutex*es and *std::condition_variable*s. The pseudo-code of the new `CAF work_stealing::dequeue` (called `DEQUEUE_NEW`) and `work_stealing::external_enqueue` methods that use the event protocol are sketched in Alg. 1 and Alg. 2, respectively.

Algorithm 1 New version of the work-stealing **dequeue**.

```
1:  function DEQUEUE_NEW( )
2:    ▷ try to get a valid job making use of the aggressive
      and moderate polling strategies only ...
3:    job=dequeue_old();     ▷ ... the job might not be valid
4:    while ( ! validJob(job) ) do
5:      mutex.lock()
6:      sleeping = true       ▷ the thread might go to sleep
7:      while ( ! thereAreJobsInTheQueue() ) do
8:        ▷ waiting for a message or timeout
9:        cv.wait_for(mutex,timeout)
10:     end while
11:     sleeping = false    ▷ woken-up, sleeping flag reset
12:     mutex.unlock()
13:     if (timeout.expired) then
14:       job = tryToSteal()
15:     else
16:       job = dequeueLocal()  ▷ dequeue without polling
17:     end if
18:   end while
19:   return job
20: end function
```

Algorithm 2 New **external_enqueue** version.

```
1: procedure EXTERNAL_ENQUEUE(job)
2:   queue.append(job);
3:   mutex.lock()
4:   ▷ checking if the thread has to be woken up
5:   if ((sleeping==true) and policy.thereAreJobs()) then
6:     cv.signal() ▷ send a signal to the condition variable
7:   end if
8:   mutex.unlock()
9: end procedure
```

Let us consider first the *dequeue_new* method. This method retrieves a valid job to execute either from thread's local queue or by stealing a job to one of the other workers' queues. The worker thread first tries to get a job by calling the `dequeue_old` method (line 3), making use of only the first two polling strategies (i.e. *aggressive* and *moderate*) with their configuration parameters. Then, if the job is not valid, i.e. the local queue is empty and, within the number of attempts of the first two polling strategies, the worker was not able to steal a valid job, then we assume that there are no (or very few) jobs to execute in the system and so the thread can be safely suspended waiting for a message notification or the timeout expire (line 9). The timeout is needed to periodically wake-up the thread to check if there are jobs in other threads' queues and, in that case, to try to steal jobs. More precisely, the worker is suspended on a condition variable (line 9), whose test checks if the local queue is not empty (the method *thereAreJobsInTheQueue()* at line 7 tests the emptiness of the local queue). Before suspending itself, the thread sets the flag `sleeping` to `true` (line 6). When the thread will be woken up by a signal or because the timeout expired, the `sleeping` flag is reset to `false` (line 11) and then either the thread tries to get the job from its local queue (line 16), or tries to steal a job (line 16).

Whenever a new work item has to be transferred to a worker by the coordinator entity or by another thread, the `work_stealing::external_enqueue` method is called (see Alg. 2). In this function, if the `sleeping` flag is set to `true` and if there are jobs in the worker queue selected to execute the job (line 5), the sleeping worker thread will be notified signalling the condition variable (line 6). The signal on the condition variable will wake-up the thread that was previously suspended.

The algorithm proposed, is independent of the kind of queue used and of the backoff policy employed inside the `dequeue` method. Furthermore, the algorithm is correct even if, instead of using the `wait_for` call, the blocking `wait` method is used (it blocks the thread until a notification is received without any timeout), provided that the method for checking the emptiness of the queue is correct.

IV. Evaluation

A. Experimental settings

All tests in this section were conducted on an Intel Xeon Server equipped with two Intel E5-2695 Ivy Bridge CPUs running at 2.40GHz and featuring 24 cores (12 per socket). Each hyper-threaded core has 32KB private L1, 256KB private L2 and 30MB of L3 shared cache. The machine has 64GB of DDR3 RAM, running Linux 3.14.49 x86_64 with the CPUfreq *performance* governor enabled.

We used `CAF` version 0.15.3 and the GNU gcc compiler version 6.4.0 with -O3 optimization flag. For all tests, the `CAF` runtime uses 24 worker threads [2]. The MAMMUT library [17] version 1.0.0 is used to collect power consumption and CPU utilization. On the specific architecture considered, MAMMUT relies on RAPL counters to collect power consumption.

The code changes proposed in this paper can be downloaded from the GitHub `CAF` repository[3]. At the time of writing, the patch has been accepted by the `CAF` maintainer as a separate branch (*topic/latency*) and are currently under consideration to be merged in the *master* branch.

B. Benchmark description

The benchmark used for the tests is a linear pipeline of `CAF` actors. The actor topology is sketched in Fig. 5. The first actor,

[1]Intel 64 and IA-32 Architectures Software Developer Manuals.

[2]Command line option `--caf#scheduler.max-threads=24`.

[3]CAF at GitHub: https://github.com/actor-framework/actor-framework

defined as a `blocking actor`, produces messages (either weightless or with a payload) at a given constant rate. The rate is provided to the benchmark as command line parameter.

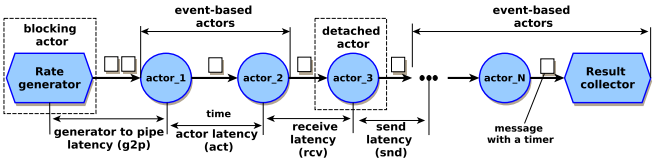

Fig. 5. Pipeline benchmark.

In the following, we report only the results for weightless messages. The reason is twofold: (1) we want to analyze the overheads introduced by the `CAF` runtime; (2) the C++ *moving semantics* extensively used in the `CAF` runtime implementation makes the cost of sending messages with payloads almost negligible since only memory references are moved without any memory copies. Finally, the benchmark can also spawns one actor as *detached* and this allows to analyze the latency of messages between *event-based* and *detached* actors.

Each message generated has associated a timer that is started just before the `Rate generator` actor sends the packet to the first pipeline actor (actor_1 in Fig. 5) and stopped as soon as the packet is received by the `Result collector` actor. The `Result collector` computes a moving average with an overlapping constant size of 5 values and eventually produce in output the average value.

The pipeline benchmark is able to extract not only the total average message latency but also the message latency between the `Rate generator` and the first pipeline actor (*g2p* in Fig. 5) and the *receive* and *send* latency between an *event-based* actor and the *detached* actor (or viceversa), referred as *rcv* and *snd* in Fig. 5, respectively.

The benchmark code can be downloaded from GitHub[4].

C. Configurations tested

We considered four distinct benchmark configurations:

default: it uses the default CAF values for the three polling strategies used in the work-stealing scheduler (see Table I in Sect. II).

custom: it uses customized values for the stealing interval and for the sleeping time of the *moderate* and *relaxed* polling strategies to increase threads reactivity. In particular for the *moderate* strategy, we reduced the sleeping time from 50us to 1us, whereas for the *relaxed* strategy, we increase the stealing interval from 1 to 5 and we drastically reduced the sleeping time from 10ms to 1us.

aggressive: this configuration uses 0us as sleeping duration for all polling strategies. This means that the runtime threads will keep polling their input queues.

new: this version uses the code changes proposed in Sect III. Moreover, as in the custom version, we set the sleeping time of the *moderate* strategy from 50us to 1us and the timeout of the `wait_for` call (line 9 of Algorithm 1) is set according to the *relaxed* sleep time (see Table. I).

D. Results

Here we discuss the results obtained running the benchmark for 100s in the four different configurations studying the message latency, the power consumption, and CPUs' utilization. Our Xeon server has approximately 40 Watts idle power and a max total power consumption of about 180 Watts.

Fig. 6 reports the message latency, the power consumption and the CPUs' utilization varying the number of actors in the pipeline benchmark for a message rate of 10msg/s. As expected, the *default* version has a latency that is two orders of magnitude higher than the other versions. The *aggressive* version provides the lower latency but the highest power consumption and CPU utilization. The *new* version has an average latency comparable with the *custom* and *aggressive* versions consuming less power and less CPU cycles.

If we consider higher message rate whose results are shown in Fig 7, we can see that the *new* version has a lower latency and power consumption than the *default* version. The CPU utilization is marginally higher: from $5-8\%$ for the *default* version to $8-10\%$ of the *new* version.

In the previous tests, we used 10ms as timeout value for the `wait_for` call in the *new* version. In Fig. 8, we study the impact of the timeout value on both the message latency, power consumption and CPUs' utilization at low data rate.

Using timeout values lower than 500us has a positive impact on the message latency (from 45us to 25us), whereas the effect on the power consumption is about 15% increase and the CPU utilization remains almost constant (about 1% increase). This suggests that there is an optimal value for the timeout in between 100us and 500us, which reduces the message latency without significantly affecting power consumption and CPU utilization. Tuning the timeout value is essential for reducing the message latency, but differently from the polling time used in the *default* version, finding the optimal timeout value for the event mechanism in the *new* version is just an optimization step. In fact, selecting larger values does not have severe adverse effects as in the *default* version while selecting lower values has only a minor impact on power consumption.

In Fig. 9 (left-hand side) we compare the *custom* and the *new* version for a pipeline of 12 actors varying the *relaxed* polling time and the *wait_for* timeout, respectively. The message latency of the *custom* version has a range of variation that spans from 3.5ms to about 40us while the latency for the *new* version varies from about 55us to 30us. Moreover, the optimal value for the timeout of the *new* version is about 200us: lower values increase CPUs' utilization (and power consumption), higher values slightly increase the message latency.

The jump in the message latency between 500us and 200us is due to the *wait_for* implementation[5] that, on Linux systems, uses a smart *spin-then-sleep* policy to avoid putting the thread to sleep if the sleeping time is lower than a threshold [18].

[4]http://github.com/ATS-Advanced-Technology-Solutions/caf_latency_benchmarks

[5]Indeed, it is implemented on top of the `pthread_cond_timedwait`.

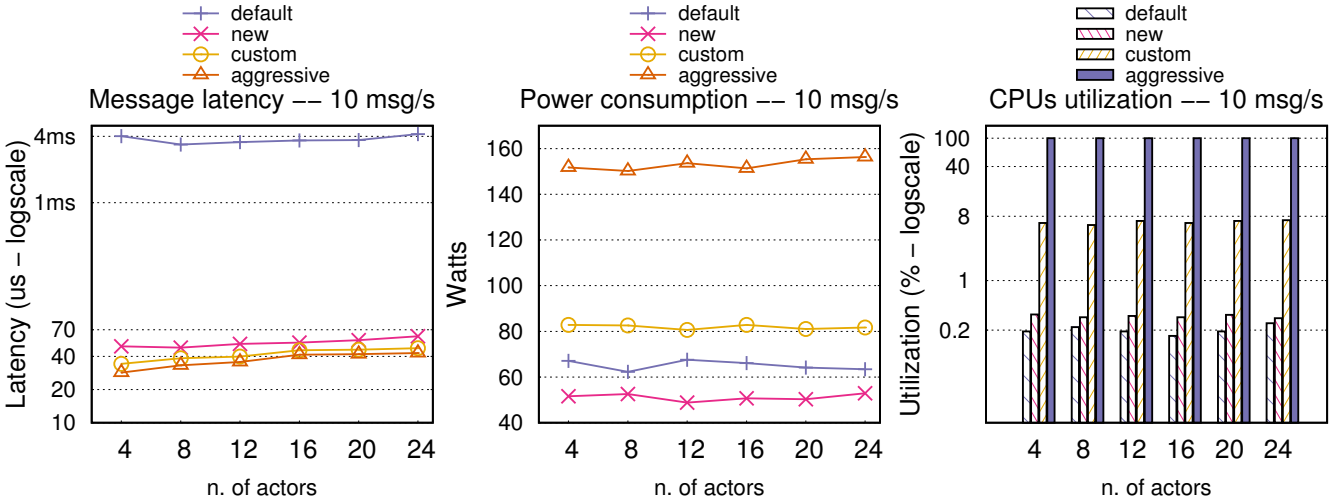

Fig. 6. Average message latency (microseconds – Left), power consumption (Watts – Center) and CPUs' utilization (% – Right) of the different configurations in the pipeline benchmark varying the n. of actors when the message rate is low: 10msg/s.

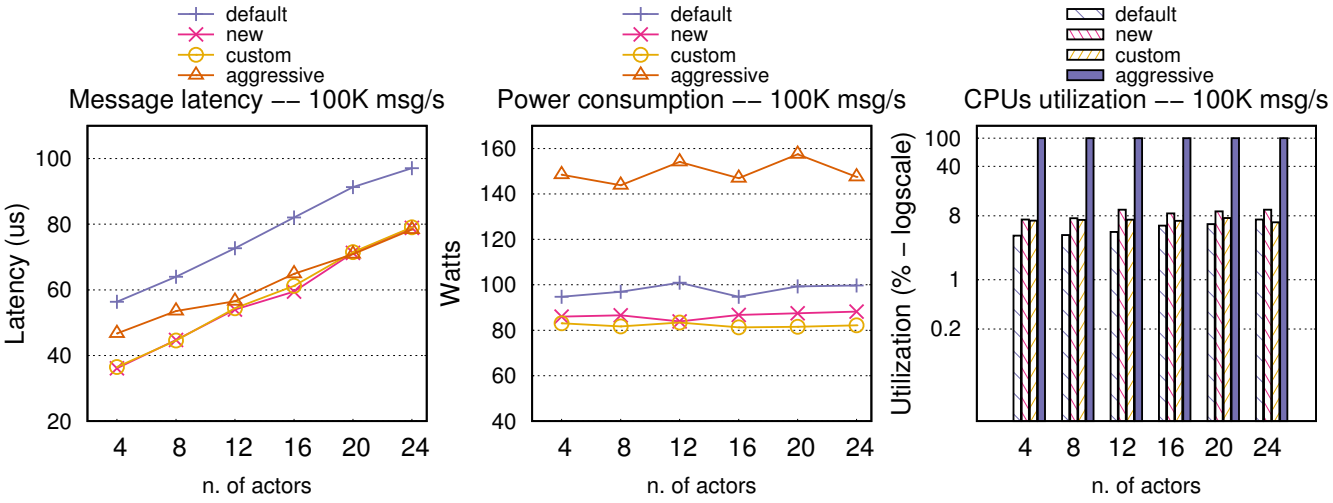

Fig. 7. Average message latency (microseconds – Left), power consumption (Watts – Center) and CPUs' utilization (% – Right) of the different configurations in the pipeline benchmark varying the n. of actors when the message rate is high: 100Kmsg/s.

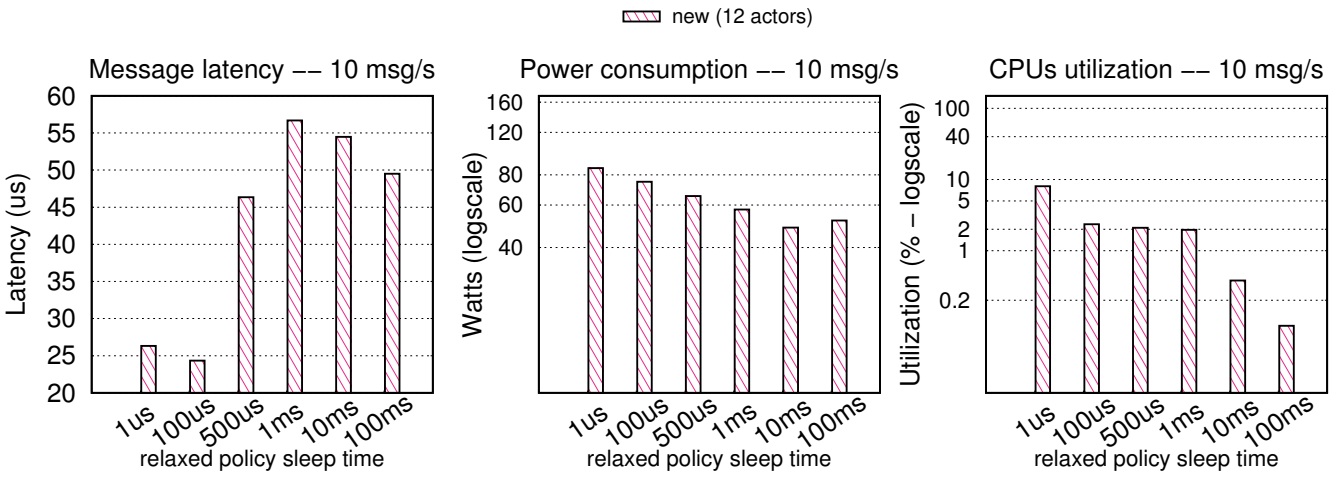

Fig. 8. Average message latency (microseconds – Left), power consumption (Watts – Center) and CPUs' utilization (% – Right) varying the relaxed sleeping time. Low message rate: 10msg/s.

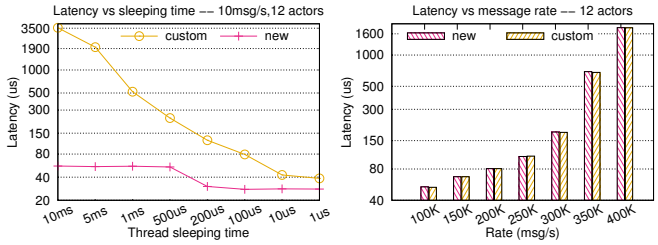

Fig. 9. Left:) Message latency (microseconds) varying the *relaxed* polling time. Right:) Message latency varying the message rate for the *new* version with a `wait_for` timeout of 300us; the a*custom* version uses a polling time of 1us for the *relaxed* policy.

In the right-hand side of Fig. 9, we reported a comparison between the *custom* and *new* version when the message rate is high (from 100Kmsg/s to 400Kmsg/s) and the timeout is set to 200us. As shown the two versions provide almost identical message latency proving that the proposed changes to improve the message latency at low data rates do not produce negative effects at high data rates.

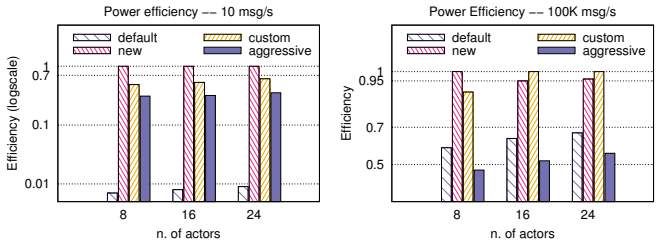

Fig. 10. Power efficiency (throughput/power) of the different configurations varying the n. of actors in the pipeline benchmark.Two message rates considered: 10msg/s (Left) and 100Kmsg/s (Right).

Fig. 10 reports the power efficiency (measured as *throughput per power*) of the tested configurations considering two message rate scenarios (low rate – left-hand side – and high rate – right-hand side) and different pipeline configurations (8, 16 and 24 actors) normalized to the most efficient version for each case. The higher the value, the more efficient the execution of the program. The *new* version uses a timeout value of 200us. For low data rate, the *new* version is always the most efficient for all cases tested. As expected, the *default* version is the worst with an efficiency lower than 1%. For high data rate, instead, the *new* version provide the best power efficiency only when few actors are used in the benchmark (up to 12 actors), in the other cases the best version is the *custom* one. Notwithstanding, the *new* version provides a power efficiency that is greater than 95%. Not surprising, the *aggressive* version is always the worst one. In fact, since there is no backoff between two distinct polling operations, worker threads fight one each other contending CPU's cores and eventually increasing the overhead for making progress. Also, if the worker is active on a core either doing useful work or spinning, the underlying hardware context remains busy as well, so the OS cannot switch-off core's components and sets the core in a low-power state. Such core draws power regardless of the type of work the thread executes.

TABLE II
ACTOR LATENCY BREAKDOWN FOR A PIPELINE OF 12 AND 24 ACTORS WHERE THE 8-TH ACTOR IS SPAWNED AS *detached*.

Rate	new							
	g2p		act		rcv		snd	
	12	24	12	24	12	24	12	24
10	19	24	≤1	≤1	19	18	22	22
100K	20	24	3	3	20	20	8	9

In Table II we reported the message latency breakdown considering the different kinds of `CAF` actors (*blocking*, *event-based* and *detached*) spawned in the pipeline benchmark (see Fig. 5). The table reports the data collected for two pipeline configurations (12 and 24 actors) and two message rates (10msg/s and 100Kmsg/s). As can be noted, the *event-based* actor latency (*act*) is small compared with the other values. This is because almost all communications occur within the same worker thread that pushes/pops messages to/from mailbox queues that are implemented using lock-free techniques. Instead, *blocking* and *detached* actors perform inter-thread communications pushing and popping jobs to/from a (spin-)lock-based dequeue that might also require event signaling, thus increasing the communication overhead.

Finally, we have evaluated the performance of the `new` version considering the Savina benchmarks, a benchmark suite for actor-oriented programs [19]. The Savina benchmark suite focuses on computationally intensive applications and includes both numeric and non-numeric problems. The CAF porting of the benchmark suite comprises 23 out of 30 applications available. Some of them can be executed in different configurations.

Fig. 11 shows the average performance variation results (in percentage) obtained running the Savina benchmarks on the target platform. We considered the `default` CAF version vs. the version with the code changes proposed in Sect III (`new`). Higher values are better, meaning that the `new` version provides increased performance with respect to the `default` CAF version. As shown in the plot, there are only few cases where the use of event notifications in the work-stealing runtime negatively affects the performance (in the range 1-13%). In particular, only for one benchmark of the suite (*14logmap-slow*) we experienced an average performance degradation of 13%. Conversely, there are many test cases where the performance improves significantly (up to 72%). This large improvement is mainly due to the fact that several benchmarks use few active workers and the proposed changes in the CAF runtime remove contention on the active workers' queue.

V. CONCLUSIONS

In this paper, we have experimentally evaluated the message latency of the actor-based framework `CAF` on a multi-core platform. We used as testing workload a configurable benchmark where a set of actors are connected in a pipeline topology. Our experimental results show that message-passing

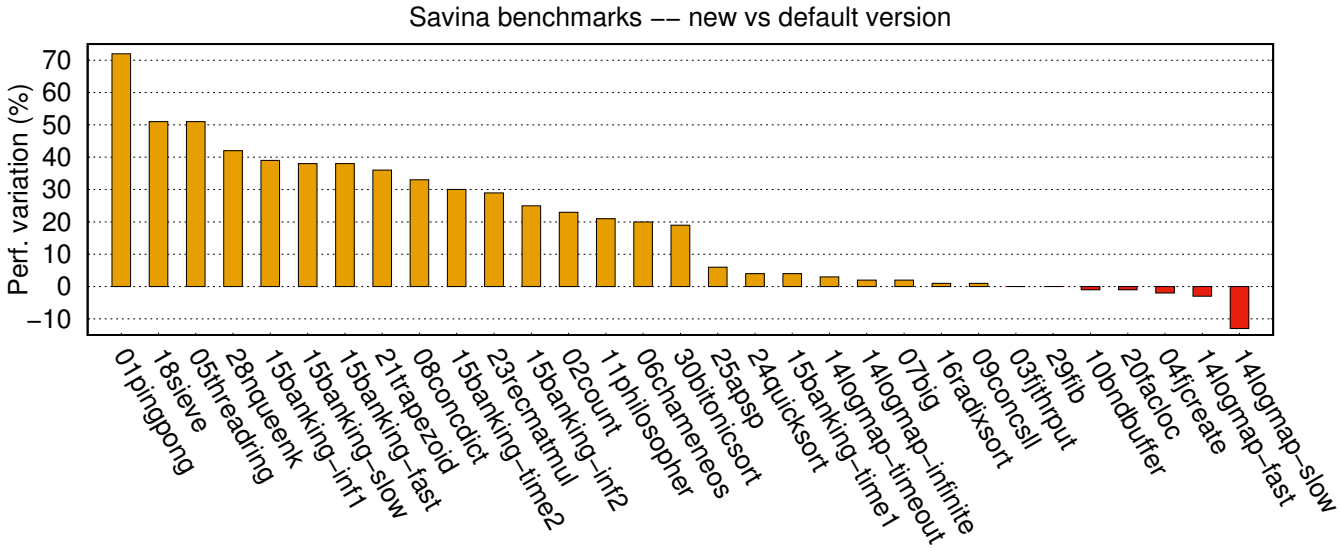

Fig. 11. Savina benchmarks performance results obtained running the `new` vs. the `default` version of the work-stealing runtime. The bars measure the average performance variation (in percentage) over 5 executions of each benchmark. A value greater than 0 means better performance.

latency between `CAF` actors depends on the input message rate, precisely, at low and moderate rates messages experience higher latency than at high data rates.

To overcome this issue, we proposed a modification in the work-stealing polling strategy employed in the `CAF` runtime, consisting in removing passive thread sleeping and adding explicit event notifications by using a per-thread event object implemented in a portable way by using C++ condition variables. The protocol used is independent of the work-stealing implementation used in `CAF`, and it is general enough to be used in others runtime with similar issues. The proposed code changes significantly reduce message latency (up to two orders of magnitude for low data rates), power consumption, and CPU utilization at low and moderate data rates without compromising system throughput at high data rates.

ACKNOWLEDGEMENTS

This work has been supported by ATS Spa with the project "*Optimizing mailbox latency and reactivity in CAF*".

REFERENCES

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

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 365–376, Jun. 2011.

[3] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments," in *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2013, pp. 87–96.

[4] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proc. of the 3rd Int. Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[5] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[6] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting actor programming in c++," *Computer Languages, Systems & Structures*, vol. 45, no. Supplement C, pp. 105 – 131, 2016.

[7] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *J. Funct. Program.*, vol. 7, no. 1, pp. 1–72, 1997.

[8] J. Armstrong, "The development of erlang," *SIGPLAN Not.*, vol. 32, no. 8, pp. 196–203, Aug. 1997.

[9] "Typesafe Inc." Akka. http://akka.io.

[10] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324234

[11] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. "O'Reilly Media, Inc.", 2007.

[12] C. E. Leiserson, "The cilk++ concurrency platform," in *Proc. of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 522–527.

[13] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick, "Deadlock-free scheduling of x10 computations with bounded resources," in *Proc. of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '07. New York, NY, USA: ACM, 2007, pp. 229–240.

[14] Ž. Vrba, H. Espeland, P. Halvorsen, and C. Griwodz, *Limits of Work-Stealing Scheduling*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 280–299.

[15] A. Agarwal and M. Cherian, "Adaptive backoff synchronization techniques," *SIGARCH Comput. Archit. News*, vol. 17, no. 3, pp. 396–406, Apr. 1989.

[16] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein, "Optimal strategies for spinning and blocking," *Journal of Parallel and Distributed Computing*, vol. 21, no. 2, pp. 246 – 254, 1994.

[17] D. De Sensi, M. Torquati, and M. Danelutto, "Mammut: High-level management of system knobs and sensors," *SoftwareX*, vol. 6, pp. 150 – 154, jul 2017.

[18] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis, "Unlocking energy," in *USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 393–406.

[19] S. M. Imam and V. Sarkar, "Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, ser. AGERE! '14. New York, NY, USA: ACM, 2014, pp. 67–80.