Adaptive message passing polling for energy efficiency: Application to software-distributed shared memory over heterogeneous computing resources

Loïc Cudennec

Adaptive Message Passing Polling for Energy Efficiency: Application to Software-Distributed Shared Memory over Heterogeneous Computing Resources

Loïc Cudennec[1,2]

[1]CEA, LIST
F-91191, PC 172, Gif-sur-Yvette, France
[2]DGA MI, Department of Artificial Intelligence
BP 7, 35998 Rennes Armées, France
loic.cudennec@def.gouv.fr

Abstract

Autonomous vehicles, smart manufacturing, heterogeneous systems and new high-performance embedded computing (HPEC) applications can benefit from the reuse of code coming from the high-performance computing world. However, unlike for HPC, energy efficiency is critical in embedded systems, especially when running on battery power. Code base from HPC mostly relies on the MPI message passing runtime to deal with distributed systems. MPI has been designed primarily for performance and not for energy efficiency. One drawback is the way messages are received, in an energy-consuming busy-wait fashion. In this work we study a simple approach in which receiving processes are put to sleep instead of constantly polling. We implement this strategy at the user level to be transparent to the MPI runtime and the application. Experiments are conducted with OpenMPI, MPICH and MPC, using a video processing application and a software-distributed shared memory system deployed over two heterogeneous platforms, including the Christmann RECS|Box Antares Microserver. Results show significant energy savings. In some particular cases involving process colocation, we also observe better performance using our strategy which can be explained by a better sharing of the computing resource.

1 Introduction

High-Performance Embedded Computing (HPEC) refers to application contexts that require important computing capabilities while running with limited power and computing resources. This includes autonomous vehicles, smart manufacturing, edge computing. Applications from the HPC domain are also relevant for the HPEC domain: there is a need for vision, video processing, machine learning and inference, multi-physics simulation and decision systems to be mixed with monitoring, entertainment and real-time control-command systems. In HPEC, the hardware can be heterogeneous with a mix of general purpose processors, low-power processors, GPUs, FPGAs and dedicated accelerators.

These architectures are distributed and the computing nodes are connected through specific CAN or generic Ethernet networks. Message passing is usually under the hood, whether the proposed programming model is service-oriented, dataflow or shared memory. The Message Passing Interface (MPI) standard, while being dedicated to HPC, is a good candidate for HPEC. It is a well-established programming framework with an important community and a huge HPC application library in numerous application domains. It also features automatic deployment using classic and specific communication mediums (Ethernet, high-speed networks) and convenient tools for debugging and tuning.

The HPEC context not only focuses on performance, but also on the energy consumption: some devices run on battery. However, the primary goal of most MPI implementations is performance. One important drawback is the massive energy consumption spent when a process is waiting for messages. This is a reasonable strategy in HPC for which tasks should not wait for data, and in that case, MPI implementations provide a high degree of responsiveness by continuously polling for new messages. A second drawback is the assumption that an MPI process is pinned to a computing element and is not supposed to share the resource with other processes.

Message polling and exclusive access to resource strategies are not adapted to HPEC. Some applications include specific roles that might run idle most of the time. For example, input and output (I/O) tasks, monitoring, remote procedure call (RPC) servers. These roles lead to sparse communications and significant waiting times for processes. Some processes might also have to be colocated on the same device because they

need access to a specific hardware element. Another simple reason is that the targeted hardware may have less computing nodes than required by the minimal instance of the application, hence oversubscribing some processes.

In this work, we explore the possibility to relax the busy-wait loop for message reception in order to limit the energy consumption. Our approach is to implement micro-sleeping at the user level, which is transparent to the MPI runtime. We study this strategy onto distributed heterogeneous architectures using synthetic applications running on top of the OpenMPI, MPICH and MPC runtimes. We thereafter evaluate the benefit of using micro-sleeping within a software-distributed shared memory (S-DSM) running on top the OpenMPI runtime. Results show a significant drop of the CPU load and the expected energy savings. In some cases, we also observe a performance increase due to a better scheduling of colocated processes.

This article is organized as follows. Section 2 discusses about implementations of the receive function in popular MPI runtimes. Some experiments are presented in which the CPU load is measured on the receiver side using OpenMPI, MPICH and MPC runtimes. Section 3 presents how to implement micro-sleeping at the user level and shows how it drops the CPU load compared to previous experiments. It also discusses the use of the `clock_nanosleep` function in `C` and it evaluates its accuracy given different hardware platforms and Linux OS distributions. Section 4 introduces a software-distributed shared memory (S-DSM) running on top of the OpenMPI runtime as a motivating example and explains the expected benefits of implementing micro-sleeping into this system. In section 5 the S-DSM is deployed onto a distributed heterogeneous platforms and the micro-sleeping strategy is evaluated using an image processing application and a video stream processing application. Section 6 focuses on the energy consumption improvements obtained by deploying the video stream processing application over an industrial-grade micro-server. Section 7 gives some insights into related work that address the problem of energy savings in point-to-point communications as well as at the message passing runtime scale. Finally, section 8 concludes and proposes some perspectives to this work.

2 Busy Wait

Distributed applications that rely on message passing usually perform explicit communications. The most simple primitives are the complementary `Send` and `Recv` functions called by most likely two different processes. In this work we consider classical distributed systems for which there is no *global clock*: events are solely linked by *causal dependencies* and messages are expected to be delivered *eventually*. For instance, we do not consider real-time networks and there is no determinism on message reception. The latter assumption prevents us from calculating the exact date of arrival of messages using off-line compilation analysis. It is not possible to build a tight schedule for process communications. Therefore, while sending a message from A to B, it is not possible to calculate the local time between process B starts to wait for this message and the local time it effectively receives it, as there is no determinism in communications [Lamport, 1978]. However, another approach is to rely on heuristics, which is not addressed is this work.

Popular implementations of the message passing runtime MPI assume to be deployed in an HPC context, as it was intended to be when first introduced in the early nineties. In this context, a process shall never wait for incoming messages and the whole processing time should be dedicated to run numerical code. This explains why the optimization and tuning of distributed applications are largely focusing on reducing the communication costs, including message waiting times. Receiving messages is not only explicitly triggered by the user code, it is also hidden behind built-in framework synchronization mechanisms such as *lock*, *barriers*, *rendez-vous*, *split-join*, *map-reduce* and other collective operations. In other words, if some application processes spend too much time waiting for data, then the application has probably to be redesigned and the data granularity should be changed for better parallelism. From this HPC context, most of the `Recv` function implementations will somehow busy-wait and continuously poll for new messages, the latter being CPU-demanding. Another assumption is the exclusive deployment of MPI processes onto computing resources, limiting to one MPI process per core (one `rank` per `slot`). Process colocation (also referred to `oversubscribing`) is usually not advised because it requires a tight understanding of process behaviors to avoid CPU and network contention (using *job striping* [Breslow et al., 2016] for example). Some of the performance in the MPI runtime are achieved thanks to the busy-wait strategy, without process colocation, to get the best reaction time when doing nothing but expecting a message.

A classical MPI user code for receiving messages is made of two steps: 1) a call to the blocking, busy-wait `Probe` function that returns when a message is available and provides information such as the size and the sender id, followed by 2) a call to the `Recv` function that provides the message (Figure 1a). The size of the message given by the first step is used to allocate a buffer large enough to store the message in the second step. Applications that exhibit regular communication patterns with hard-coded message size may skip the first step to simplify the code. It is also possible to use the `Iprobe` non-blocking function. In that case, a common user implementation is to loop onto `Iprobe` until a message is available, hence implementing a busy-wait (Figure 1b).

While it is not imposed by the MPI specifications, these implementations choices are not adapted to

```
1   void receiver_blocking() {
2     char * buffer;
3     size_t size = 0;
4     /* implicit busy wait */
5      Probe(&size);
6     buffer = malloc(size);
7      Recv(buffer);
8   }
```

(a) Blocking Probe receiver

```
1   void receiver_non_blocking
       () {
2     char * buffer = NULL;
3     size_t size = 0;
4     int flag = 0;
5     /* explicit busy wait */
6     while (flag == 0) {
7        Iprobe(&flag, &size);
8     }
9     buffer = malloc(size);
10     Recv(buffer);
11  }
```

(b) Non-blocking Probe receiver

Figure 1: MPI blocking and non-blocking Probe receiver.

the HPEC context for two main reasons: 1) some applications include processes that do nothing but wait for an incoming event and 2) the application deployment over a heterogeneous system might require the colocation of some processes. In order to tackle these issues, some implementations offer different running modes. For example, OpenMPI [Gabriel et al., 2004] provides a Modular Component Architecture (`mca`) option called `mpi_yield_when_idle` that is used to decrease the process priority to enhance resource time-sharing. This running mode is referred to `Degraded` in opposition to the regular `Aggressive` mode. This mode is activated when running the application using a specific command line parameter. A description is given in the documentation:

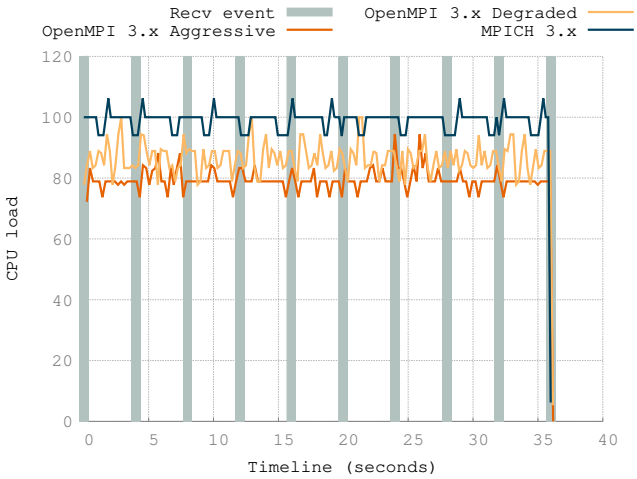
> "Degraded: When OpenMPI thinks that it is in an oversubscribed mode (i.e., more processes are running than there are processors available), MPI processes will automatically run in degraded mode and frequently yield the processor to its peers, thereby allowing all processes to make progress."

The MPICH [Thakur and Gropp, 2007] runtime also implements a specific mode for oversubscription. From the frequently asked question section, under the question "*Why does my MPI program run much slower when I use more processes?*", the documentation indicates that the current channel implementation named `nemesis` is based on busy polling to improve intra-node performance. The proposed workaround is either to avoid running more processes than available cores in the system, or to use the old channel implementation `sock` (probably based on regular network/Unix sockets). The latter solution is said to handle oversubscription in a better way but at the price of degrading intra-node communications. Therefore, the user has to guess what is the most efficient option for the application and the hardware, keeping in mind that only a subset of the processes are concerned by oversubscription. It also requires to rebuild MPICH with a specific option which does not allow to activate oversubscription on a per process basis.

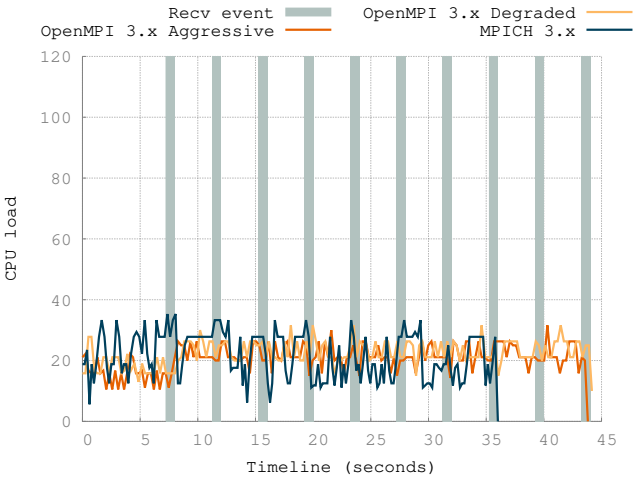
In the following experiments, if not otherwise specified, we use the OpenMPI 3.x runtime because of its popularity and the possibility to compile the source code without a glitch onto different Linux distributions (Ubuntu, Debian, Raspbian, Lebian) and processors (Intel Core i7, Arm Cortex) deployed in our heterogeneous platform.

▷ **Experiment A: measuring the CPU load with different MPI implementations.** In this experiment, we deploy a simple MPI program that sends and receives messages. The first process (`send`) performs 10 cycles, in each cycle it sends 10 messages then sleeps 4 seconds. The second process (`recv`) loops on message reception using the blocking `Probe` function as in Figure 1a. In a first set of experiments, processes are deployed on an Odroid XU4 embedded computing device (Cortex-A15 and Cortex-A7 big.LITTLE processor) and a Raspberry Pi 3B+ (Cortex-A53 processor). The devices are connected through a Gigabit Ethernet network. We run this MPI program using the OpenMPI 3.x runtime in both `Aggressive` and `Degraded` modes, and we compare with the MPICH 3.x runtime. We measure the CPU load (%CPU) by deploying a concurrent thread within the `recv` process that periodically logs the CPU usage of the UNIX PID using the `top` command every 0.2ms. In the remaining of the article, CPU load refers to this measure and is given as %CPU according to the `top` command [1].

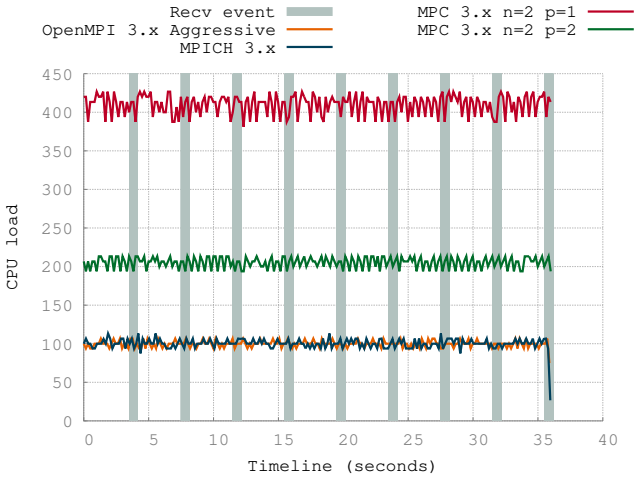
[1] The Unix *top* command displays the percentage of use of a single-core machine. On a multi-core system, this returns the sum of all cores. Therefore, a multi-threaded process can exceed 100% CPU load by using more than one core at the same time.



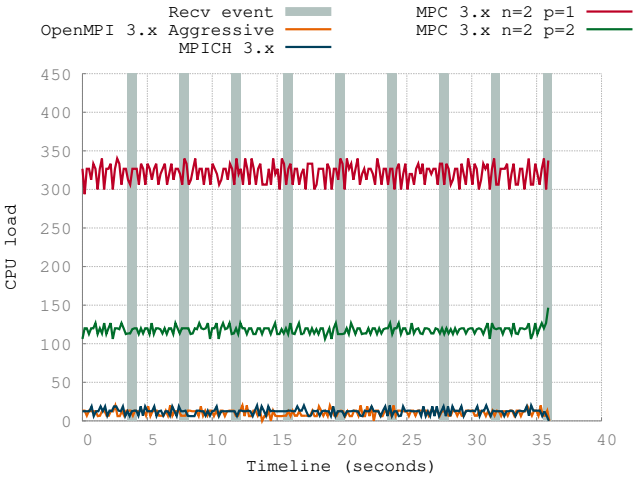
(a) Odroid XU4, Raspberry Pi 3B+, regular MPI Probe



(b) Odroid XU4, Raspberry Pi 3B+, adaptive MPI Probe



(c) Intel Core i7 6800K (6 cores), regular MPI Probe



(d) Intel Core i7 6800K (6 cores), adaptive MPI Probe

Figure 2: Measuring the CPU load (top %CPU) of the MPI process on the receiver for a basic MPI send-receive program.

```
1   void receiver_adaptive() {
2     char * buffer = NULL;
3     size_t size = 0;
4     int flag = 0;
5     unsigned long sleep_time = 0;
6     while (flag == 0) {
7        Iprobe(&flag, &size);
8       if (flag == 0) {
9          clock_nanosleep(sleep_time);
10        sleep_time += sleep_time_step;
11      }
12    }
13    buffer = malloc(size);
14     Recv(buffer);
15  }
```

Figure 3: Pseudo-code for Adaptive Probe receiver. A GNU General Public License implementation is available in the EEProbe project [EEP,].

Figure 2a shows the CPU load of the `recv` process using the regular MPI Probe implementation. The main conclusion is that, between two bursts of messages, the CPU keeps running high for both OpenMPI and MPICH. Using such a simple experiment, it is not possible to explain why the OpenMPI `Degraded` mode implies slightly higher CPU load compared to the OpenMPI `Aggressive` mode, except that this scenario does not co-locate processes on the same node, which is the purpose of the `Degraded` mode. Furthermore, the runtime scheduler does not yield to another MPI process, which might only add a scheduling overhead. However, the general behavior remains the same and there is no noticeable evidence that the OpenMPI runtime yields the processor to other processes.

In a second set of experiments, we co-locate the `send` and `recv` processes onto an Intel Core i7 6800K 6-core processor. We run the `send-recv` program using the OpenMPI runtime, the MPICH runtime and the MPC runtime [Pérache et al., 2008]. MPC is an MPI implementation focusing at the thread level to enhance parallelism management when deploying multi-threaded MPI code onto NUMA machines. In a classical MPI deployment, the number of MPI processes is given using the `-np` parameter, one MPI rank per process. With MPC, the `-n` parameter sets the number of MPI ranks, while the `-p` parameter sets the number of processes, making possible to share memory regions between MPI ranks located in the same process. Therefore, in Figure 2c, we deploy the `send-recv` program using two MPC configurations: 1) `n=2,p=1` uses only one process to host the two MPI ranks (`send` and `recv`) and 2) `n=2,p=2` one process per rank as for the other MPI runtimes. Results show that the OpenMPI and MPICH runtimes use the equivalent of one full-time core. MPC configurations use respectively 2 and 4 full-time cores for `p=2` and `p=1`. The MPC runtime achieves internal thread management that overloads more than one core per process. Using only one process to host several MPI ranks noticeably increases the CPU load compared to using one process per rank. However, please note that we only report on the CPU load of the runtimes and we do not evaluate the general computing performance of the application, which is what the end-user is expecting.

In short:
- MPI processes keep the CPU busy even when waiting for incoming messages.
- This happens when using OpenMPI, MPICH and MPC runtimes.
- This study aims at reducing the CPU load in this particular situation.

3 Sleeping at Work

One of the best strategies to save energy is to sleep. The counterpart is to know when and how to wake up. A reasonable way of doing this is based on interrupts: a process is put to sleep and waked up by the operating system scheduler on a given signal. In our context, a signal should be raised whenever a matching message is received. This requires to modify the message passing runtime to interact with the OS scheduler, which is OS-dependent. In this work, we stick to a portable implementation that can be deployed at the user level. The basic idea is to loop onto the `Iprobe` non-blocking function and to sleep after each call. The sleep time is increased after each unsuccessful probe and set back to the minimal sleep time on a message reception. This way, the system is able to handle communication bursts as well as long idle times. Such approach does not require the deployment of a modified version of the message passing runtime. In the remaining of the work, we

refer to the following implementation as the `Adaptive` mode. A pseudo-code is given in Figure 3 and a GNU General Public License implementation is available in the EEProbe project [EEP,].

▷ **Experiment B: measuring the CPU load with the Adaptive mode.** These experiments are based on previous `send-recv` experiments, sharing the same user code, MPI runtimes and computing nodes. We only introduce a wrapper between the user code and each call to the MPI receive function. This wrapper implements the `Adaptive` mode as presented in Figure 3. Results are given in Figures 2b and 2d. When using OpenMPI and MPICH, the CPU load has dropped to values around 20% of the CPU usage on ARM-based processors and below 10% on the Core i7 processor. Unfortunately, the MPC runtime is still using a large amount of CPU time despite a significant decrease by an order of a full core usage (around 90%). The MPC runtime is massively multi-threaded. Therefore, when putting the user thread into sleep, other MPC threads obviously keep the CPU busy, limiting the benefit of using the `Adaptive` mode.

In short:	•Micro-sleeping while waiting for messages leads to a significant drop of the CPU load. •This CPU load drop is important when using the OpenMPI and MPICH runtimes and noticeable -hence a disappointment- when using the MPC runtime.

Most of the operating systems used in HPC and -to a lesser extent- in embedded computing are based on *non real-time* (RT) distributions. In non-RT OS, the scheduler works in best effort, without any consideration for time and deadlines. Processes yield and run without determinism but taking into account the starvation problem. Therefore, we cannot expect the `clock_nanosleep` function to behave accurately as in an RT system. In the receiver design, an erratic call to `clock_nanosleep` would cause bad performances in the case of a burst of messages and few-to-none energy savings by not sleeping enough in the case of sparse communications.

Waiting a given amount of time and waking up when expected -with a high level of accuracy- is usually implemented using a busy-wait loop so as not to miss the deadline. In this work, we want to avoid this busy-wait loop. Therefore, the `clock_nanosleep` function allows to yield to another process and set a resolution for the sleep duration down to the nanosecond (clock resolution is given in the `/proc/timer_list` system file).

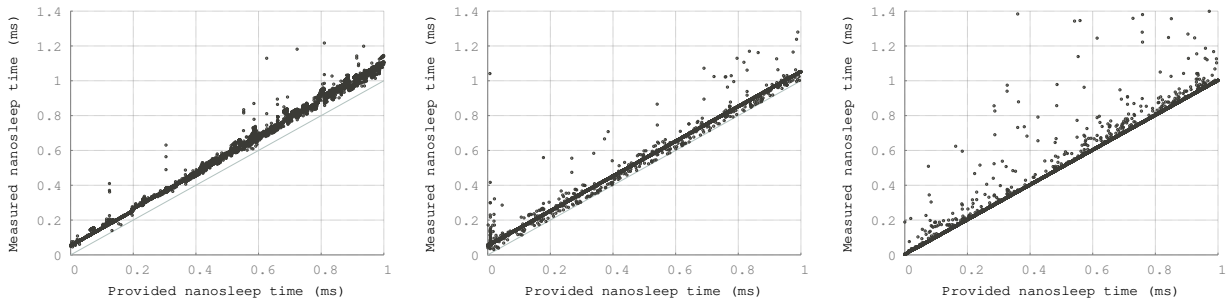
▷ **Experiment C: Benchmarking clock_nanosleep with different devices.** In this experiment, we measure the time that a process is put to sleep given different values for `clock_nanosleep` and different hardware/OS devices, as found in heterogeneous systems. Considered systems include a dual-core Intel Core i7 5600U (Laptop), a dual-core Arm Cortex A9 (Adapteva Parallella board), a 4-core Arm Cortex A53 (Raspberry Pi 3B+ board) and a 2x4-core big.LITTLE Arm Cortex A15/A7 (Odroid XU4 board). These devices run different Linux kernels from version 4.4, 4.6 to 4.14. The main difference resides in the kernel type: a standard kernel for the Intel i7 Core and the Raspberry Pi, and a low-latency kernel for Adapteva Parallella and Odroid XU4. In this low-latency kernel, also known as PREEMPT, the timer frequency (timer wheel, jiffies) is set to 1000Hz (instead of 250Hz for the standard kernel) to increase the responsiveness of the system, including when dealing with IRQs. This is however different from a real-time kernel (PREEMPT-RT) in which the system has to cope with hard deadlines. Therefore, we expect low-latency kernels to increase the accuracy of the sleep duration when calling `clock_nanosleep` compared to the standard kernel.

Sleep values given in Figure 4 range from 0ms to 1ms: 10 measures are performed every 500ns step, with a total of 20000 measures per figure. Figures on the left side run in idle mode, in which only the OS system and non-intensive services (less than 10% of the processor load) are deployed. Figures in the middle and right side run in busy mode, in which 8 threads overrun the processor cores by executing a loop onto nil.

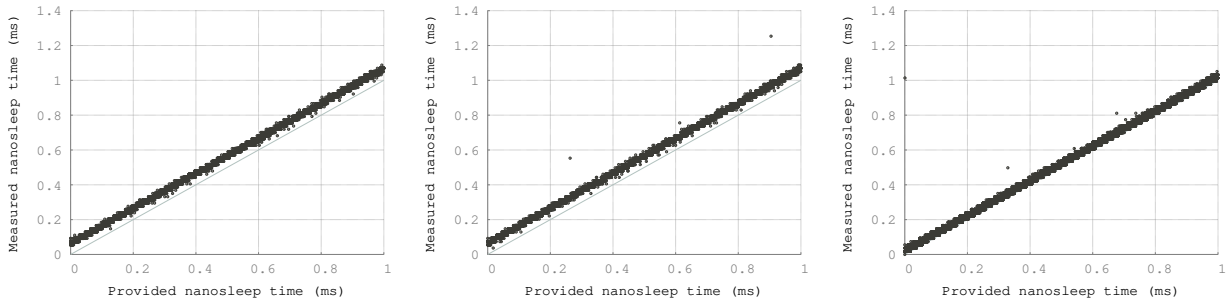
Results show that the `clock_nanosleep` implementation in all considered systems globally follows the ideal behavior, with slightly different resilience to processor load. Sleeping zero is not zero: there is an offset caused by 1) the processor cycles needed to measure the time before and after the `clock_nanosleep` instruction, 2) the OS kernel scheduler priority choice and 3) the adequacy between the hardware capacity and the OS kernel. In Linux, since kernel version 2.6.28, this offset can be significantly reduced by setting the `prctl PR_SET_TIMERSLACK` local value of the process to 1. Results using `prctl PR_SET_TIMERSLACK` are given on the right side of Figure 4 and show significant improvements regarding accuracy. However this workaround is OS-dependent, which is always a limitation when coping with distributed heterogeneous systems. The documentation also states that the timer slack is not applied to threads that are scheduled under a real-time scheduling policy.

Some noticeable behaviors include the accuracy of the wake up time for the Adapteva Parallella and Raspberry Pi 3B+ platforms. The Pi platform is less accurate in busy CPU load with values under $10\mu s$, probably due to the standard kernel compared to the Parallella low-latency kernel. There is also a quite relaxed result for the Odroid XU4 platform when coping with CPU load, even though it is a low-latency PREEMPT kernel.

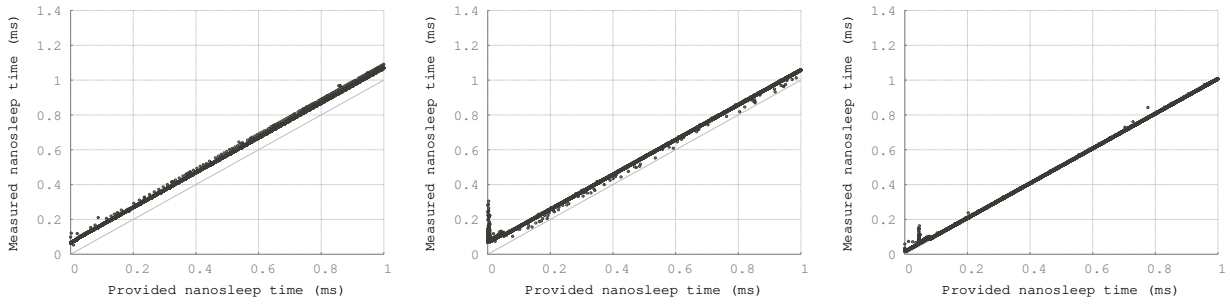
From these results, it appears that `clock_nanosleep` can be used to adapt the sleep time for each MPI process. However, the lack of accuracy for small yield times in most of non-RT systems advocates for a coarse tuning of the value. Furthermore, sleeping zero or very small sleep times still yields the process by an offset time, depending on the platform. This sleep duration is not only the result of the system subroutine call, but



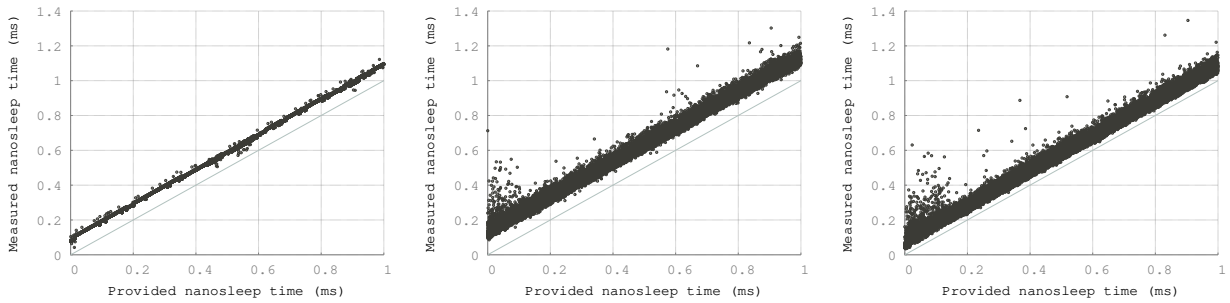
(a) (Idle) Intel Core i7 5600U, Linux 4.4 SMP (b) (Busy) Intel Core i7 5600U, Linux 4.4 SMP (c) (Busy) Intel Core i7 5600U, Linux 4.4 SMP, PR_SET_TIMERSLACK



(d) (Idle) Adapteva Parallella, Linux 4.6 SMP PREEMPT (e) (Busy) Adapteva Parallella, Linux 4.6 SMP PREEMPT (f) (Busy) Adapteva Parallella, Linux 4.6 SMP PREEMPT, PR_SET_TIMERSLACK

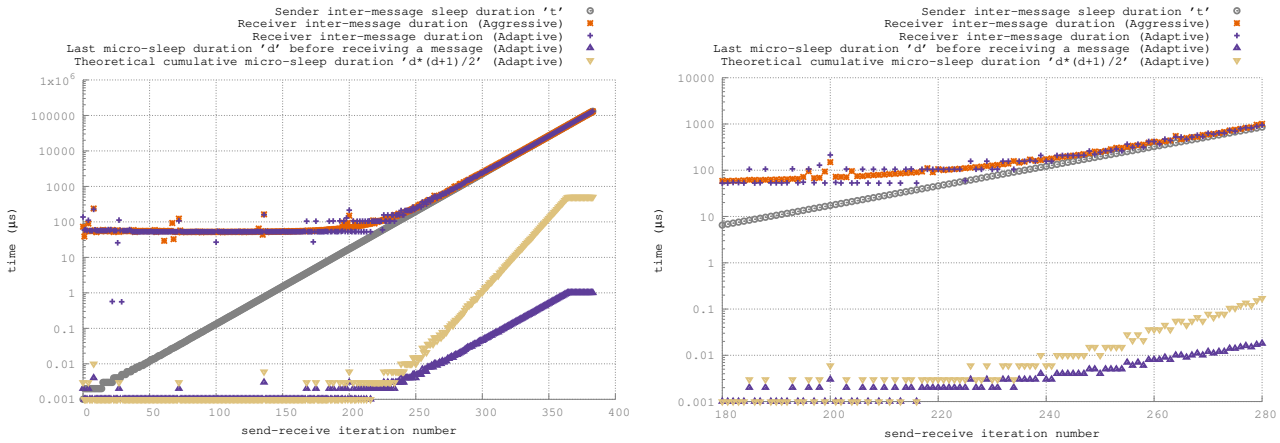


(g) (Idle) Raspberry Pi 3B+, Linux 4.14 SMP (h) (Busy) Raspberry Pi 3B+, Linux 4.14 SMP (i) (Busy) Raspberry Pi 3B+, Linux 4.14 SMP, PR_SET_TIMERSLACK



(j) (Idle) Odroid XU4, Linux 4.14 SMP PREEMPT (k) (Busy) Odroid XU4, Linux 4.14 SMP PREEMPT (l) (Busy) Odroid XU4, Linux 4.14 SMP PREEMPT, PR_SET_TIMERSLACK

Figure 4: Measuring clock_nanosleep yield time for different computing devices and configurations.



(a) All iterations, from burst to sparse message sendings

(b) Selected number of iterations to highlight micro-sleeping

Figure 5: Measuring the reaction time on message reception (busy-loop for the Aggressive mode and micro-sleeping for the Adaptive mode) using a simple send-receive pattern and an increasing sleep duration between two message sendings, from burst to sparse.

it also includes some unpredictable sleep time as in a thread context switch, even when sleeping zero. It is possible to tune the Linux scheduler priority using the `nice` root command. Setting a high priority might ensure a more accurate sleep time, but would also cause balance issues with colocated processes during wake up. It also seems that setting an erroneous negative value to `clock_nanosleep` does not only returns an error code, it also locally set the process priority higher in the scheduler as to meet a missed deadline. This can give quite interesting results for message bursts but should not even be considered as a solution because of the out-of-specs use of the function.

In short:	•`clock_nanosleep` globally follows the ideal behavior and is accurate enough to adapt the sleep time while waiting for an MPI message. •However, the lack of accuracy for small yield times in most of non-RT systems (e.g., Standard and low-latency Linux kernels) advocates for a coarse tuning of the value.

▷ **Experiment D: Comparing the reaction time on message reception with and without micro-sleeping.** In this experiment, we deploy a simple send-receive MPI program as in Experiments A and B (sections 2 and 3). We use OpenMPI 3.x and both processes are colocated on an Intel Core i5 1035G1 4-core processor to avoid uncertainties with network contention. The `send` process iterates 384 times on the following sequence in which it sends a small message (1024 bytes) to MPI process `recv` and sleeps a given amount of time t. Sleep duration t starts at 1ns and monotonically increases along with iteration i using formula $t_i = t_{i-1} * 1.05, i \in [1..384]$, which allows to evaluate burst and sparse sequences of message sendings, from 1ns to $\sim$ 130ms with a focus on burst sequence when t is small. The `recv` process receives messages and measures the time between two message receptions. In the case of the `Adaptive` mode, the last micro-sleep duration (that is, the last parameter used to call `clock_nanosleep` in the receive function) is also plotted for each iteration.

Figure 5 presents the results of the experiment, showing all iterations on Figure 5a while Figure 5b focuses on the iteration range where the micro-sleeping mechanism is starting to take over. The x-axis represents the iteration number. On the y-axis, given as a time duration in μs using a logarithmic scale, several measurements are represented: 1) the sender inter-message sleep duration t which strictly depends on the iteration number, 2) the measured receiver duration between two message receptions using the regular `Aggressive` mode, 3) the measured receiver duration between two message receptions using the `Adaptive` mode, 4) the last duration time given as a parameter to `clock_nanosleep` before receiving a message and 5) the theoretical cumulative sleep duration as explained thereafter.

The `Adaptive` mode is implemented according to the algorithm presented in Figure 3 and is configured using the following parameters: the minimum sleep duration is set to $d_{min} = 0$ns, the maximum sleep duration is set to $d_{max} = 1000$ns and the incremental step is set to $d_{step} = 1$ns. According to this algorithm and parameters, if the last sleep duration is d ($1 \leq d \leq d_{max}$) then the `Adaptive` receive function iterates at least d times on the MPI `Iprobe` and `clock_nanosleep` primitives using sleeping values incremented from 1 to d with step 1. In theory, with $d_{step} = 1$ns, the global sleep time is $\sum_{i=1}^{d} i = d * (d+1)/2$ if $d \leq d_{max}$ (using the triangular number formula) and $\sum_{i=1}^{d_{max}} i + n * d_{max}, n \in \mathbb{R}_+^*$ otherwise (because `clock_nanosleep` is thereafter

called using d_{max} once this maximum value is reached). In practice it is slightly different as already seen in Experiment C, depending on the accuracy allowed by the hardware, the OS and the software environment.

Figure 5a is decomposed into four sections. 1) From the first iteration to iteration 180, the inter-message sleep duration t on the `send` process is small (between 1ns and 10μs) in comparison with the time needed for the MPI runtime to cope with message delivery. Therefore, inter-message duration on the receiver are constant, around 52μs, for both `Aggressive` and `Adaptive` modes. There are a few exceptions in which the MPI runtime obviously takes a little more time to deliver the message (probably due to external reasons) and in that case the `Adaptive` mode triggers micro-sleep episodes. 2) From iteration 180 to 280 (see Figure 5b for more details) the sender sleep duration t outpaces the constant MPI runtime delivery time. The difference between the inter-message delivery duration on the receiver and the inter-message sleep duration on the sender can be interpreted as the *reaction time*. And this reaction time increases and stabilizes towards 200μs for both `Aggressive` and `Adaptive` modes (beware of the logarithmic scale, the reaction time is four times greater at iteration 280 than at iteration 180). At the same time, the micro-sleep duration starts to increase as t becomes significantly greater than the message delivery time. 3) From iteration 280 to 365 the reaction time is constant and, as the last observed micro-sleep duration d increases, the theoretical cumulative micro-sleep duration quickly converges to the receiver inter-message duration. 4) Finally, from iteration 365 to the end, d reaches d_{max} so that each further call to `clock_nanosleep` is kept under control and cannot dramatically put the `recv` process into sleep while an incoming message is made available by the MPI runtime. Note that the theoretical cumulative micro-sleep duration is not accurate in this fourth section as the triangular number formula is only valid for $d \leq d_{max}$.

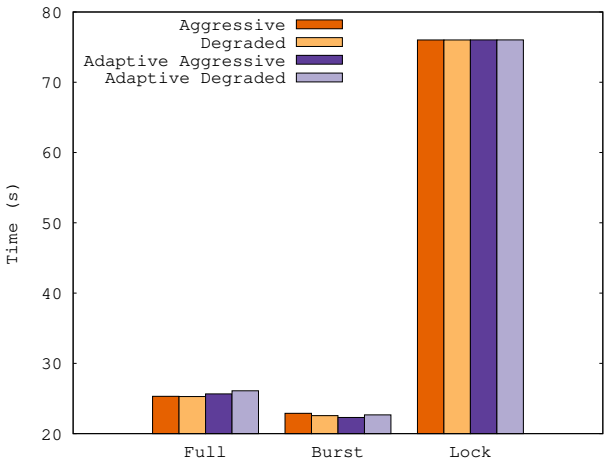
In short:
- No overhead is reported in this experiment using the `Adaptive` mode compared to the regular `Aggressive` mode.
- Once a receiver waits for a time that exceeds the MPI runtime message delivery time (not including communications), the micro-sleeping mechanism is taking over,
- The maximum sleep value prevents from putting the `recv` process into sleep while an incoming message is available.

In terms of process behavior, the `Adaptive` mode differs from the regular modes when the micro-sleep mechanism is activated. In Experiment D, the activation occurs roughly at iteration 220 when the time between two message delivery is $t \sim 40\mu$s. We refer to this particular time as $t_{activate}$. A second singular event, denoted as $t_{nominal}$, corresponds to the moment when the total micro-sleep duration converges and joins the receiver inter-message duration (not represented in this experiment). From $t_{nominal}$ and above, the micro-sleep mechanism is fully working: the process will be put to sleep the time it receives a new message *plus*, in the worst case, the maximum sleep value d_{max} which tends to be non-significant as t increases. The `Adaptive` mode is considered efficient if $t_{nominal} - t_{activate}$ is small, reaching the nominal state as soon as possible. $t_{activate}$ is platform-dependent: it is the direct consequence of the MPI runtime, the underlying system and hardware performances to deliver incoming messages. Therefore, $t_{activate}$ should be evaluated or predicted for each node in the computation. Conversely, $t_{nominal}$ is the result of the gradient induced by the `Adaptive` algorithm parameters d_{min}, d_{max} and d_{step}. The tuning of these parameters is not in the scope of this work, however it should be addressed either in a static way using heuristics or in a dynamic way at runtime. For the remaining of the paper the minimum sleep duration is set to $d_{min} = 0$ns, the maximum sleep duration is set to $d_{max} = 1000$ns and the incremental step is set to $d_{step} = 1$ns as in Experiment D. While these values have been set in a naive way, and that the minimum sleep duration and the incremental step are obviously small compared to the system capabilities, we can observe that it quickly increments to values that help in saving CPU computing time, which is a first demonstration for this contribution.

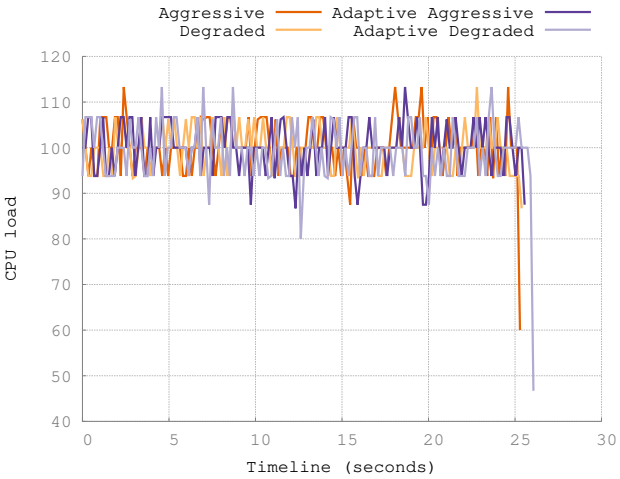
4 Adaptive Sleep Time in S-DSM

The `Adaptive` mode has been implemented within a Software-Distributed Shared Memory (S-DSM) that relies on the MPI runtime. S-DSM is a system that aggregates distributed physical memories into a global logical space. Applications are able to allocate and access shared data in this logical space from any node of the platform. Data localization and transfer are made transparent for the application and managed by the S-DSM runtime. Concurrent accesses are protected according to a coherence model. These systems have been introduced and studied from the late eighties [Li, 1988] for networks of workstations, clusters, computing grids and recent systems have been proposed for modern HPC architectures [Nelson et al., 2015, Capul et al., 2018, Kaxiras et al., 2015].

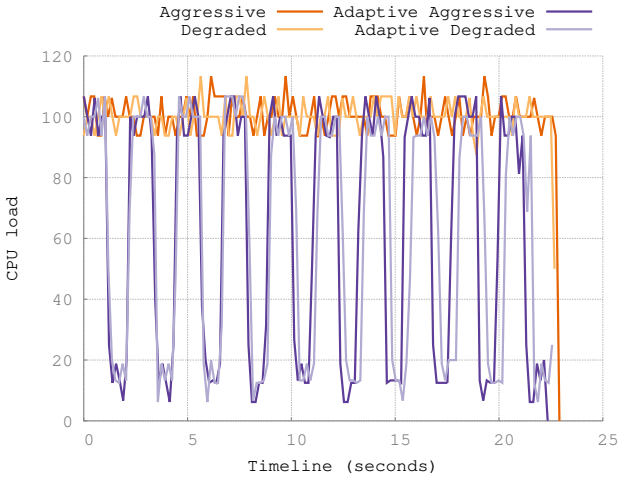
We consider the following S-DSM [Cudennec, 2017] in which coherence protocols are implemented as finite-state automata, with most of the transitions occurring on message reception. In this context, processes that run automata might spend a significant time waiting for messages, depending on the shared memory access patterns.



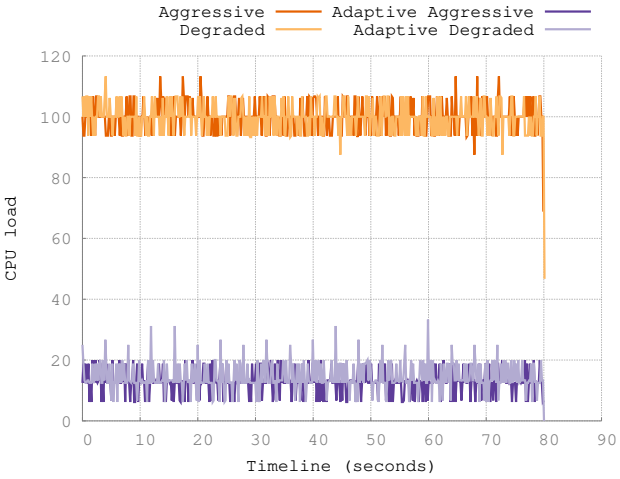
(a) Processing times



(b) (Full) CPU load (%CPU) of the S-DSM server



(c) (Burst) CPU load (%CPU) of the S-DSM server



(d) (Lock) CPU load (%CPU) of the S-DSM server

Figure 6: Synthetic processing times and CPU loads using OpenMPI modes (Aggressive and Degraded) and the Adaptive mode.

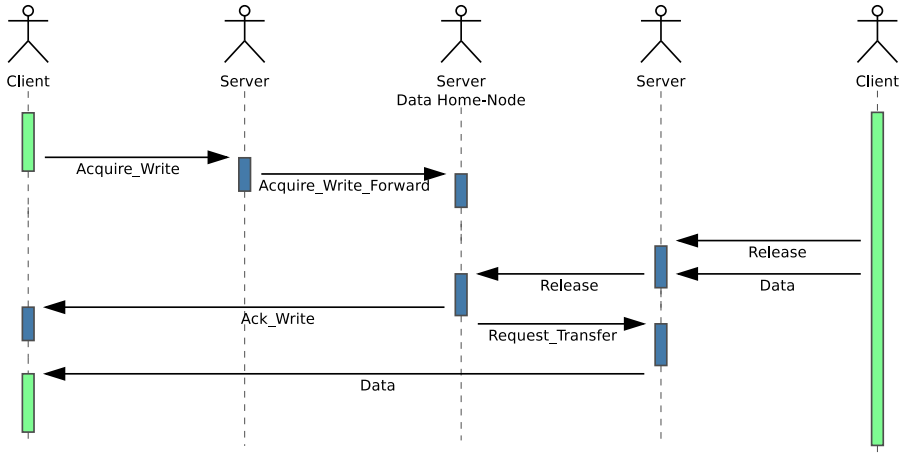


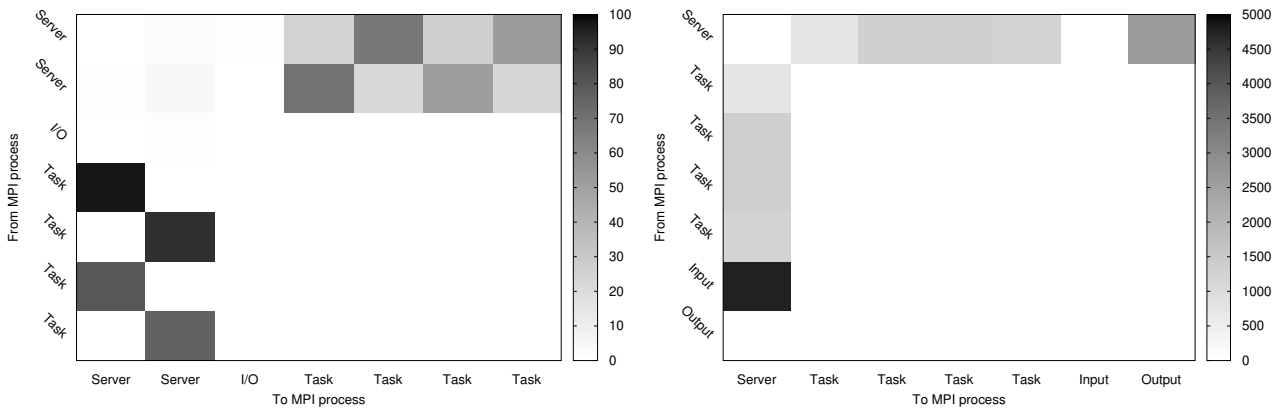
Figure 7: UML lifeline diagram for a write request in the S-DSM, illustration of the single-writer access mode (MRSW). Green events on the clients represent the user code while blue events are S-DSM runtime code. Dotted lines correspond to waiting times.

The S-DSM is organized as a semi-structured super-peer topology made of a peer-to-peer network of servers and a set of clients. Clients run the user code and provide the interface to the shared memory. Servers execute coherence automata and manage metadata. Both client and server processes are subject to hang for message reception. Clients wait for acknowledgment messages when performing synchronous operations such as data accesses and synchronization primitives. Servers wait for incoming requests and acknowledgment messages. Interestingly enough, we expect clients not to wait for messages in order to maximize the computing time (as in HPC systems), and servers to spend their time waiting for messages (not to waste computing resources). However, some clients might also rely on event programming as described in [Cudennec, 2018] and spend most of the time waiting for notifications. This S-DSM exhibits MPI process behaviors that are susceptible to waste CPU time and energy.

All MPI calls within the S-DSM are wrapped into a limited set of functions. This portable approach allows to switch to another runtime or to instrument the message passing codes. However, in the remaining of the paper we use the OpenMPI 3.x runtime because it offers a straightforward compilation and a smooth `mpirun` deployment over heterogeneous computing resources in terms of processors and operating systems. In order to implement the `Adaptive` mode, there are basically only two functions to modify: a function that receive any message from any process, and a function that receive any message from a given process. The code is very similar to the one presented in Figure 3 and can be activated or not at compile time. If not activated, the regular implementation is to use the synchronous `Probe` function. The `Degraded` mode is expected to deal with the scheduler in order to share resources with peer processes, and we suspect this is something less feasible if the process runs outside the MPI runtime, which is the case with the asynchronous `iProbe` implementation.

▷ **Experiment E: Reducing the S-DSM CPU load while maintaining the computing performance.** In this experiment, we measure the processing time and the CPU load for three synthetic applications running on top of the S-DSM. All three applications deploy three MPI processes: one S-DSM server and two clients that concurrently write the same shared data. Processes are deployed on a Raspberry Pi 3B+ (write), an Odroid XU4 (write) and a Core i7 machine (S-DSM server). The first application (*Full*) performs 20000 consecutive writes onto a shared data the size of an integer, with a total of 360000 small MPI messages (a mix of S-DSM data and protocol control messages delivered using MPI messages). The *Full* application generates an important network traffic in which processes do not have to wait for new messages. The second application (*Burst*) performs 10 cycles in which it writes 1000 times the shared data and sleeps 1 second, with a total of 180000 messages at the MPI level. The *Burst* application generates different S-DSM loads on the server, with periodic waiting for new access requests. Finally, the third application (*Lock*) performs 10 cycles in which it acquires the shared data write lock, sleeps 4 seconds and releases the lock, with a total of 188 messages at the MPI level. The *Lock* application generates heavy contention on shared data making the S-DSM server and the pending client wait for the release of the lock. For each application we compare the regular OpenMPI 3.x `Aggressive` mode, the `Degraded` mode, and a mix of both modes with the `Adaptive` mode.

The *Full*, *Burst* and *Lock* applications are simplistic and coarsely emulate classical access patterns to shared data. However, the inner mechanisms of the S-DSM, and more specifically the data coherence protocol lead to complex distributed automata with a mix of communications, processing and waiting times. In this work we consider a regular home-based 4-state MESI coherence protocol [Culler et al., 1999]. Home-based indicates that for each shared data a specific node (a participant in the coherence protocol) is in charge of the management



(a) Hough transform communication heatmap

(b) Videostream communication heatmap

Figure 8: Communication heatmaps. Scales are given in the total amount of data (MB) sent between two MPI processes.

of the associated metadata, including current data localization, replicas, versioning, current list of accesses and the list of pending requests. MESI stands for the 4 states that are used to describe the shared data: Modified, Exclusive, Shared and Invalid, which implements a multiple reader, single writer (MRSW) protocol. Figure 7 represents an UML lifeline diagram of the S-DSM processes in the case of a single write request, each agent being a MPI process. There are two clients: a first client asks for write permission and the other client is owning the write lock. We represent the case in which clients are connected to different S-DSM servers, that are also different from the S-DSM server in charge of the shared data (the home-node). This constitutes the worst case regarding the number of hops. Dotted lines on the diagram correspond to idle times in which the process is waiting for a message. On the client side, this waiting time depends on the ability of the S-DSM servers to grant access. This example shows that there is potentially a causal dependency with other clients that must release the lock prior to grant access to the requester. On the S-DSM server side, the computing complexity of the S-DSM code is very small and most of the time is spent managing communications. Therefore, the MPI process activity depends on the access pattern density, the coherence protocol and the number of S-DSM servers deployed to balance requests. This description of the S-DSM inner mechanisms shows that despite issuing simple access patterns at the user level, the resulting intricacy of communications, processing and idle times reveals to be complex, even compared to regular HPC communication patterns.

The results of this experiment are given in Figure 6 and the conclusions are twofold. First, from Figure 6a, there is no significant differences regarding the processing times for the three applications using the different MPI receive modes. Regular modes perform slightly better with the *Full* application and the `Adaptive Aggressive` mode performs slightly better with the *Burst* application. However, it is difficult to get into deeper conclusions at this point. Second, the CPU load is significantly different between the regular modes and the `Adaptive` mode. Figures 6b, 6c and 6d show the CPU load on the S-DSM server side during the execution time. The server process is running high on CPU for all modes with the *Full* application. For the *Burst* application, the `Adaptive` mode is able to drop the CPU use on the server each time the clients are sleeping 1 second. The regular modes keep polling for new messages even if clients are not requesting access to the shared data, and they do not yield in favor of another process or an idle state. Finally, application *Lock* exhibits very sparse accesses to shared data, which explains the CPU load on the server is kept very low using the `Adaptive` mode in comparison to the regular modes. These results are quite ideal because of the regular access patterns and the absence of computing tasks, in opposition to the applications used in the remaining of the work.

In short: •Using micro-sleeping in the MPI receive procedure exhibits significant drops in CPU load.
•These results are obtained using synthetic applications based on communication patterns conducive to highlight the expected gains of the approach.

5 Experiments on Heterogeneous Embedded Platforms

In the next experiments we deploy two real applications (Hough transform and Videostream) on top of the S-DSM in order to measure the benefits of using the `Adaptive` mode with more realistic access patterns and

MPI process behaviors.

Hough transform is an image processing application commonly used for line detection. It basically works in two steps. First, for each pixel of the input image, and above a given threshold, the Cartesian coordinates of the pixel are converted into polar coordinates and represented as a sinusoid in an intermediate image. Sinusoids that intersect lead to brighter pixels. Second, for each pixel of the intermediate image, and above a second threshold, polar coordinates are converted back into Cartesian coordinates by drawing a line using the angle and distance into the output image. We choose to implement this application over the S-DSM in a way it generates heavy contention on shared data. All images are stored as shared data in the S-DSM. The data granularity is set to the line of pixels [2]. Several tasks process the lines of the input image in a eager scheme and concurrently draw sinusoids to the intermediate image. The contention comes from the fact that drawing a sinusoid requires exclusive access to several lines of the shared intermediate image and generates false-sharing. This application is very demanding for the coherence protocols and the S-DSM spends most of the time dealing with access requests. The input image is a 256x256 grayscale bitmap with an intermediate 1800x724 image. We use a configuration with two S-DSM servers in order to balance access requests, one Input/Output task and 4 processing tasks. Each run generates around 1688000 messages at the MPI level. The communication heatmap is given in Figure 8a. There are four types of communication: 1) server to server: small control messages are sent for metadata management, 2) client to server: small control messages for access requests and large data commits after a write, 3) server to task: small control messages for acknowledgments and large data push when granting access and 4) client to client: no communications allowed in this application. From this heatmap, we expect the I/O task to benefit from the `Adaptive` mode by sleeping most of the running time.

Videostream is a video processing application that applies an edge detection followed by a line detection on each frame. Edge detection is implemented using a 3x3 convolution stencil. Line detection is achieved using a Hough transform implemented in Pthread [3]. For each processing task, input and output buffers are allocated in the S-DSM to store the input and processed frames. An input process is dedicated to the video decoding and the scheduling of frames onto input buffers. An output process is dedicated to the reordering of processed frames and the video encoding. This application is more CPU-demanding than the Hough transform because of the data granularity that is set to the frame size. The tasks spend more time in processing frames than accessing shared buffers because of the algorithm complexity and the data size. The input is a 1-minute video file, with a total of 1730 frames and a resolution of 1280x720 pixels (a raw frame is around 1MB). Processing a frame using Pthread takes around 0.2s on a Core i7 4700EQ, 0.9s on a Cortex-A15, 1s on a RPI 3B+ Cortex-A53 or Odroid XU4 Cortex-A15 and 2s on a Parallella Cortex-A9 [Olofsson et al., 2014] (we do not use the Parallella embedded 16-core processor because of the hybrid programming model it implies). We use a configuration with one S-DSM server, 4 processing tasks, one input/scheduling task and one reordering/output task. Each run generates around 52000 messages at the MPI level, which is far less than the Hough application (33 times). However, the total amount of transferred data is around 50 times larger for the Videostream application. The communication heatmap is given in Figure 8b. The processing tasks are CPU-demanding and we expect the server and the I/O tasks to benefit from the `Adaptive` mode because they are not much solicited. The implementation also relies on the publish-subscribe model which makes some tasks wait for an event, thus being good candidates for micro-sleeping.

The mapping for both applications is given in Table 1. Two platforms are considered: a sandbox composed by an Intel Core i7 computer, an Adapteva Parallella board, two Raspberry Pi 3B+ boards and an Odroid XU4 board connected via a Gigabit Ethernet network. The second platform is a RECS microserver [Griessl et al., 2014], a 1U rackable server that is composed of a backplane that provides power and networking capabilities to extension slots. Slots are filled with two Intel Core i7 boards and two COM Express boards, each board hosting 4 Apalis Arm Cortex-A15 boards. The RECS also provides access to power and temperature probes on each node. The configuration of the deployment consists in several steps including sizing the application (how many instances of each task), the logical topology (how instances are connected), the mapping of the instances onto the physical resources. In this work we consider an arbitrary configuration for each application: the optimization of the deployment is addressed in another contribution [Trabelsi et al., 2019].

▷ **Experiment F: Reducing the S-DSM CPU load with real applications.** In this experiment, we measure the CPU load, total processing time and sleep time for the Hough and Videostream applications comparing the different MPI receiving modes. We also evaluate two simple static modes that systematically sleep 0 and 100μs after each unsuccessful call to `Iprobe`. As presented in section 3, sleeping zero is not zero, and there is a significant yield time that depends on the hardware and the OS scheduler. The sleep duration of 100μs corresponds to the time over which there is less uncertainties on the effective sleep duration for the considered

[2] Granularity differs between input/output images and the intermediate image due to the different image size obtained by the coordinate transformation. This is however not a problem with the S-DSM as it is possible to allocate shared data of different sizes.

[3] This implementation is intended for (NUMA) shared memory computers, not distributed systems as the S-DSM implementation previously described in this work.

Application		Heter platform	RECS
Hough	Server	Core i7	
	Server	Parallella	
	I/O	RPI 3B+	
	Process	RPI 3B+	
	Process	Odroid XU4	
	Process	Core i7	
	Process	Core i7	
Videostream	Server	Core i7	Core i7 (B)
	Process	Parallella	A15
	Process	RPI 3B+	A15
	Process	RPI 3B+	A15
	Process	Odroid XU4	A15
	Input	Core i7	Core i7 (A)
	Output	Core i7	Core i7 (A)

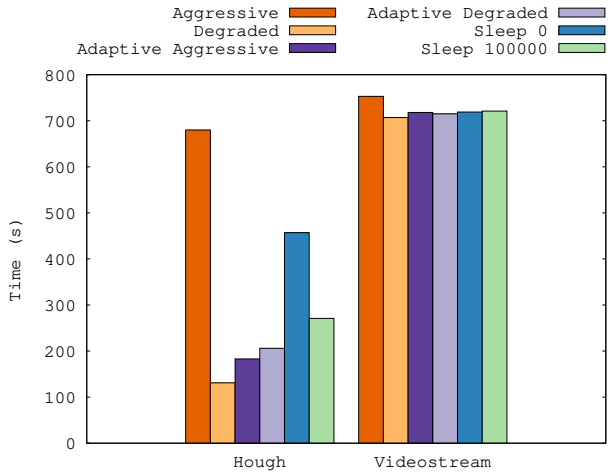
Table 1: Hough transform and Videostream applications mapping on the heterogeneous platform and the Christmann RECS|Box Antares Microserver. MPI processes that are colocated on the same computing device share the same colored background.

hardware as presented in Figure 4. Figure 9 shows several results from the execution of Hough and Videostream onto the heterogeneous platform.

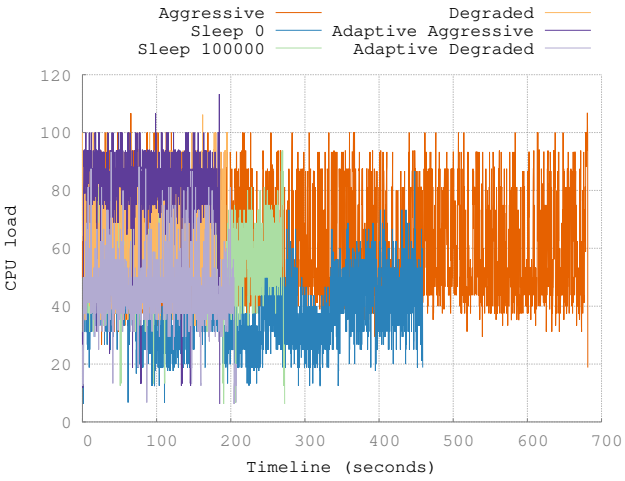
In Figure 9a, the Hough application performs differently when using different MPI receive modes. The regular `Aggressive` mode is noticeably slower than the other modes, especially compared to the regular `Degraded` mode. We explain this difference because of the colocation of the S-DSM server and two processing tasks onto the same Core i7 processor: if processing tasks are slowing down, then the other tasks have much work to do (the two other processing tasks are deployed on remote RPI and XU4) and if the server is slowing down then the overall system is slowing down as well. Conversely, using the `Adaptive Aggressive` mode is more efficient than the `Adaptive Degraded` mode. We think that the calls to `clock_nanosleep` already helps the OS scheduler to manage a more balanced use of the Core i7 between colocated tasks, and that the `Degraded` option does not add much. Finally, the static sleep modes are slower, and it seems that the scheduler discards the process longer when sleeping zero compared to sleeping a small amount of time. This can be observed in Figure 9b where the S-DSM server using static sleep-0 receive mode has less access to the CPU than the other modes.

Figure 9c shows the standard deviation of the CPU load for one S-DSM server, one processing task and the I/O task. The most important result is the ability of the `Adaptive Aggressive` mode to maintain a high CPU load for tasks that need responsiveness on message reception (S-DSM server and Processing task) while drastically dropping the CPU load for idle tasks (I/O task). Such a good strategy is not reproduced by other MPI receive modes. Figure 9e shows the processing time per task, decomposed into 4 parts: 1) the *Sleep time* is the total time spent in the `clock_nanosleep` function plus the measuring time (we are not able to avoid this at the user level), 2) the *Sync MP* corresponds to the time spent in the message passing receive primitive, excluding the sleep time. 3) The *S-DSM code* time corresponds to the local S-DSM data management. It is usually not significant, with some exceptions when dealing with low-performance processors (in this example we use the Parallella Cortex-A9 host processor, not the embedded manycore, which explains slower computing capabilities). 4) The *User code* corresponds to the time spent in the user code execution, excluding S-DSM calls. This figure confirms the important time spent by the I/O task sleeping during the execution, hence freeing the CPU for other duties. Colocated processing tasks running on the Core i7 processor also sleep significantly because the OS scheduler switches between them more frequently, putting one or the other into sleep. In this particular case, the turn-over might benefit to the S-DSM server that, in turn, improves the responsiveness of the S-DSM when coping with access requests to shared data, and decrease the waiting time (*Sync MP*) for processing tasks. One open-ended question is about the consequences on the global computing time and energy consumption of colocating specific tasks with different computing behaviors (or not). Such a question probably requires to explore different deployment configurations, build and train models to predict the efficiency of a given configuration, which is out of scope for this work.

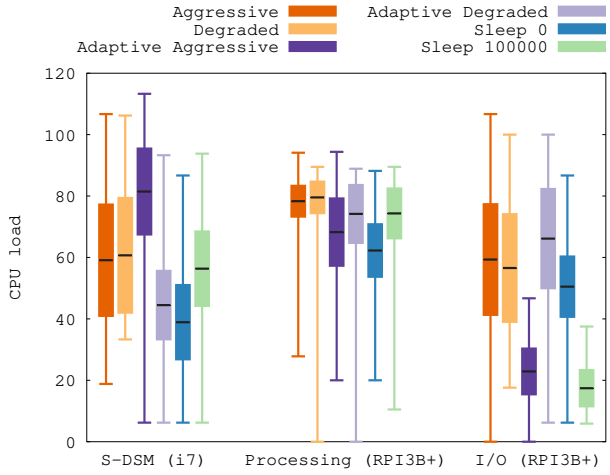
The task behaviors of the Videostream application are different from the Hough application, with far less activity on the S-DSM server and more time spent in user code for processing tasks. However, the same general conclusions can be applied to the Videostream application, as illustrated in Figures 9d. Using the `Adaptive` modes, both S-DSM server and I/O tasks significantly decreases the CPU use, while keeping a high CPU use for processing tasks. Figure 9f reveals that server and I/O tasks spend most of their time sleeping while the processing tasks are busy with frame processing, which appears to be a reasonable design for such a distributed



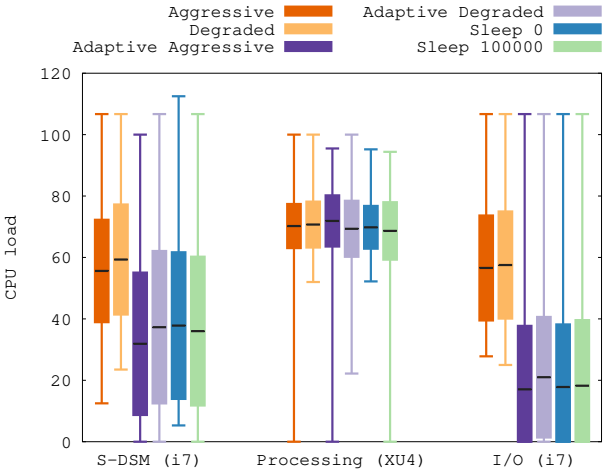
(a) Hough and Videostream processing times



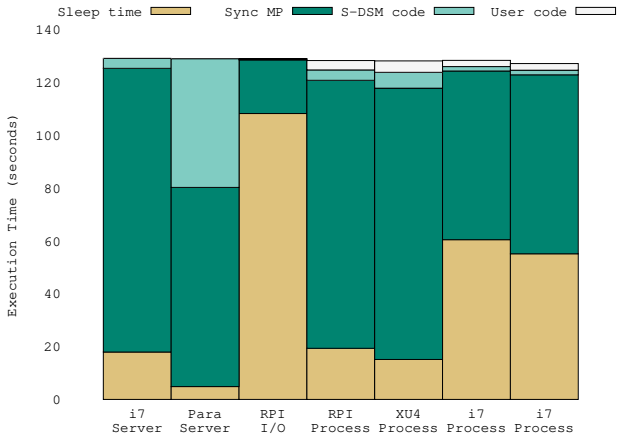
(b) Hough CPU load timeline (S-DSM server, Core i7)



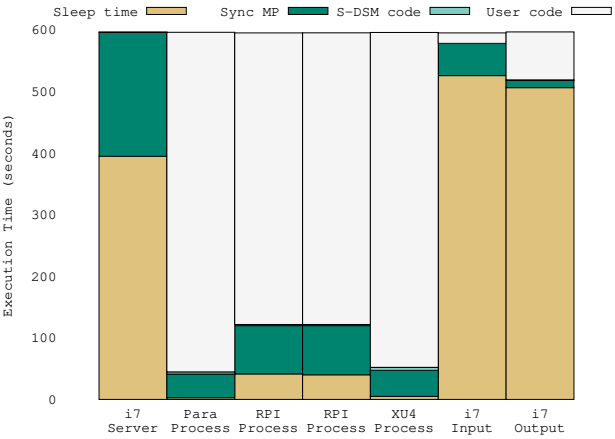
(c) Hough CPU load standard deviation



(d) Videostream CPU load standard deviation

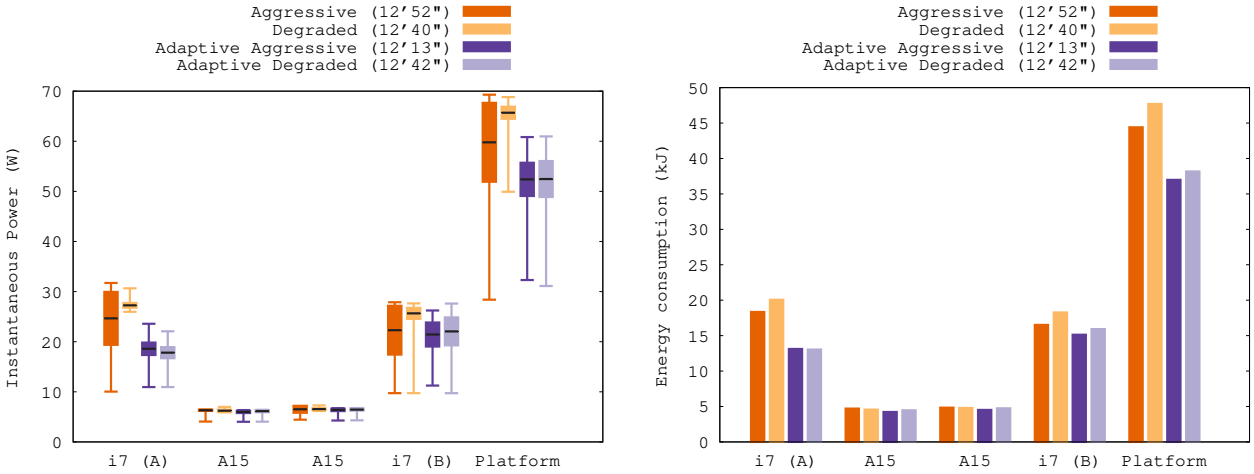


(e) Hough time decomposition per process (Adaptive Aggressive)



(f) Videostream time decomposition per process (Adaptive Aggressive)

Figure 9: Application processing times and CPU loads using OpenMPI modes (Aggressive and Degraded) and the Adaptive mode.



(a) Standard deviation for the instantaneous power consumption.

(b) Energy consumption in kJ.

Figure 10: Instantaneous power and energy consumption for Videostream on the Christmann RECS|Box Antares Microserver. Only two out of the four Cortex-A15 nodes used in these experiments are represented, for the sake of clarity, and because these nodes exhibit similar results as they run similar tasks. Results for the platform are calculated using these two Cortex-A15 nodes: results for the full platform, including the four Cortex-A15 nodes, show a global offset for all MPI polling modes due to the two additional nodes, which leads to the same conclusions.

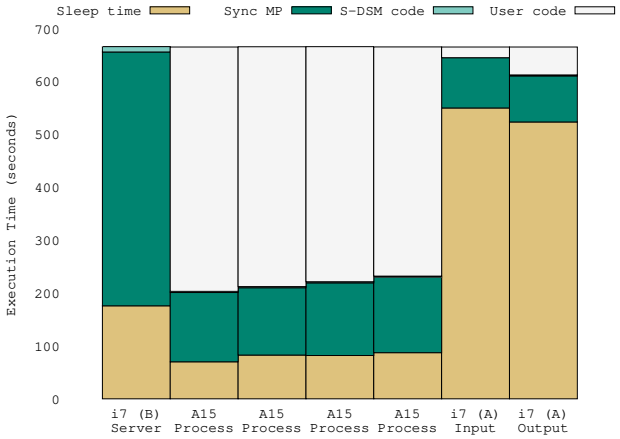


Figure 11: Execution time decomposition per process (Adaptive Degraded).

application. The `Adaptive` modes, while reducing the CPU load, do not sacrifice performances as shown in Figure 9a. Both modes perform better than the regular `Aggressive` mode and appear slightly behind the `Degraded` mode. Interestingly enough, the static sleep modes compete with the `Adaptive` mode because of the more relaxed communication patterns in Videostream compared to the Hough application (there are no bursts of messages, and there is no need to be very responsive on message reception).

These experiment tend to demonstrate that the `Adaptive` mode can significantly reduce the CPU load compared to regular MPI receive modes. The influence on performances is however complex to determine because it depends on the application communication pattern and the task colocation. In the next section, we analyze the influence on the energy consumption.

In short:	•Introducing micro-sleeping does not severely decrease the performance of the applications. When colocating tasks on processors, it even competes with the vanilla `Degraded` mode. •Using the `Adaptive` modes gives significant improvements regarding the CPU use which opens reasonable expectations to tune the performance/energy efficiency.

	TIME	FPS	W	kJ	FPS/W
Aggressive	12'52	2.23	60	44.39	0.037
Degraded	12'40	2.26	66	47.68	0.034
Adaptive Aggressive	12'13	2.35	52	36.98	0.045
Adaptive Degraded	12'42	2.26	52	38.16	0.043

Table 2: Total compute time, frames per second (FPS), mean instantaneous power consumption (Watt), total energy consumption (kJ) and frames per second per Watt (FPS/Watt) for the Videostream application on the Christmann RECS|Box Antares Microserver.

6 Energy Consumption

Measuring the energy consumption of a computing device follows different rules than measuring the CPU load for a process. There is not a strong correlation between the MPI process activity and the instantaneous power consumption. First, if no MPI process is using the CPU, other system or user processes can use it and the power consumption might not decrease. Second, the power consumption of the CPU is the consequence of power management techniques such as clock gating (removing the clock signal when the circuit is not used) and dynamic frequency voltage scaling (DVFS, increasing and decreasing the voltage depending on the processor use). Slowing down the processor to achieve energy savings is not trivial as it has to cope with the user feedback and tolerance regarding the system speed as discussed in [Halimi et al., 2013]. Therefore, even if the CPU is not overloaded, the system might not take the decision to slow down because of the prediction algorithm. Furthermore, peripherals and other I/O equipment might interfere in the measurement such as integrated network switches as discussed in [Schlitt and Nebel, 2016]. However, decreasing the CPU load globally leads to energy savings and the micro-sleep mechanism is expected to contribute in that way.

The nature of the platform used in previous experiments (a collection of heterogeneous computers and development boards) does not allow to monitor the energy consumption in an accurate and simple way: this would require external monitoring of some of the boards, using electronic probes plugged between the power supply and the board. In this section, we use the Christmann RECS micro-server. It is an industrial-grade hardware partly designed within European projects FiPS [Griessl et al., 2014] and M2DC [Oleksiak et al., 2017] for HPC and HPEC applications. It allows to monitor all nodes using the same type of probes, which is important for comparisons and statistics. We frequently measure the instantaneous power consumption of each processor, as well as the whole platform. To achieve this, a thread is created on the S-DSM server that periodically requests power consumption values to the RECS frontend using the provided REST interface. This interface allows to read values roughly every 180ms, which is small enough in comparison with the time spent for each run: more or less 4000 measures are logged for a 12-minute run. We write these values into a file together with timestamps. From these discrete instantaneous power consumption measures, and according to the timestamps, we calculate the global energy consumption in kJ per node corresponding to the application running time.

▷ **Experiment G: Reducing the S-DSM power consumption.** In this experiment, we deploy the Videostream application onto the RECS micro-server. This application exhibits three different process behaviors: 1) the processing tasks deployed on the Cortex A15 nodes are kept busy because the processing time is greater than the time to access the next frame. 2) the I/O tasks colocated on a Core i7 processor take little processing time to decode, deliver and encode frames compared to the demand from the processing tasks. Therefore, they spend most of their time waiting for messages. 3) The S-DSM server is located on a different Core i7 processor and manages all communications related to every access to shared data in the S-DSM, including control messages, metadata and data. Most of the time is spent dealing with access requests which leaves little room for micro-sleeping.

Results are given in Figure 10, showing the instantaneous power consumption in Figure 10a and the energy consumption in Figure 10b. The Core i7 processor usually spends around 10 W in idle mode and 30 W when busy. The Cortex-A15 nodes spend 3 W each when idle and 7 W when busy. There are two main improvements when using the `Adaptive` mode: first, the maximum instantaneous power consumption peak is reduced, which is important for battery-operated systems as discussed in [Furset and Hoffman, 2011] and second, the standard deviation and the mean instantaneous values for energy consumption are also reduced using the `Adaptive` mode, which advocates for a better battery life if we consider the basic equation:

$$Battery\_lifetime(h) = Battery\_capacity(mAh)/Average\_current\_consumption(mA)$$

The most important power savings are achieved using the `Adaptive` modes on Core i7 (A) onto which have been colocated both input and output tasks. This is explained in Figure 11 with a significant sleep time for both tasks. There are also some small improvements on Core i7 (B) and very small improvements on Cortex-A15

(beware of the scale, there is a factor 5 between Core i7 and Cortex-A15 power consumption). Furthermore, Cortex-A15 processors only run user computing codes that are not prone to wait for messages, hence not being good candidates for micro-sleeping. One rather interesting result is the computing time (given in Figure 10 and Table 2): the `Adaptive Aggressive` mode is performing better than the other modes.

From these results we calculate some performance/energy metrics that are given in Table 2. This reveals that the `Adaptive Aggressive` mode outperforms the other modes in all domains: frames per second, total energy spent (kJ) and performance-per-Watt. This is quite an unexpected result because configuring an application usually leads to a trade-off between performance and energy: there does not exist one solution that fulfills all constraints, but rather a Pareto front into which it is possible to choose a solution that fits to the current goals and objectives [Friese et al., 2013]. We would like to alleviate the conclusions about the performance increase using the `Adaptive` mode, as only one application has been tested onto one testbed. However, these results appear promising to us and were repeatable.

In short: •The `Adaptive` modes are able to decrease the instantaneous power consumption of the Videostream application up to 22% compared to classical modes.
•Some speed-up (6%) can also be observed due to a better management of colocated tasks.
•The combination of energy savings and application speed-up leads to better results regarding the total energy spent in kJ.

From the experiments conducted in this work, and as a wrap-up for this contribution, we propose a synthesis table in Table 3. The table is composed by three criteria and a conclusion: 1) "*Wait for messages*" indicates whether some processes are prone to wait for incoming messages and have nothing else to do in this situation. 2) "*Process colocation*" indicates whether two or more processes have to share a computing resource. 3) "*Primary goal*" indicates if the running context is seeking computing performance or energy savings. 4) "*Suggested mode*" indicates what MPI receive mode is the most relevant for the given combination of criteria. Only three combinations of criteria are given here as there are no elements in the experiments to decide for other situations. One of the most important result for the adoption of MPI-based applications within the HPEC context is that the `Adaptive` mode is an efficient approach in presence of sparse communications, especially when dealing with loosely coupled processes such as I/O and services that are occasionally solicited.

Wait for messages	Process colocation	Primary goal	Suggested mode
Never	No	Performance	Aggressive
Never	Yes	Performance	Degraded
Sometimes	*	Energy consumption	Adaptive Aggressive

Table 3: Suggested MP *receive* mode depending on the application and deployment context.

7 Related Work

As early as in 2008, a similar work has been conducted within the MPC runtime [Pérache et al., 2008] that is specialized in the colocation of tasks for HPC NUMA nodes. MPC provides a unified runtime for distributed and shared memory with a shift of semantics: an MPI process becomes a MPC thread. The runtime defines an integrated polling method that avoids busy-waiting. This allows the integrated scheduler to fairly balance threads over the same computing nodes. Unfortunately our experiments with MPC were not conclusive and we were not able to observe the expected CPU and energy savings using the default installation. OpenMPI [Gabriel et al., 2004] and MPICH [Thakur and Gropp, 2007] provide mechanisms to support the oversubscription of MPI processes over the same computing node. However their runtime schedulers are designed to yield to other MPI processes and not to solve the issue of busy-waiting for incoming events. Another energy-aware MPI runtime implementation proposed in 2015 is the EAM system [Venkatesh et al., 2015], also referred as MVAPICH2-EA [MVA,] and based on MVAPICH2. It uses a communication model to predict the time to send data and decide whether or not to select a different power level. It relies on advanced network functionalities to manage interrupts. Experiments are conducted on the Infiniband high-speed network interconnect, which is more adapted to HPC hardware than HPEC. The energy consumption aspect is monitored by a tool that relies on the Intel Sandy Bridge technology, therefore not being a good candidate for heterogeneous systems. Results show that energy savings are up to 41% with a performance loss not exceeding 5% using popular HPC applications and a deployment of up to 4096 MPI processes. These are undoubtedly promising results for saving energy in HPC.

In this work we try to demonstrate that these concepts are well adapted for embedded heterogeneous computing architectures, with a focus on energy consumption. We propose and evaluate a lightweight tuning of the *receive* function that can be transparently implemented at the user level on top of any MPI runtime, which is convenient when deploying over heterogeneous systems.

▷ **Energy Efficient Ethernet.** Energy consumption for message passing is also studied in network hardware and transport layers, as with the Energy Efficient Ethernet (EEE, IEEE 802.3az) that specifies a sleep mode for the transmitter when no data is sent. Several works are conducted based on EEE. In [e Silva and Carpenter, 2015], the authors propose a tight configuration of packet coalescing over EEE to increase micro-sleeping. Experiments are conducted with Hadoop applications and results show energy savings but at the price of delays and bursts that affect performance. In [Haudebourg and Orgerie, 2017], the authors compare the efficiency of two communication strategies involving micro-sleeping (LPI) and the possibility to slow down transmitters (ALR) as with DVFS for CPU. Results show that combining both LPI and ALR gives better energy savings. In [Saravanan et al., 2015] the authors introduce different sleep modes: fast-wake and deep-sleep modes in which the communication link is not completely off: it periodically refreshes and awaits frames to wake-up. Finally, in [Cenedese et al., 2017] the authors expand EEE with prediction (EEEP), using statistical analysis of the current traffic for predicting forthcoming traffic.

▷ **MPI Management Frameworks.** Power management can be monitored and optimized at the platform and runtime levels. In the GEOPM project [Eastep et al., 2017], MPI applications are deployed under the supervision of a hierarchical system that aims at building a trade-off between performance and energy consumption. In this approach, the system is able to dynamically remap some MPI jobs onto computing resources. Monitoring the MPI runtime can also be used to apply dynamic spawning of processes, also called dynamic malleability, depending on performance and energy consumption constraints as in [Rodriguez-Gonzalo et al., 2016]. Green Queue [Tiwari et al., 2012] is also able to dynamically adapt the processor voltage and frequency using DVFS to slow down a processor running an MPI process. However, this requires an offline application characterization step using profiling tools in order to determine the DVFS values based on a power model. Processor frequency and voltage can be adapted for each MPI communication phase as proposed in this work [Lim et al., 2006]. It uses a dynamic training system to identify the phases and a shifting system to change the `p-state` of the processor. In this work [Howard et al., 2011], a multi-core processor is proposed with a specific management of on-die message passing with DVFS for power scaling. While not being directly related to MPI, this latter work addresses the problem of energy consumption at the platform scale when dealing with message passing.

▷ **Trade-off between Performance and Energy Consumption.** Some research efforts are made to build a trade-off between computing performance and energy consumption when configuring distributed applications. These methods include heuristics based on genetic algorithms [Friese et al., 2013], design space exploration [Gadioli et al., 2015], static scheduling optimization [Zaourar et al., 2018] and local search [Trabelsi et al., 2019]. The latter work is applied to distributed applications relying on the OpenMPI runtime and deployed onto heterogeneous machines. In all these systems, the application is monitored and a configuration is put into a Pareto front according to the energy consumption that has been measured. Hence, this measure is important to determine if the configuration is part of the Pareto front (non-dominated) or not. In this work, we show that popular MPI implementations generate an important CPU overhead when polling messages, leading to maximum load onto any CPU hosting a MPI process, even if it is only waiting for messages. These MPI implementation do not allow to finely evaluate the expected behavior of the user code and runtime activity. With the proposed `Adaptive` polling method, MPI processes that are deployed with an exclusive access to a node can decrease the global CPU load, which in turn enhances the evaluation of the solution regarding the energy consumption, and even promote it within the Pareto front.

▷ **MPI Simulation and Power Models.** Whenever it is not possible to conduct experiments on a platform, the MPI runtime can be simulated and some power models can help characterize the energy consumption. In [Bielert et al., 2015] the authors propose a simulator for several programming paradigms including MPI. It uses an input trace generated from the instrumented code of the application and outputs the total energy consumption using an energy model adapted for each computing node. In [Heinrich et al., 2017] the authors propose quite a similar approach with the SimGrid toolkit in an HPC context. They highlight the difficulty of characterizing the MPI `Iprobe` function to model the power consumption in case of a repeated polling. This is close to the original question that led to this work.

8 Conclusion

Message passing runtimes are commonly used in HPC systems and a large number of distributed applications rely on them. We think that HPC applications can be used in the High-Performance Embedded Computing (HPEC) context without code modification. This requires to keep the underlying MP interface. In this work we have discussed the fact that implementing the receive function with a busy-wait loop as it is in MP runtimes for HPC is not adapted to HPEC, mainly because of the waste of energy it generates, and the poor resource sharing management that occurs when colocating processes. By slightly modifying the receive procedure with micro-sleep episodes, which can be done as a small transparent tuning for the application and the runtime, we obtain significant CPU load and energy savings. When colocating processes, we are also able to get speed-up. The combination of energy savings and application speed-up leads to better results regarding the total energy spent in kJ, which is a valuable metric for super-computing centers and embedded devices.

There are several direct follow-ups for this work. First, the `Adaptive` algorithm is configured using three parameters: minimum and maximum sleep duration and the incremental sleeping step. These parameters need to be finely tuned to reach the nominal working state in which the process is put to sleep as long as possible. The main issue is to find a trade-off between sleeping for longer periods to save computing resources and energy, while not increasing the reaction time when receiving message as it would result in a loss of computing performance and might also spend more energy at the end. These values can be determined offline using heuristics for example: for each computing resource onto which a MPI process is deployed, design space exploration can be used to find local optimum, build models and predict a configuration for given constraints in terms of performance and energy consumption. This would result in a Pareto front from which it is possible to pick a configuration that fits to the context. Another possibility is to rely on dynamic tuning in which the parameters are modified online according to a set of rules and thresholds in order to maintain some performance goals. This dynamic tuning can be decided locally on each site or can be the result of a distributed decision, using neighbor gossiping for example. Dynamic tuning is also well adapted for HPEC as the environment might change: if a volatile resource based on opportunistic connections vanishes, this results on more computing demand on the remaining resources, if a resource faces a depleted battery it suddenly requires more energy savings to continue working.

A second perspective is to explore different configurations of micro-sleeping for the deployment of applications with a focus on process colocation. Deploying computing tasks on the same resource brings advantages and drawbacks: some processes might benefit from using the physical shared memory to communicate with colocated processes instead of sending messages to remote nodes over the network. Conversely, some processes might have difficulties to efficiently access the CPU when colocated with other CPU-greedy processes. There is a trade-off to find between shortening communications and scaling with more distant processors. By introducing micro-sleeping episodes, this trade-off might shift in favor of colocating processes, provided these processes are prone to busy-wait for incoming events.

A third perspective to this work is the implementation of the micro-sleep strategy within different MPI runtimes and the evaluation using well-established benchmarks. Benchmarks such as the MPI version of the NAS Parallel Benchmark (NPB) usually come from the HPC context. We have discussed in this work the fact that HPC applications do not usually exhibit MPI processes that wait for incoming messages, neither the colocation of processes onto the same resource, therefore not being good candidates for micro-sleeping. However, this is probably drawing a hasty conclusion: in some complex simulation codes, the MPI process is multi-threaded, with a particular thread dedicated to I/O and MPI interfacing. This thread is de facto colocated with other threads on the same resource as it belongs to the same process. It is also probably used to synchronize the other working threads with sporadic control and data messages, therefore waiting for incoming events most of the time. This kind of thread might benefit from the micro-sleep mechanism so as not to use a full processing core when there is no need. As a result, some energy savings or improvements in computing performance could be obtained for HPC applications as well.

To conclude, this work advocates for developing and adopting energy-aware MPI runtimes in the future, as a standard implementation, relieving the user from tuning its own application. Many MPI runtime implementations have been proposed and are still in active development. Each version comes with new functionalities and improvements. However, this also leads to an array of compilation and running options in order to finely tune the runtime behavior, especially when seeking for specific optimizations. This should change in the future in order to offer a better management of energy consumption as it has now become a major constraint alongside computing performance and a conditional acceptance of MPI-based applications within HPEC systems.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 688201. The author would like to thank the reviewers for their thoughtful comments and efforts towards improving this manuscript.

References

[EEP,] Energy efficient probe for mpi (eeprobe). https://github.com/lcudenne/eeprobe/.

[MVA,] Mvapich2-ea 2.1. https://mvapich.cse.ohio-state.edu/userguide/ea/.

[Bielert et al., 2015] Bielert, M., Ciorba, F. M., Feldhoff, K., Ilsche, T., and Nagel, W. E. (2015). HAEC-SIM: a simulation framework for highly adaptive energy-efficient computing platforms. In Theodoropoulos, G., editor, *Proceedings of the 8th International Conference on Simulation Tools and Techniques, Athens, Greece, August 24-26, 2015*, pages 129–138. ICST/ACM.

[Breslow et al., 2016] Breslow, A. D., Porter, L., Tiwari, A., Laurenzano, M., Carrington, L., Tullsen, D. M., and Snavely, A. (2016). The case for colocation of high performance computing workloads. *Concurr. Comput. Pract. Exp.*, 28(2):232–251.

[Capul et al., 2018] Capul, J., Morais, S., and Lekien, J. (2018). Padawan: a python infrastructure for loosely coupled in situ workflows. In Wolf, M., Moreland, K., and Bethel, E. W., editors, *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, ISAV@SC 2018, Dallas, TX, USA, November 11, 2018*, pages 7–12. ACM.

[Cenedese et al., 2017] Cenedese, A., Tramarin, F., and Vitturi, S. (2017). An energy efficient ethernet strategy based on traffic prediction and shaping. *IEEE Trans. Communications*, 65(1):270–282.

[Cudennec, 2017] Cudennec, L. (2017). Software-distributed shared memory over heterogeneous micro-server architecture. In Heras, D. B., Bougé, L., Mencagli, G., Jeannot, E., Sakellariou, R., Badia, R. M., Barbosa, J. G., Ricci, L., Scott, S. L., Lankes, S., and Weidendorfer, J., editors, *Euro-Par 2017: Parallel Processing Workshops - Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers*, volume 10659 of *Lecture Notes in Computer Science*, pages 366–377. Springer.

[Cudennec, 2018] Cudennec, L. (2018). Merging the publish-subscribe pattern with the shared memory paradigm. In Mencagli, G., Heras, D. B., Cardellini, V., Casalicchio, E., Jeannot, E., Wolf, F., Salis, A., Schifanella, C., Manumachu, R. R., Ricci, L., Beccuti, M., Antonelli, L., Sánchez, J. D. G., and Scott, S. L., editors, *Euro-Par 2018: Parallel Processing Workshops - Euro-Par 2018 International Workshops, Turin, Italy, August 27-28, 2018, Revised Selected Papers*, volume 11339 of *Lecture Notes in Computer Science*, pages 469–480. Springer.

[Culler et al., 1999] Culler, D. E., Singh, J. P., and Gupta, A. (1999). *Parallel computer architecture - a hardware / software approach*. Morgan Kaufmann.

[e Silva and Carpenter, 2015] e Silva, R. F. and Carpenter, P. M. (2015). Exploring interconnect energy savings under east-west traffic pattern of mapreduce clusters. In Kanhere, S., Tölle, J., and Cherkaoui, S., editors, *40th IEEE Conference on Local Computer Networks, LCN 2015, Clearwater Beach, FL, USA, October 26-29, 2015*, pages 10–18. IEEE Computer Society.

[Eastep et al., 2017] Eastep, J., Sylvester, S., Cantalupo, C., Geltz, B., Ardanaz, F., Al-Rawi, A., Livingston, K., Keceli, F., Maiterth, M., and Jana, S. (2017). Global extensible open power manager: A vehicle for HPC community collaboration on co-designed energy management solutions. In Kunkel, J. M., Yokota, R., Balaji, P., and Keyes, D. E., editors, *High Performance Computing - 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18-22, 2017, Proceedings*, volume 10266 of *Lecture Notes in Computer Science*, pages 394–412. Springer.

[Friese et al., 2013] Friese, R., Khemka, B., Maciejewski, A. A., Siegel, H. J., Koenig, G. A., Powers, S., Hilton, M., Rambharos, J., Okonski, G., and Poole, S. W. (2013). An analysis framework for investigating the trade-offs between system performance and energy consumption in a heterogeneous computing environment. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 19–30. IEEE.

[Furset and Hoffman, 2011] Furset, K. and Hoffman, P. (2011). High pulse drain impact on cr2032 coin cell battery capacity. Technical report, Nordic Semiconductor and Energizer.

[Gabriel et al., 2004] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: goals, concept, and design of a next generation MPI implementation. In Kranzlmüller, D., Kacsuk, P., and Dongarra, J. J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer.

[Gadioli et al., 2015] Gadioli, D., Palermo, G., and Silvano, C. (2015). Application autotuning to support runtime adaptivity in multicore architectures. In Soudris, D. and Carro, L., editors, *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2015, Samos, Greece, July 19-23, 2015*, pages 173–180. IEEE.

[Griessl et al., 2014] Griessl, R., Peykanu, M., Hagemeyer, J., Porrmann, M., Krupop, S., vor dem Berge, M., Kiesel, T., and Christmann, W. (2014). A scalable server architecture for next-generation heterogeneous compute clusters. In *12th IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2014, Milano, Italy, August 26-28, 2014*, pages 146–153. IEEE Computer Society.

[Halimi et al., 2013] Halimi, J., Pradelle, B., Guermouche, A., Triquenaux, N., Laurent, A., Beyler, J. C., and Jalby, W. (2013). Reactive DVFS control for multicore processors. In *2013 IEEE International Conference on Green Computing and Communications (GreenCom) and IEEE Internet of Things (iThings) and IEEE Cyber, Physical and Social Computing (CPSCom), Beijing, China, August 20-23, 2013*, pages 102–109. IEEE.

[Haudebourg and Orgerie, 2017] Haudebourg, T. and Orgerie, A. (2017). On the energy efficiency of sleeping and rate adaptation for network devices. In Ibrahim, S., Choo, K. R., Yan, Z., and Pedrycz, W., editors, *Algorithms and Architectures for Parallel Processing - 17th International Conference, ICA3PP 2017, Helsinki, Finland, August 21-23, 2017, Proceedings*, volume 10393 of *Lecture Notes in Computer Science*, pages 132–146. Springer.

[Heinrich et al., 2017] Heinrich, F. C., Cornebize, T., Degomme, A., Legrand, A., Carpen-Amarie, A., Hunold, S., Orgerie, A., and Quinson, M. (2017). Predicting the energy-consumption of MPI applications at scale using only a single node. In *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*, pages 92–102. IEEE Computer Society.

[Howard et al., 2011] Howard, J., Dighe, S., Vangal, S. R., Ruhl, G., Borkar, N., Jain, S., Erraguntla, V., Konow, M., Riepen, M., Gries, M., Droege, G., Lund-Larsen, T., Steibl, S., Borkar, S., De, V. K., and der Wijngaart, R. F. V. (2011). A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling. *J. Solid-State Circuits*, 46(1):173–183.

[Kaxiras et al., 2015] Kaxiras, S., Klaftenegger, D., Norgren, M., Ros, A., and Sagonas, K. (2015). Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In Kielmann, T., Hildebrand, D., and Taufer, M., editors, *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*, pages 3–14. ACM.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

[Li, 1988] Li, K. (1988). IVY: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Processing, ICPP '88, The Pennsylvania State University, University Park, PA, USA, August 1988. Volume 2: Software*, pages 94–101. Pennsylvania State University Press.

[Lim et al., 2006] Lim, M. Y., Freeh, V. W., and Lowenthal, D. K. (2006). MPI and communication - adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*, page 107. ACM Press.

[Nelson et al., 2015] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., and Oskin, M. (2015). Latency-tolerant software distributed shared memory. In Lu, S. and Riedel, E., editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 291–305. USENIX Association.

[Oleksiak et al., 2017] Oleksiak, A., Kierzynka, M., Piatek, W., Agosta, G., Barenghi, A., Brandolese, C., Fornaciari, W., Pelosi, G., Cecowski, M., Plestenjak, R., Cinkelj, J., Porrmann, M., Hagemeyer, J., Griessl, R., Lachmair, J., Peykanu, M., Tigges, L., vor dem Berge, M., Christmann, W., Krupop, S., Carbon, A.,

Cudennec, L., Goubier, T., Philippe, J., Rosinger, S., Schlitt, D., Pieper, C., Adeniyi-Jones, C., Setoain, J., Ceva, L., and Janssen, U. (2017). M2DC - modular microserver datacentre with heterogeneous hardware. *Microprocess. Microsystems*, 52:117–130.

[Olofsson et al., 2014] Olofsson, A., Nordström, T., and Zain-ul-Abdin (2014). Kickstarting high-performance energy-efficient manycore architectures with epiphany. In Matthews, M. B., editor, *48th Asilomar Conference on Signals, Systems and Computers, ACSSC 2014, Pacific Grove, CA, USA, November 2-5, 2014*, pages 1719–1726. IEEE.

[Pérache et al., 2008] Pérache, M., Jourdren, H., and Namyst, R. (2008). MPC: A unified parallel runtime for clusters of NUMA machines. In Luque, E., Margalef, T., and Benitez, D., editors, *Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008, Proceedings*, volume 5168 of *Lecture Notes in Computer Science*, pages 78–88. Springer.

[Rodriguez-Gonzalo et al., 2016] Rodriguez-Gonzalo, M., Singh, D. E., Blas, J. G., and Carretero, J. (2016). Improving the energy efficiency of MPI applications by means of malleability. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 627–634. IEEE Computer Society.

[Saravanan et al., 2015] Saravanan, K. P., Carpenter, P. M., and Ramírez, A. (2015). Exploring multiple sleep modes in on/off based energy efficient HPC networks. In *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015*, pages 54–61. IEEE Computer Society.

[Schlitt and Nebel, 2016] Schlitt, D. and Nebel, W. (2016). Data center performance model for evaluating load dependent energy efficiency. In Atlantis, editor, *ICT for Sustainability*. Atlantis Press.

[Thakur and Gropp, 2007] Thakur, R. and Gropp, W. (2007). Test suite for evaluating performance of MPI implementations that support mpi_thread_multiple. In Cappello, F., Hérault, T., and Dongarra, J. J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*, pages 46–55. Springer.

[Tiwari et al., 2012] Tiwari, A., Laurenzano, M., Peraza, J., Carrington, L., and Snavely, A. (2012). Green queue: Customized large-scale clock frequency scaling. In Liu, J., Chen, J., and Xu, G., editors, *2012 Second International Conference on Cloud and Green Computing, CGC 2012, Xiangtan, Hunan, China, November 1-3, 2012*, pages 260–267. IEEE Computer Society.

[Trabelsi et al., 2019] Trabelsi, K., Cudennec, L., and Bennour, R. (2019). Application topology definition and tasks mapping for efficient use of heterogeneous resources. In Schwardmann, U., Boehme, C., Heras, D. B., Cardellini, V., Jeannot, E., Salis, A., Schifanella, C., Manumachu, R. R., Schwamborn, D., Ricci, L., Sangyoon, O., Gruber, T., Antonelli, L., and Scott, S. L., editors, *Euro-Par 2019: Parallel Processing Workshops - Euro-Par 2019 International Workshops, Göttingen, Germany, August 26-30, 2019, Revised Selected Papers*, volume 11997 of *Lecture Notes in Computer Science*, pages 258–269. Springer.

[Venkatesh et al., 2015] Venkatesh, A., Vishnu, A., Hamidouche, K., Tallent, N. R., Panda, D. K., Kerbyson, D. J., and Hoisie, A. (2015). A case for application-oblivious energy-efficient MPI runtime. In Kern, J. and Vetter, J. S., editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 29:1–29:12. ACM.

[Zaourar et al., 2018] Zaourar, L., Aba, M. A., Briand, D., and Philippe, J. (2018). Task management on fully heterogeneous micro-server system: Modeling and resolution strategies. *Concurr. Comput. Pract. Exp.*, 30(23).