Journal of Systems Architecture 56 (2010) 392–406

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Hardware/software support for adaptive work-stealing in on-chip multiprocessor

Quentin Meunier [a], Frédéric Pétrot [a,*], Jean-Louis Roch [b]

[a] TIMA Laboratory, INP Grenoble, 46, avenue Félix Viallet, 38031 Grenoble Cedex, France
[b] LIG, INP Grenoble and INRIA, 51, avenue Jean Kuntzmann, 38330 Montbonnot Saint-Martin, France

ARTICLE INFO

Article history:
Received 1 October 2009
Received in revised form 28 May 2010
Accepted 14 June 2010
Available online 25 June 2010

Keywords:
MPSoC architectures
Concurrent programs
Work-stealing
Hardware/software codesign
Design space exploration

ABSTRACT

During the past few years, embedded digital systems have been requested to provide a huge amount of processing power and functionality. A very likely foreseeable step to pursue this computational and flexibility trend is the generalization of on-chip multiprocessor platforms (MPSoC). In that context, choosing a programming model and providing optimized hardware support to it on these platforms is a challenging task. To deal in a portable way with MPSoCs having a different number of processors running possibly at different frequencies, work-stealing (WS) based parallelization is a current research trend.

The contribution of this paper is to evaluate the impact of some simple MPSoCs' architecture characteristics on the performance of WS in the MPSoC context. The previous evaluations of WS, either theoretical or experimental, were done on fixed multicores architectures. This work extends these studies by exploring the use of WS for the codesign of embedded applications on MPSoC platforms with different hardware capabilities, thanks to cycle-accurate measures.

We firstly study the architectural choices suited to WS algorithms and measure the benefit of these architectural modifications. To assert whether WS is suited to the MPSoC context, we experimentally measure its intrinsic implementation overhead on the most efficient architectural designs. Finally, we validate the performances of the approach on two real applications: a regular multimedia application (temporal noise reduction) and an irregular computation intensive application (frames of the Mandelbrot set).

Our results show that enhancing MPSoC platforms having up to 16 processors with widespread hardware support mechanisms can lead to important performance improvements at acceptable hardware cost for the considered applications.

1. Introduction

Even though in the past years many, if not all, embedded systems were made of one General Purpose Processor (GPP) and one Digital Signal Processor (DSP), there is a trend (not only in academia) to think about having farms of power efficient processors (small GPP or dual issue VLIW) to achieve differentiating compute intensive functions in software [1,2]. Thus, the parallel specifications used for embedded appliances tend to be implemented for a large part as parallel programs.

Several algorithms used in the devices that are MPSoCs based are currently essentially specified as coarse-grain sequential tasks communicating through lossless fifos, *e.g.* boolean/synchronous data-flow representations [3] for which optimal schedules can be derived, or Kahn process networks [4] that have the property of having an output that does not depend on the schedule while being less constrained than the previous formalisms. This trend has been adopted by several players of the consumer electronic industry in order to benefit from the properties of these models [5–7]. Other approaches are more loose, using subsets of well known parallel programming libraries for which the properties of the programming model is less clear: MPI [8], light versions of Corba [9], OpenMP [10,11], or even bare shared memory threads.

In order to provide optimized *ad-hoc* hardware implementations at the price of flexibility, many authors have proposed codesign approaches to support these programming models. Among many initiatives, we can cite [9] that introduces hardware support to Corba-like communication and efficient SMP task management and [12,13] that define specific hardware IP and software APIs to automate the mapping of process networks with different kinds of FIFOs. [14] provides a very broad survey on the general codesign approaches, whereas [15] focuses more on the topic of this paper: providing an optimized hardware/software interface for given programming models.

Indeed, in this work, we look towards another type of parallel programming based on a work-stealing scheduling paradigm [16,17]. We focus on a specialization of work-stealing based algorithms, denoted in the following AWS (adaptive work-stealing)

* Corresponding author. Tel.: +33 4 76 57 48 70; fax: +33 4 76 57 49 81.
E-mail addresses: Quentin.Meunier@imag.fr (Q. Meunier), Frederic.Petrot@imag.fr (F. Pétrot), Jean-Louis.Roch@imag.fr (J.-L. Roch).

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.06.007

[18]. AWS algorithms are based on the principle that each processor executes its own task until it becomes idle, and then steals a fraction of the remaining work on a randomly chosen busy processor.

Work-stealing algorithms have been shown to behave well in practice when the workload cannot be well estimated *a priori* [19–22], which is the case for algorithms like encoding or compression often used in embedded consumer devices. These algorithms also fit well applications running on embedded platforms in which all the cpus do not run at the same frequency, thus creating an unbalanced workload.

Being able to bound the execution time is important for many embedded applications. This can be achieved by an *a priori* knowledge of execution times, but this is hardly feasible on multiprocessor platforms. To deal with this issue, one common strategy is to use iterative improvement algorithms when possible, such as iterative turbodecoding [23] or video decoding [24].

As this approach may fit well with the foreseeable massively multiprocessor integrated architectures that are currently being developed in the industry, we believe that this programming model may be well suited for some typical MPSoC applications, and that it is thus worthwhile to perform a codesign analysis for it.

Our goal in this work is therefore to perform an analysis of the effect of some simple MPSoC architecture characteristics on the performances of algorithms based on the AWS programming paradigm. The type of architectural modifications we target is the use of coherent caches, local memories and DMAs, and distributed locks, for which we also determine how it must be taken into account in the software. We do this by systematic cycle-accurate simulation of different platforms for several program examples. We furthermore measure the overhead of the AWS algorithms compared to the corresponding static parallel execution – called PAR in the remainder of the article – for the same hardware configurations. Based on the pre-partitioning of the input into chunks of equal size, the PAR algorithm represents a lower bound on the execution time in case of applications for which the workload is known beforehand: indeed supplying work-stealing at runtime has a cost, which is to be differentiated from the resulting possible time saved due to the dynamic load balancing.

More precisely, we will:

- Measure the execution time for a synthetic application parallelized with AWS on different architectures in order to evaluate how some usual design choices can impact the performances and to which extent.
- Measure the speedup with either a PAR or an AWS parallelization compared to the sequential execution time for these different architectures.
- Infer from these measures the overhead of an AWS algorithm compared to the corresponding PAR.
- Validate the approach on two computationally intensive applications: one with a uniform workload, and one with a non-uniform workload.

To the best of our knowledge, this work is the first one which exploits the specificity of the AWS behavior to drive the definition of both a hardware architecture and the run-time library which makes an optimized use of it.

The remainder of the paper is organized as follows. Section 2 briefly summarizes the main points of the work-stealing programming paradigm and gives an overview of the works which have evaluated the implementation cost of work-stealing on, often idealized, general purpose parallel machines. Section 3 introduces the initial hardware platform, the basic operating system, and the application template. Section 3.4 details the WS interface suited to data processing, and gives analytical runtimes on a synthetic, thus theoretically analyzable, application that fits our template. Section 5 concurrently explores several hardware architectures and their associated low level software to minimize the communication and synchronization costs. We then analyze the results in Section 6, and give the limitations of the approach before concluding.

2. Background on work-stealing and related work

2.1. Background

Work-stealing is a scheduling paradigm for parallel computations. It is a decentralized thread scheduler [25]: whenever a processor runs out of work, it steals work from a randomly chosen processor.

From work-stealing reference implementation Cilk [20], a three keywords parallel programming extension of C, interest for work-stealing on multicore architectures is growing. In July 2007, Intel launched the open-source Threading Building Blocks (TBB), a set of high-level C++ primitives similar to the STL with thread safe containers and parallel algorithms; available from mid 2008, Cilk++ extends C++ for portable programming of multicores.

Cilk and TBB adopt the oldest-first randomized work-stealing strategy. The implementation is based on double-ended queues (deque). Each processor manages a deque of ready tasks that it uses as a stack for its own tasks (LIFO): it pushes the tasks it creates or unblocks at the bottom of the deque; when it completes a task, it pops a new one from the bottom of its deque if not empty. Otherwise, the processor is idle and becomes a stealer: it sends a steal request to a randomly chosen processor, until finding a *victim* processor with a non-empty deque; then it picks a task from the top of the deque of the victim, the oldest one. Oldest-first work-stealing achieves provable performances. Arora et al. [16] showed that for any parallel program, the time T_p on p identical processors verifies with high probability (w.h.p.): $T_p \leqslant \mathcal{O}\left(\frac{T_{seq}}{p} + T_\infty\right)$ where T_{seq} is the sequential execution time (that corresponds to the computational work) and T_∞ is the maximum depth (execution time on an unbounded number of processors). With slight variants, similar bounds are achieved when the number of processors allocated for the computation varies during execution [16] and for processors with variable speeds [17]. In particular, the number of steal requests is $\mathcal{O}(pT_\infty)$ w.h.p.; therefore it is small in the case where $T_\infty \ll T_{seq}$.

This paper restricts to the case $T_\infty \ll T_{seq}$ which matches many important embedded streaming applications. Since the number $O(p \cdot T_\infty)$ of steal requests is small with respect to the total work, the *Work First Principle* consists of putting most of the overhead of the scheduling at steal request operations and optimize the sequential execution of the parallel algorithm. In [22], the *deque-free* work-stealing implementation consists of decreasing the overhead for the management of the deque by delaying tasks creation only after a steal request occurs on a victim processor. In this case, the operation named "parallelism extraction" [22] creates a new task which is assigned to the theft processor. Similarly, in the *Tascell* framework [26], a worker creates a real task only when requested by another idle worker. This strategy is named *bactracking-based load balancing*; the worker performs parallel extraction by temporarily "backtracking" and restoring its oldest task-spawnable state [26].

2.2. Related works

Several works analyze the performance of work-stealing on SMPs machines, either discrete or integrated (multicores), especially in the context of Cilk [20]. Considering CMPs [27], focuses

on the number of cache misses by comparing the performance of two different implementations of work-stealing, *i.e.* traditional (WS) and Depth first (PDF). In the case of our application template presented in Section 3.4, both strategies are equivalent (as exhibited by the theoretical analysis of the cache misses).

Experiments of the use of the Capsule environment, that proposes a run-time support for recursive splitting of work are presented in [28]. These results are obtained on a predefined four cores platform and do not explore alternatives implementations of data placement and synchronization.

In [18] is presented a processor-oblivious algorithm which is the base of the AWS algorithms used here. The algorithm is proved to be asymptotically optimal and it is shown that the algorithm has a good experimental behavior. This work however does not encompass the embedded field nor architecture properties and remains mainly theoretical.

Ref. [29] studies the problem of the memory hierarchy in MPSoC systems, and proposes a methodology to compare different hierarchies, for a large range of applications. Different criterion are used to evaluate the results (time, energy, latency and bandwidth issues, etc.), but none of the benchmarked algorithms is implemented using a work-stealing technique. Additionally, the aim is not to compare the cost of different parallelism techniques for the same program.

There is much work dealing with data-partitioning [30] and architecture properties for parallel applications, but to the best of our knowledge, the architecture/software support study for the adaptive work-stealing that we conduct in this research is the first of its kind.

3. The MPSoC architecture and applications templates

3.1. Hardware

Since the MPSoC design space is huge, we define a template architecture for which we set the parameters that we estimate being either non-relevant to or not interacting significantly with our performance evaluation study. As it can be seen on Fig. 1, the platform is an interconnection of CPU sub-systems. All CPU subsystems share a common address space, and can access local or shared memory modules, a timer and a lock engine that allows to take a lock by simply reading from it, *i.e.* a hardware IP which implements a *test-and-set* for a range of addresses [31]. The chosen processor is a simple 4 stage pipeline, 4 windows, Sparc V8. The processor accesses separated direct mapped data and instruction caches. This may be a bit less efficient than a 2-way set associative cache, but simplifies the hardware implementation and ensures that instructions cannot trash data and *vice-versa*. There is no automatic data prefetching (excepted for filling a cache line), as in many integrated solutions, speculatively loading instructions or data is not considered energy efficient. The architecture configuration is presented Table 1, and all the architectures used in this study are variations of this one.

In order to avoid high contention on a unique shared memory location or high latencies as it would occur on a *dance-hall* architecture, the platform on which we based our study has two levels of hierarchy. Each processor is connected on a local interconnect to its peripherals and a local storage through a crossbar. These local interconnects are connected via a bridge on a global interconnect on which are also connected the shared memories.

The interconnect used is a Network-on-Chip based on the work of [32], as the scalability of buses is very limited, and the complexity of crossbars becomes too important for the number of processors targeted. The topology used is a 2D-mesh, since it has a good crossing-time versus complexity ratio, and has good layout properties for silicon implementation.

Table 1
Simulated platforms characteristics.

Number of processors	$p = \{1, \ldots, 16\}$
Number of memory banks	$p + 3$
Processor model	SPARC-V8 with FPU
Data cache size	16 Kb
Data block size	8 words (32 bytes)
Instruction cache size	16 Kb
Instruction block size	8 Words
Cache associativity	Direct-mapped
Write-buffer size	8 Words
DMA Controller	2 Initiator interfaces to issue 1 read and 1 write per cycle at full speed
NoC topology	2D Mesh
Global NoC latency	$\sqrt{2n}$ Cycles for n interfaces
Local NoC latency	1 Cycle

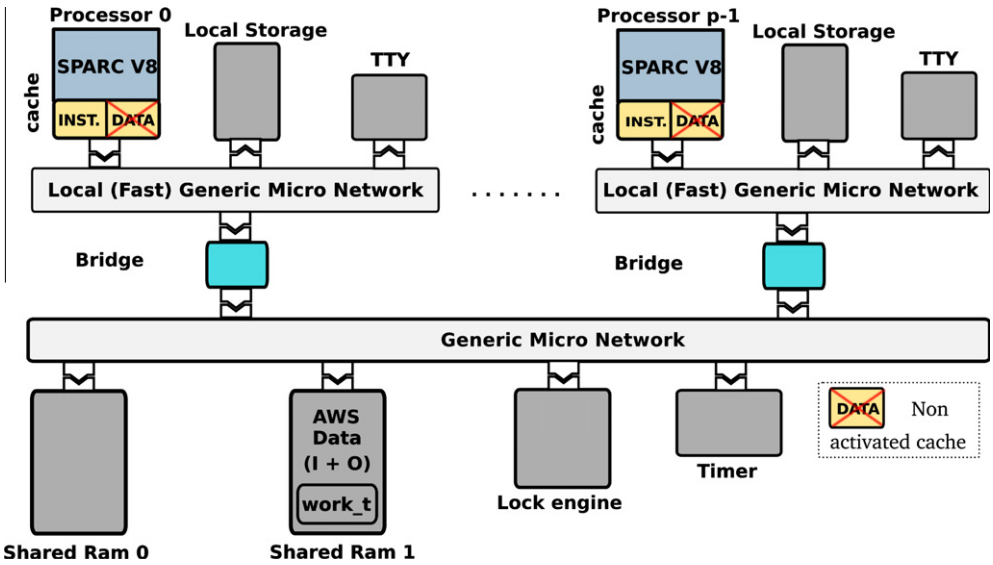


Fig. 1. Basic architecture schematic view.

It would have been interesting to use a scratch pad instead of a memory plugged on the local interconnect (i.e., a memory directly connected to the processor through a dedicated interface [33]), but this was not explored as being a limitation of processor models available in the simulation environment. However, thanks to the local crossbar latency, the timing behavior of the two solutions would be very similar.

The address space seen by the processors is partitioned in a set of segments. One or more segments can be mapped on a peripheral or a memory, respecting the following constraints: all segments mapped on a peripheral cannot be cached, and all the segments relative to the same memory component must have the same cache attribute (i.e. cached or not).

To summarize, the hardware design space that will be explored in our study will mainly consists of evaluating how DMAs and/or caches can improve data locality and how the physical placement of locks can speedup synchronization.

3.2. Operating system and task assignment

We use the Decentralized Scheduling (DS) configuration of a lightweight kernel called Mutek [34], which provides an implementation of the POSIX pthreads for shared memory multiprocessor machines. As opposed to its SMP configuration in which all processors share a single scheduler structure to perform task selection, the DS configuration greatly limits contention as each processor has its own scheduler. Tasks can be pinned on a desired processor in order to avoid migration. In that case, each thread is assigned to a processor at creation time. The thread's stack and local data are stored in the local storage of the processor. All experimentations are done using this identical OS configuration.

3.3. Selected applications template

Our choice of a template for the applications has been motivated by three points. Firstly, we must be able to accurately evaluate the overhead of work-stealing as compared to static optimal parallelization when it is known. Secondly, multimedia applications are compute and communication intensive: the application has to be fine-grained and representative of a class of multimedia processing such as digital filters (temporal noise reduction, deblocking) or transforms (DCT). Lastly, it should enable a theoretical analysis of the implementation of the work-stealing in order to have feedback on the experimentations.

We selected a synthetic template application fitting into these constraints, which consists of having its input and output data stored in a buffer array, the operations on the different elements of this array being independent. This array is located in shared memory.

By considering an empty processing operation on each element, this synthetic application has a very high communication *vs.* computation ratio, which enables an analysis of the work-stealing overhead in number of cycles. Furthermore, considering a fixed number of identical processors and a constant time parameter of the processing of each element, this overhead can be compared to the number of cycles of the standard static parallelization, denoted PAR: the input data of size n is equally shared between all the p processors, each processor being in charge of a contiguous block of size $\frac{n}{p}$.

3.4. Codesign based on work-stealing: AWS

Codesign is usually considered as the process of producing the hardware and software fitting the performances and cost constraints of high-level specifications. Among the codesign topics, the design of hardware/software interfaces is very important and our goal in this paper is to define a proper hardware/software interface for applications which can be implemented using work-stealing.

In our codesign approach, presented in Fig. 2, the work-stealing implementation is the adaptation layer between the hardware platform and the application. Each processor acts as a work-stealer: when idle, it sends a steal request to another processor. The work-stealing implements the management of steal requests, based on specific hardware support provided by the Hardware Steal Engine, and the creation of tasks at the application level.

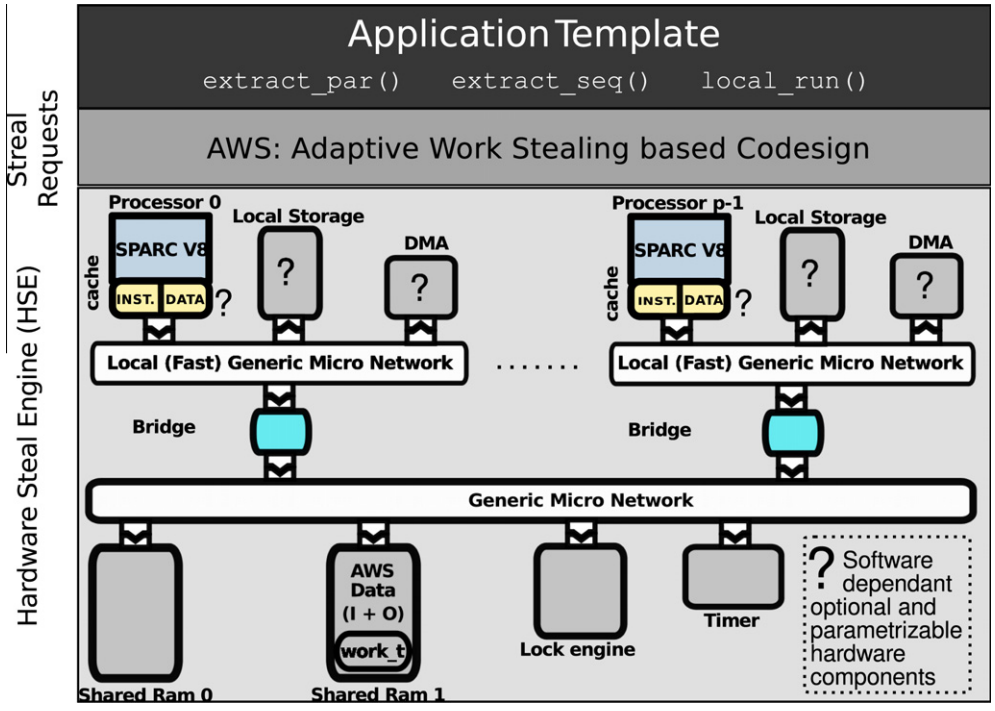


Fig. 2. AWS: adaptive work-stealing based codesign.

When the victim is in a stealable state, it creates a task corresponding to its oldest spawnable task. The effective description of the new created task is provided by the application. The next section details the tunable implementation of the AWS work-stealing layer and the application interface.

4. AWS interface for data processing

4.1. Rationale of AWS

The implementation of the work-stealing is the bridging interface between the hardware and the application. A major constraint in embedded systems and MPSoCs is that the memory space has to be statically bounded. As a consequence, our specification of the work-stealing does not allocate extra memory at runtime, nor requires concurrency since at any time only one serial computation is in progress on each non-idle processing unit.

Indeed, instead of managing on each unit a collection of ready tasks, AWS follows the deque-free work-stealing algorithm proposed in [22] which is based on lazy task creation. A non-idle processor j performs the computation of its assigned task; when a processor i tries to steal work from j, j can continue its local execution while i creates a new task corresponding to a ready part of the remaining work on j; then this task is assigned to processor i which can resume its processing of data.

In the following, the operation which constructs the description of a public task is named `extract_par()`, and the one which constructs a private task is called `extract_seq()`. These operations are implemented by the programmer at the application level. AWS manages on its side the steal requests on idle processors and the local execution of local work on non-idle ones.

When a processor becomes idle, AWS calls a function named `steal` which in turn calls `extract_par()` iteratively on the other processors, until finding some work to steal. On a successful steal operation, the victim's remaining work is split into two parts: one is stolen, and the other one is left to the victim. Thus, this synchronization between the steal requests and the local work is managed at the work-stealing level. The local work on a non-idle processor is specified at the application level, via a function called `local_run()`. AWS serially iterates through the local work until its completion, calling alternatively `extract_seq()` and then `local_run()` on the data extracted.

4.2. AWS API and guarantees

AWS comes in two variants: a high-level API containing a set of standard algorithms for arrays, and a low level interface. The high-level API is restrictive in several ways and does not encompass our framework; for this reason, we will only consider the lower level interface.

In this low level API, the user has to define the following elements:

- the `work_t` structure
- the `extract_par()` function, defining how the data is extracted on a steal operation
- the `extract_seq()` function, defining how the data is extracted on a local extraction – to fit the theoretical model, the amount of data extracted should be $\alpha \log(\text{remaining data})$
- the `local_run()` function, defining how to process a piece of data
- the ratio α of local extraction, if used in the `extract_seq()` function
- the threshold under which a steal operation fails

Basically, the programmer has to specify the work definition and the operations related to this work. The ratio and treshold parameters allow the calibration to a specific program.

Once these elements are defined, AWS can create the tasks and manage their synchronization, as described in the previous section.

The guarantees provided by the system are that a processor is either active or performing a steal, or that no more work can be stolen, meaning that each processor is completing locally the last part of its work. Also, the system is guaranteed to stop after all the work has been computed.

To bound the scheduling overhead, the `extract_seq()` function must extract $\alpha \times \log$ of the remaining amount of work and `extract_par()` must extract a fraction, generally half, of the remaining work. Following these constraints, the implementation guarantees an asymptotic time optimality of stream computations via a theoretical analysis on the size of the chunks for `extract_par()` and `extract_seq()`, while being processor-oblivious (i.e. it does not depend on the number of processors [18]).

More complex parallelisms, like recursive ones, are not supported by the system. However, the programmer is free to implement it at the application level, being then in charge of the induced synchronizations, and the memory constraints related to it.

4.3. Adaptive work-stealing implementation

Based on previous design choices, this paragraph presents AWS implementation on embedded systems, taking into account their constraints.

The overall behavior is as follows. At start time, each processor is in busy state and starts a computation, usually determined using the processor identifier. When a processor goes into idle, it becomes a stealer. It cyclically selects a victim until it finds some work to steal.

From an implementation point of view, each processor manages two `work_t` structures which represent a part of the total work. At a given time, a piece of data is contained in at most one `work_t` structure in the system. The first `work_t` structure is a public structure which is visible to all the other processors. It is initialized with an amount of work, generally the same amount for all the processors. The second `work_t` structure is a private structure which is only visible for the processor and which contains the amount of work to compute locally.

The adaptive work-stealing algorithm is presented in Fig. 3.

After a successful steal request, a processor gets a work w and executes two nested loops: the steal-loop and the local-loop. In the local-loop, the processor executes the function `extract_seq()` which extracts a small amount of work from the public `work_t` w to the private `work_t` l. Then the processor computes this work l with the function `local_run()`: this computation is done sequentially without any preemption.

When the public `work_t` structure w is empty, the processor becomes a stealer and exits the local-loop. It calls the `steal` function, which scans all the public `work_t` structures via the `extract_par()` function, until it finds a non-empty one. In case of successful steal, this function extracts some work from the victim work v in its own public `work_t` structure w. The processor then enters the local-loop again. When all the public `work_t` structures are empty, the whole work has been computed.

4.4. Nature of `work_t` structures

The `work_t` structures generally do not contain the data itself, but instead a description of the work, such as an index on the beginning of the work and the size of the work. Indeed, in most dis-

```
1  /* node_mutex: lock protecting the shared (stealable)
2     work of the current node */
3  aws_lock(node_mutex);
4  has_local_work = TRUE;
5  has_global_work = TRUE;
6  while (has_global_work) {      /* steal-loop */
7    while (has_local_work) {     /* local-loop */
8      status = extract_seq(); /* extract local work l from w */
9      if (status == STATUS_OK) {
10       aws_unlock(node_mutex);
11       local_run();            /* work locally on l */
12       aws_lock(node_mutex);
13     } else
14       has_local_work = FALSE;
15   }                            /* end of local-loop */
16   /* try steal */
17   aws_unlock(node_mutex);
18   status = steal();           /* fetch shared work to do : w */
19   aws_lock(node_mutex);
20
21   if (status == STATUS_OK)
22     has_local_work = TRUE;
23   else
24     has_global_work = FALSE;
25 }                              /* end of steal-loop */
26 aws_unlock(node_mutex);
```

Fig. 3. Core of the AWS algorithm.

tributed algorithms, the data must be differentiated from the description of a subpart of it, defining conceptually which accesses can be performed. Programs written using AWS should follow this principle, as bigger `work_t` structures will result in slower steals.

For the case in which the input data is an array, Fig. 4 illustrates the node structure which contains two `work_t` structures pointing respectively towards the first and the last element of the work currently used by the local node, and the first and the last element of the remaining amount of work which can be stolen.

4.5. Theoretical analysis for PAR and AWS

Initially, the input data is pre-allocated between the processors. We assume in this section that in a successful steal operation, the `extract_par()` extracts a half of the remaining interval of the victim, and that the `extract_seq()` extracts the first $\alpha \log$ elements of its remaining interval.

In the restricted case of a very regular application, the static parallelization (PAR) in chunks of equal size will provide optimal performances. This matches some digital signal or image processing codes which are static control programs [35]. Moreover, we will first consider our fine-grained template application: this enables to quantify the overhead brought by the adaptive work-stealing paradigm.

The simplicity of the template and its implementation facilitates a theoretical analysis. The following notations are used: as defined in Section 2, T_{seq}, T_p and T_∞ denote respectively the sequential execution time, the parallel execution time on p processors, and the parallel execution time on an unbounded number of processors. We assume that the processing time τ of a single element verifies $\tau_{min} \leqslant \tau \leqslant \tau_{max}$. In the following section, we consider that the instruction caches can hold the whole application and we restrict the analysis to the data caches.

4.5.1. Theoretical analysis for PAR

Let us first consider the number of data cache misses. Let M_{seq} be the number of cache misses of the reference sequential execution which corresponds to the linear traversal of the array. In the PAR execution, each processor executes the sequential algorithm on its own chunk. Thus the number of cache misses M_p^{PAR} per

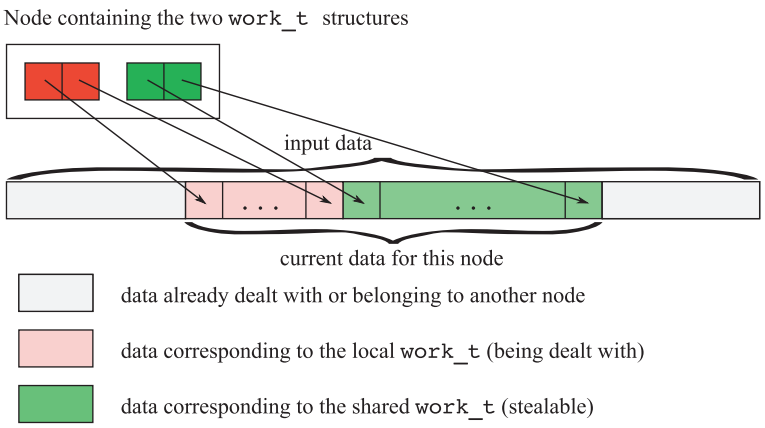


Fig. 4. Data contained in a node.

processor verifies $M_{seq} \leqslant p \times M_p^{PAR} \leqslant M_{seq} + p$. Since $p \ll M_{seq}$, the overhead induced by parallel cache misses is negligible.

Then, the execution time T_p^{PAR} is equal to the execution time of the chunk that maximizes the computational work. Assuming a constant time operation, we have

$$T_p^{PAR} \simeq \frac{T_{seq}}{p}. \quad (1)$$

In the general case in which the computation time of an element may vary, we just have

$$\frac{T_{seq}}{p} \leqslant T_p^{PAR} \leqslant \frac{\tau_{max}}{\tau_{min}} \frac{T_{seq}}{p}. \quad (2)$$

4.5.2. Theoretical analysis for AWS

For AWS, we denote τ_{steal} a bound on the time of a steal (either successful or unsuccessful) operation on a given processor. The overhead of AWS is related to the total number $\#S$ of steal operations which is proportional to T_∞.

Due to the initialization and the extraction of half of the work at steal and local extraction of log, we have that $T_\infty = \mathcal{O}\left(\log \frac{T_{seq}}{p}\right)$. Moreover, due to the cyclic search of a victim processor, the total number of steal operations is $\#S = \mathcal{O}(p \times T_\infty)$. w.h.p. and in the worst-case

$$\#S = \mathcal{O}(p^2 \times T_\infty). \quad (3)$$

Similarly to PAR, the number M_p^{AWS} of caches misses per processor for the traversal of the array is bounded in the worst-case by: the number M_{seq} of cache misses induced by the sequential execution, plus at most two additional ones after each successful steal operation: one on the stealing processor to load a new subarray, and one on the victim to update its local work. Thus we have: $M_{seq} \leqslant p \times M_p^{AWS} \leqslant M_{seq} + 2\#S$. Note that in the case of the application template, the processor which performs a steal operation is considered as idle, and therefore has no useful data in its cache. This is why we can safely ignore the caches misses before the successful steal. Finally, the expected execution time is:

$$T_p^{AWS} = \frac{T_{seq}}{p} + \mathcal{O}(p \times T_\infty). \quad (4)$$

5. Hardware/software codesign for AWS implementation

Beyond theoretical analysis, the effective performances for both PAR and AWS are heavily related to the hardware configuration. At a fine grain, the application template is communication intensive, so the use of caches and DMAs has a direct impact.

A basic way to improve performance is to overlap computations and communications. In the case of our study, it will take the form of using the local storages to reduce memory access latencies. This can be done by using a DMA to copy data in the local storage of a processor while the latter is doing computations. The use of caches will also be studied, along with the use of both caches and DMAs.

Besides, in AWS, additional synchronizations occur due to `extract_par()` and steal operations: tuning their implementation may be critical. For instance, accessing a lock at each `extract_seq()` operation will be inefficient at a fine grain. Since most accesses are local, distributing the locks and the structures on the local interconnects may allow to reduce access latencies.

5.1. Using DMAs

In order to explore the use of DMAs, the basic architecture is modified by adding a DMA unit on each local interconnect (Fig. 5). This way, the input data can be accessed in the local storage instead of the shared memory. Memory allocation in the local storages is made possible thanks to a specific system call.

The first issue to deal with when using DMAs is synchronization, i.e. we want to be sure that the data accessed locally is the expected one (in other words, that the copy is finished). This can be done either by polling or by interruption. Polling appears more attractive as it allows to start the processing of data before the end of the copy. Our implementation of polling uses a DMA register which contains the number of transferred items and uses this value to compute the next index until which the processing can be performed. This cost of these requests is thus negligible as the speed of the copy is faster than the speed of processing elements.

The second question is to decide when to perform the copy from the shared memory to the local storage. Several possibilities were envisaged: at the beginning of the `local_run()` function, in the `extract_seq()` function, or in the `extract_par()` function. Additionally, for the first two cases, the copy can refer either to the current data set (on demand memory access), or to the next data set (read-ahead) to be processed.

Copying a part of the stolen data in the `extract_par()` function allows to start the computation right when the `extract_seq()` function is later called. However, this strategy prevents communication and computation from being overlapped. The alternative strategy is to program the DMA at the beginning of

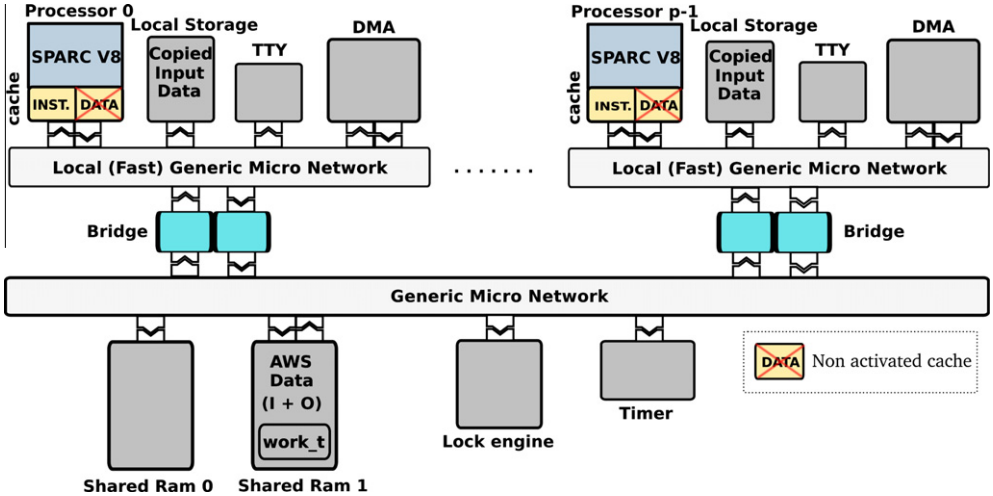


Fig. 5. Architecture with DMAs.

either the `local_run()` or the `extract_seq()` functions, with similar results. However, only the first solution is applicable on both PAR and AWS. Therefore, all experimentations presented in the following program the DMA in the beginning of the `local_run()` function.

About the read-ahead question, copying the next set of data (i.e. the one corresponding to the data processed in the next call to `extract_seq()`) requires to be able to distinguish the first and last calls to `extract_seq()` after an `extract_par()`. Since copying the current set of data only adds a few reads of the status register, we decided to limit to on demand memory access.

5.2. Using caches

For our synthetic application, caches may seem useless since we access each piece of data only once. But actually, using caches allows to prefetch a data line. In order to maintain cache coherency among all the caches, we used the implementation of a write-through directory-based hardware mechanism (Fig. 6) detailed in [36].

5.3. Using caches and DMAs

We also investigated the joint usage of both caches and DMAs, as they operate on different parts of the transfer: the DMA copies data locally and a cache prefetches it (Fig. 7). The cached memory segment is the one corresponding to the local storage. Since only one processor accesses a given local storage, it does not need to be cached in a coherent way. To optimize the accesses and prevent false sharing, the base address of the local table, and the size of the locally accessed data are systematically aligned on the cache block size. This can be ensured at allocation time using a memory allocator which guaranties alignment (`posix_memalign`).

5.4. Distributing locks and work structures

Another way to reduce access latencies is to distribute the locks on each node instead of accessing them through the global interconnect. Similarly, the `work_t` structures can be stored locally and not in shared memory.

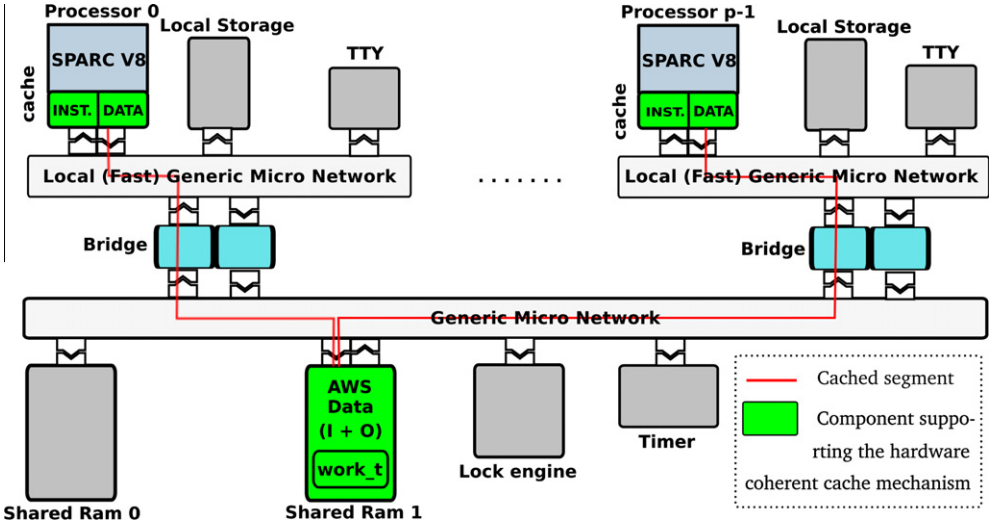


Fig. 6. Architecture with coherent caches.

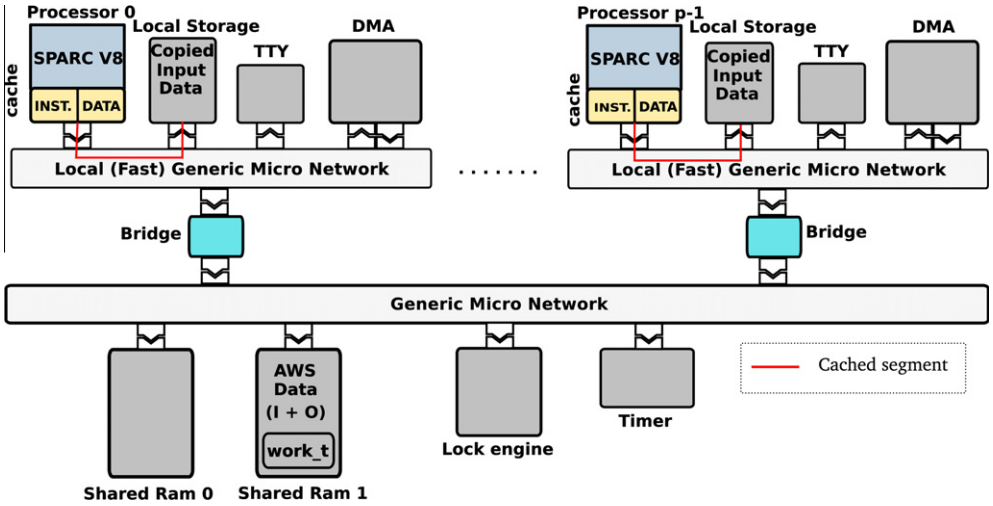


Fig. 7. Architecture with caches and DMAs.

This requires that each processor gets an access to the local interconnects of the other processors. As a consequence, this increases the access time to the lock in case of a steal. However, steals are proven to remain rare (Eq. (3) [17]). Therefore, following the Work First Principle, the majority of accesses to locks or `work_t` structures are local: it namely happens when a node extracts work to process it sequentially.

In order to limit the number of joint hardware/software configurations to compare, we decided to implement and run the following two configurations:

- the basic architecture, without neither data cache coherence nor specific DMA support (see Fig. 1 with shared values uncached), in order to provide reference performance measures,
- the architecture with coherent caches.

6. Results and analysis

The simulations were done using the Cycle-Accurate Bit-Accurate (CABA) SystemC models of the SOCLib [37] library.

We ran the experiments for two data configurations: the first one consists of arrays with a total of 100,000 elements, while the second one of arrays with a total of only 10,000 elements. As explained in section 4.3, the number of elements extracted via the `extract_seq()` function must be $O(\log(m))$, where m is the number of remaining elements.

6.1. Comparing the execution times on different architectures

The Fig. 8a and b shows the normalized execution times (w.r.t. the times on the basic architecture) for the PAR and AWS algorithms, with arrays of 100,000 elements.

The first noticeable information is that the simulation times obtained between AWS and PAR scale similarly for the three configurations, but are a bit less regular with AWS compared to PAR. This can be explained by the traffic due to the final synchronization in AWS. In fact, depending on the order of all the requests sent in this phase ($\mathcal{O}(p^2)$), some threads can commute and go into idle, which introduces overhead.

Going more into details for each architecture, we notice that coherent caches and the architecture with DMAs and caches perform the best and do equally well (speedup from 4 to 1.7), whereas the architectures with DMAs alone does not show a significant time-saving compared to the basic architecture. However, for a high number of processors, improvements are visible for the PAR for this architecture, especially in comparison with the two fastest configurations. This can be explained by the fact that for a high number of processors, the latency on the shared interconnect becomes high enough so that copying data in the local storage is worth.

We can also see that the time saved is decreasing as the number of processors is increasing for the two fastest configurations (coherent caches, DMA and caches). Indeed, these configurations optimize the data accesses, inducing faster computation. However, this does not help improving the parallelism cost nor the synchronization overheads, which represent a higher percentage of the total execution time with many processors.

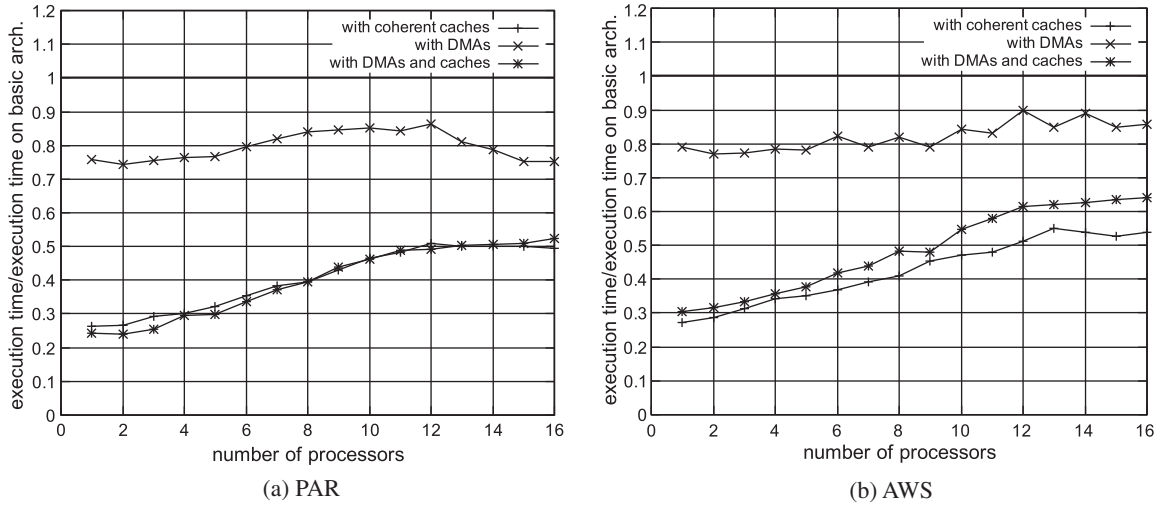
The Fig. 9a and b shows the execution times with arrays of 10,000 elements. Compared to the previous results, the non-regularity of AWS execution times is amplified by the small input data size. The management of parallelism in AWS is too expensive for this input data size. This exhibits a limitation of work stealing which is that it cannot provide good performances for too small inputs, even with optimized hardware support.

6.2. Comparing sequential and parallel execution times on a given architecture

Fig. 10 shows the normalized execution times for each architecture and for large input data size.

If all curves globally have the same shape, we can still notice that for both PAR and AWS, the relative time saved thanks to the parallelism is bigger for slower architectures (basic and then DMA): the maximal speedup obtained is almost seven for the slowest architecture and only four for the faster ones. Indeed, as the total execution time is longer, the parallelization overhead, remaining constant, is proportionally smaller w.r.t the total execution time. The same happens for AWS which performs from 5% to 35% worse than PAR depending on the configuration, because of the parallelism overhead. But an important point to notice is that the configuration with DMAs and caches is doing significantly worse than the configuration with coherent caches, especially for AWS: the overhead of AWS compared to PAR is around 20% for the coherent caches, while it is around 35% for DMAs and caches, suggesting that using coherent caches is the best solution to our problem. Finally, these graphs show that for this input data size, the optimal number of processors is around 6.

The graphics showing the same normalized execution times with the small input data size are given Fig. 11.



(a) PAR (b) AWS

Fig. 8. Execution times on the different architectures for 1–16 processors, normalized w.r.t. the times on the basic architecture, with 100k elements.

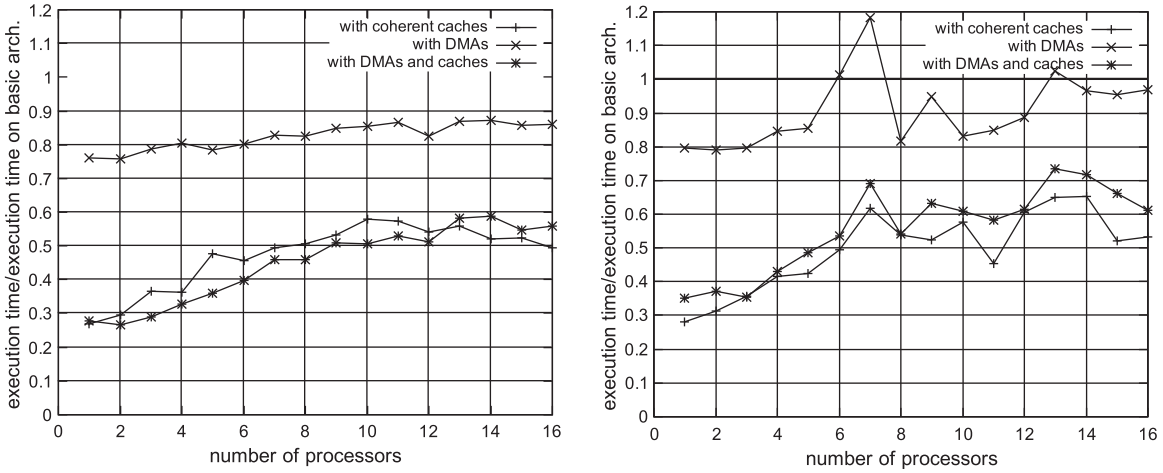


Fig. 9. Normalized execution times on the different architectures for 1–16 processors with 10k elements.

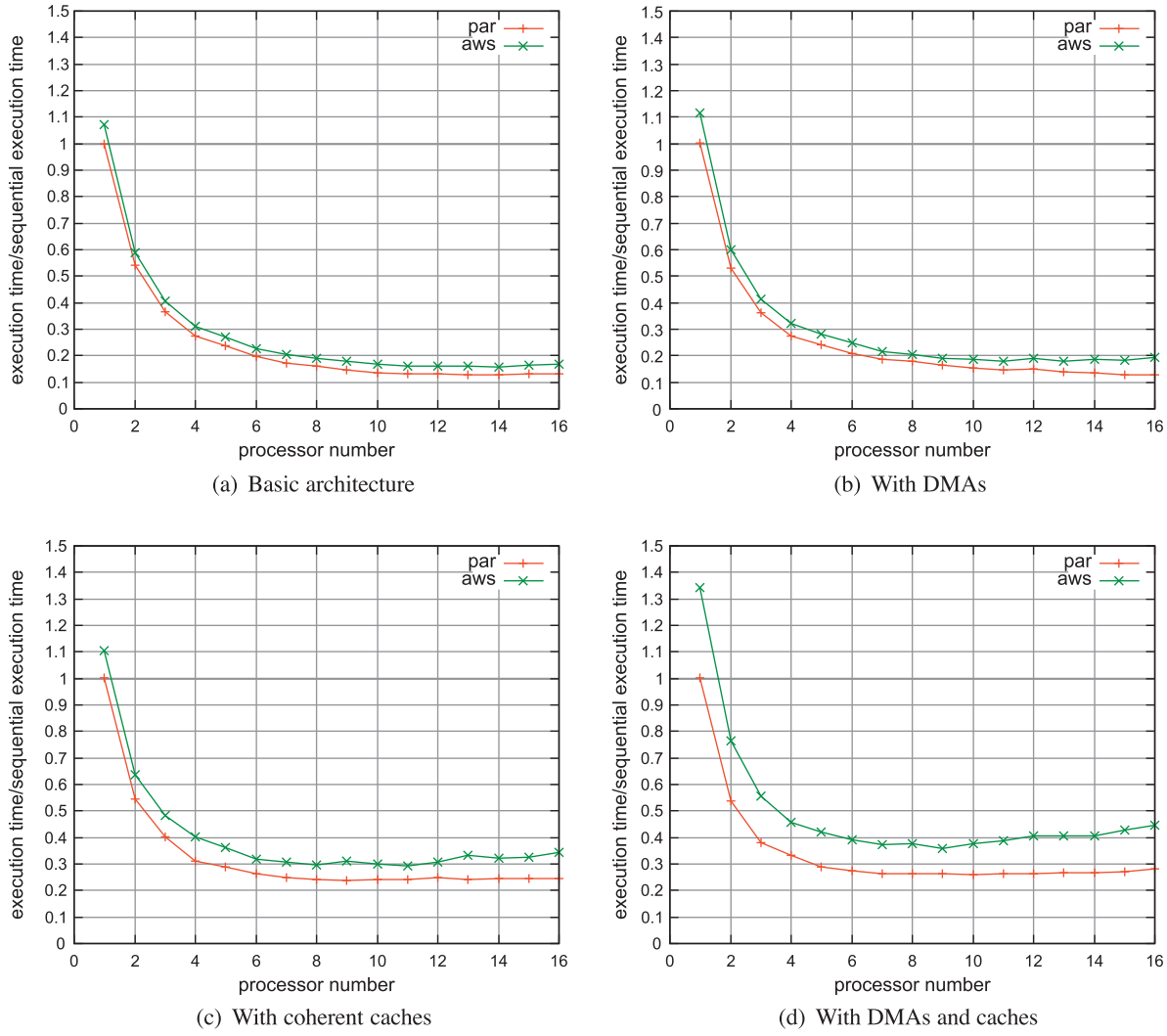


Fig. 10. Normalized execution times on the different architectures for 1–16 processors with 100k elements.

Once again, these graphics reveal that AWS overhead is prohibitive when the data size is too small. The optimal number of processors for this input size seems to be around 4, while AWS execution times on the fastest architectures (DMAs + caches and coherent caches) quickly go beyond the sequential execution time.

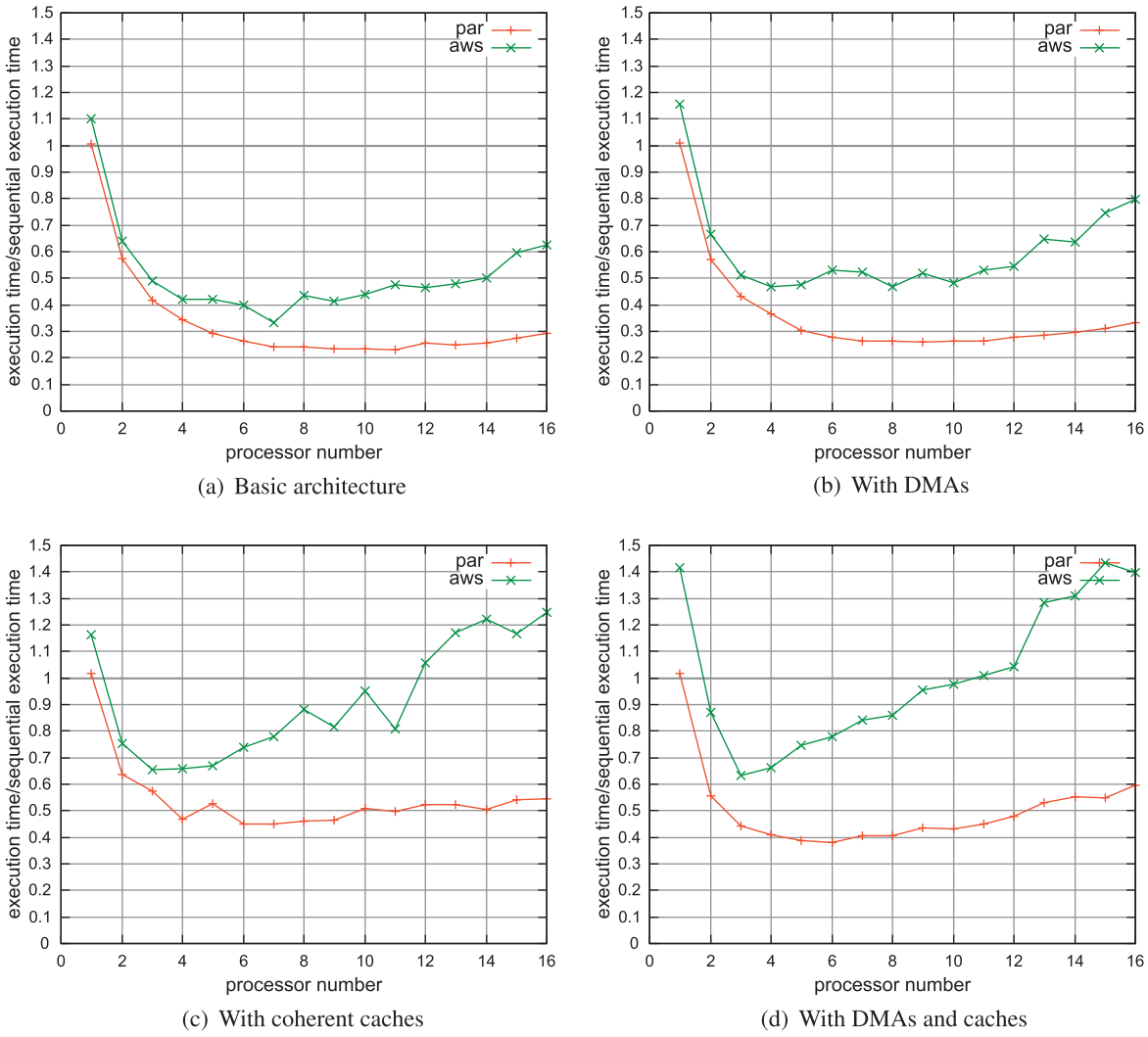


Fig. 11. Normalized execution times for different architectures for 1–16 processors with 10k elements.

Finally, the last trend to notice is that the better the performance for an architecture, the worst the speedup for a given number of processors, for both PAR and AWS. As previously, this is explainable by Amdahl's law, because a part of the application cannot be parallelized.

Though we expected AWS to be slower than PAR, we thought that the relative overhead would be smaller with a small input data size. This shows that improving performances on local runs will be limited at a certain point by the parallelism creation and by synchronizations. This fact motivated the next experiment.

6.3. Distributed locks and work structures

On Fig. 12 are presented the relative time-savings of configurations with local locks and distributed `work_t` structures, for the basic architecture and the architecture with coherent caches. The PAR times were obviously almost identical, since there are no accesses to shared `work_t` structures, nor to the corresponding locks.

First, the time saved is not negligible, even though this must be considered as a maximum time-savings since we are in a case in which there are almost no steals (max. 8 successful steals and less than 300 attempts for 16 processors). Secondly, the time saved on an architecture is increasing as the global execution time is decreasing. This is not obvious since if for a constant time-savings, the decrease of the global time makes it greater, the parallelization involves here fewer accesses to the locks and work structures, and therefore a smaller time-savings.

Besides, the time saved is relatively bigger for the basic architecture. This can be explained by the fact that in the presence of caches, the work structures will be cached as well if they are located in shared memory; therefore, moving these structures in the local storages does not save any time.

6.4. Conclusion

The two approaches presented to improve performance using DMAs or caches follow somehow the two basic memory models existing for contemporary MPSoCs: *hardware-managed, implicitly-addressed, coherent caches* and *software-managed, explicitly-addressed local memories* [29].

Our results tend to show that for AWS algorithms, the two architectures involving caches work better than using an explicit copy alone, and that the solution with hardware-managed coherent caches scales better with small data than the solution with both DMAs and caches. However, our model could be accused here, since the local storages cannot be accessed with a 1-cycle latency by the processor.

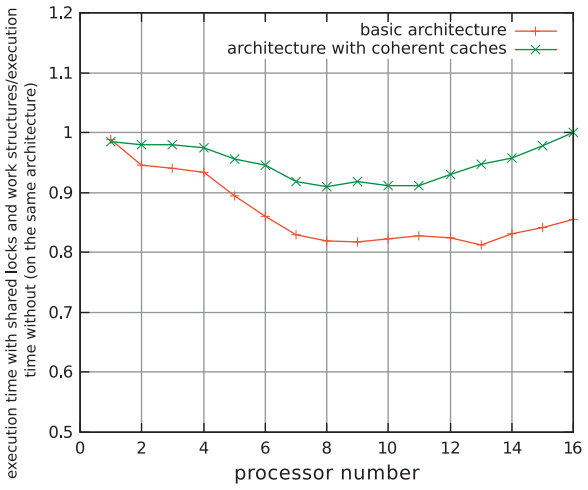


Fig. 12. Normalized execution times with distributed locks and work structures, on two architectures.

We notice that the AWS implementation of the algorithms are very sensitive to small synchronization changes. After investigation, we found that it is due to the fact that when a thread goes into idle (because the mutex required is already taken), it actually consumes a lot of time due to the double context-switch required to yield the processor and gain it again. Using spin locks instead of mutexes could lower this sensitivity. The results also show that AWS is more sensitive to data size variations than a static parallelization, therefore recommending to use AWS when the input data size is large enough.

7. Performances on two real applications

We used two applications for our study: a Luma and Chroma Temporal Noise Reduction (TNR) application, and the computation and drawing of pictures from the Mandelbrot set.

We chose these applications because we want to cover both ends of the spectrum regarding application workload regularity.

Also, these applications have the particularity to contain no dependencies between tasks, and thus enable maximum parallelism to redistribute work. This is typical of streaming applications found in multimedia or radio devices.

7.1. TNR overview

The TNR application is an image filtering program. It contains several successive computations on a frame: temporal noise reduction, spatial noise reduction, motion detection and fading. Each computation is an iteration on all the pixels of the image via a double for loop, which represents a high-level of parallelism. The frame size in the sequence used is 714×244.

The application is *a priori* very regular since the data is split into small blocks and the number of computations performed almost does not differ from one block to another.

On a successful steal, the data movement corresponds to the one of a `work_t` structure, which contains the beginning and size of the block to process.

We ran the experiments on a varying number of processors (between 1 and 16) for the decoding of 4 or 6 frames. Since the computations on this application were time-consuming, we chose the following configurations:

- the basic architecture for PAR (basic PAR)
- the basic architecture for AWS (basic AWS)
- the architecture with caches and distributed locks and work structures for AWS (enhanced AWS)

7.2. TNR results

The detailed results for the decoding of six frames on four processors are presented in Table 2. Comparing the columns for configurations Basic PAR and Basic AWS shows that the execution times are very close between PAR and AWS, and that none of them is performing significantly better than the other.

The performance improvement due to caches and distributed locks is approximately 15% per frame, with a data cache hit ratio greater than 99%. In fact, the time saved is lower than with our test-case application since here there are many more computations for one transaction or lock access.

The broad results for 1, 2, 4, 8 and 16 processors are presented Fig. 13. This graph shows the average decoding time for four frames. As with four processors, AWS and PAR perform always similarly, but this graph allows to put in evidence that the Enhanced AWS configuration is more effective when the number of processors is high. This can be explained by the fact that as the number of processors increases, the global latency increases too, so exploiting locality with caches and local locks gives better results. Furthermore, the number of locks accesses grows in $O(p^2)$, so distributing the locks when the number of processors is high reduces contention.

This shows that for real fine-grained regular applications, which are the worst-case for AWS algorithms, the overhead of AWS is very limited and that AWS still gives very acceptable results.

7.3. Mandelbrot overview

The Mandelbrot application consists of creating a sequence of pictures representing a zoom on some point located on the border

Table 2
TNR execution times (in KCycles).

Frame	Processing time on		
	Basic PAR	Basic AWS	Enhanced AWS
1	6512	6514	5559
2	6527	6518	5571
3	6529	6531	5576
4	6532	6531	5578
5	6529	6527	5571
6	6525	6523	5567

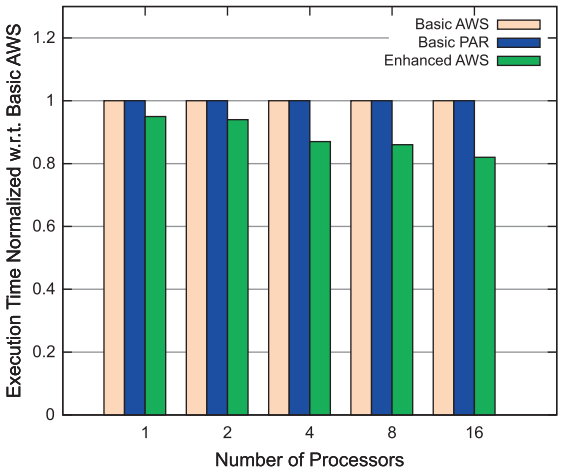


Fig. 13. TNR Execution Time for the decoding of four frames, normalized w.r.t. basic AWS.

of the Mandelbrot set, in order to create an Motion–Jpeg video output. The computation consists of checking if $z_{n+1} = z_n^2 + c$ converges to a fixed point for $c \in \mathbb{C}$ and $z_0 = 0$. This computation is very local, takes place entirely in the data cache, and benefits from the posted write provided by a write buffer in the write-through cache. As opposed to TNR, it cannot be well parallelized *a priori* since we do not know in advance for which pixels the computation will quickly diverge or at the opposite converge. For the simulation, we ran the computation of four frames (represented Fig. 14) on four processors, and for one of these frames, we ran the computation on 1–16 processors. The parameters of the application are presented in Table 3.

The maximum number of iterations is chosen so that the image has a good final rendering, i.e. high enough so that all the points converging could be colored (and not considered divergent).

We ran the frame computations on the basic architecture for PAR and AWS.

Table 3
Parameters of the frame computations.

Image size	800×600
Max iterations	Up to 5000 for frame 4
Center coordinates	$(-0.74364421961; 0.13182604688)$
Zoom relative to image 1	$\{1, 5.64, 1.02 \times 10^6, 4.53 \times 10^6\}$

Table 4
Mandelbrot results on four processors (times in Kcycles).

Frame	Processing time with PAR	Processing time with AWS	Speedup relative to PAR	Number of steals	% of pixels stolen
1	5212	2958	1.76	34	29.8
2	11,995	6139	1.95	49	30.7
3	20,549	13,706	1.50	19	15.0
4	59,269	25,591	2.31	42	18.9

7.4. Mandelbrot results

The computation results for four processors are shown in Table 4.

These results exhibit that AWS outperforms PAR on this application, with a speedup ranging from 1.5 to 2.31. Even if the number of frames is too small to allow a generalization, this already proves that AWS can do better than a static parallelization in the case of unbalanced parallel computations.

In addition to the computation of the frames in AWS, we output for each frame an image which identifies the processor having computed a particular pixel (Fig. 15). This allows to visualize the repartition of the work on the different processors in this particular cases, and how the stealing mechanism impacts the computation.

Finally, Fig. 16 shows the computation times of frame #4 on 1–16 processors. This exhibits that the time saved by AWS dynamic load balancing is not dependant from the number of processors as there is approximately a factor 2 for 2, 4, 8 and 16 processors.

8. Summary and future work

In this paper, we first focused on the problem of MPSoC architecture design choices for AWS algorithms. We showed, using a synthetic application, and thanks to the capability of adaptation of the hardware and low level software to a specification template provided by the codesign approaches, that important performance improvements could be reached for both AWS and PAR strategies. We then focused on the overhead of AWS algorithms compared to statically scheduled parallel algorithms, and put in evidence that the cost of dynamic load balancing was not negligible when the input data size was too small; we also found that distributing data or

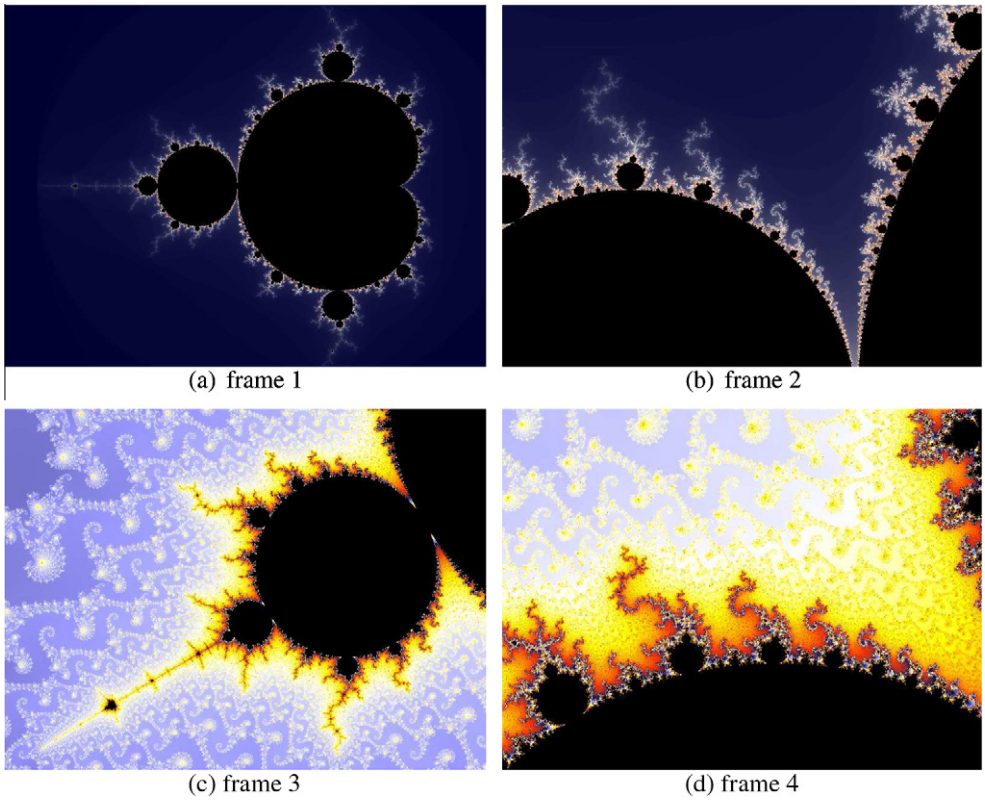


Fig. 14. Frames computed in simulation.

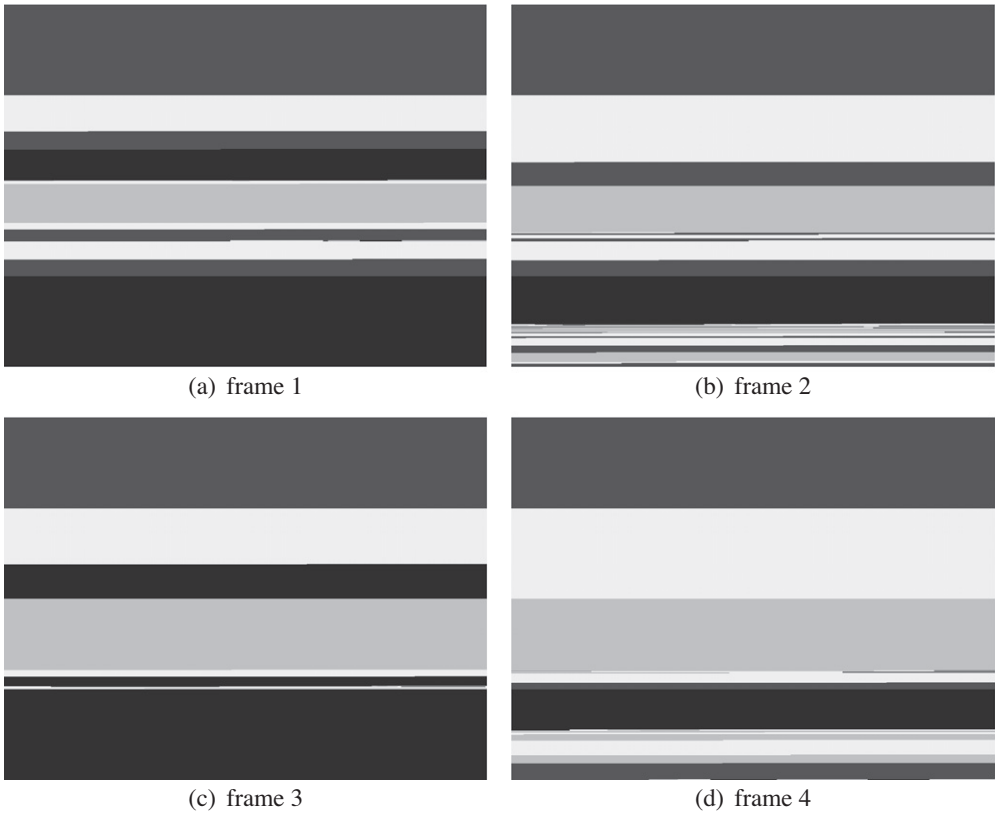


Fig. 15. Visual representation of the work on the processors for the Mandelbrot Frames. Legend: ■, Proc. 0; ■, Proc. 1; ■, Proc. 2; ■, Proc. 3.

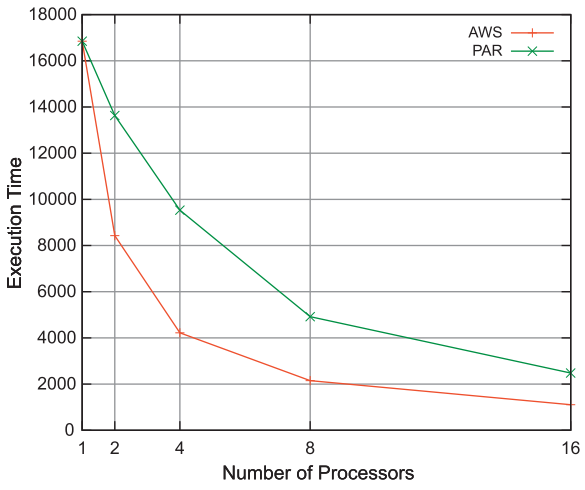


Fig. 16. Execution Time for the computation Mandelbrot frame #4.

locks which are mainly accessed locally can drastically improve performances. Finally, we applied these results on real applications, and showed that AWS could lead to a significant gain in case of applications with a non-uniform workload, while its overhead remained low for applications with a uniform workload.

The architecture model considered in this work, though being realistic for MPSoCs, remains simple and does not take into account the possibility of having higher levels of memory and/or interconnect hierarchy, or several processors located on the same local interconnect. We also limited our study to 16 processors, though in the near future, architectures with many more processors are conceivable.

Despite these limitations, and because we think that the AWS programming paradigm can be very useful including in the embedded domain, we believe our work to be important as a first study of codesign for this class of algorithms on MPSoC architectures.

Future work includes the evaluation of the influence of the cache properties (size and number of words per line) and of the use of spin locks instead of mutex locks. We also aim at applying lock-free parallel algorithms [38,39] to AWS in order to take advantage of the properties of this type of algorithms, while keeping the efficiency of AWS parallelization.

Acknowledgments

We wish to thank the anonymous reviewers for their detailed and helpful comments which helped us improve greatly the quality of this article.

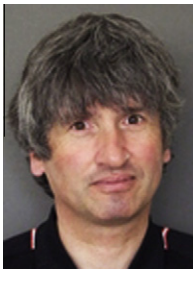
References

[1] A. Duller, D. Towner, G. Panesar, A. Gray, W. Robbins, Picoarray technology: The tool's story, in: Proceedings of the Conference on Design, Automation and Test in Europe, Munich, Germany, 2005, pp. 106–111.
[2] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J.F. Brown III, A. Agarwal, On-chip interconnection architecture of the tile processor, IEEE Micro 27 (5) (2007) 15–31.
[3] S. Edwards, L. Lavagno, E.A. Lee, A. Sangiovanni-Vincentelli, Design of embedded systems: formal Models, validation, and synthesis, Proceedings of the IEEE 85 (3) (1997) 366–390.
[4] G. Kahn, The semantics of a simple language for parallel programming, in: Proceedings of Information Processing, vol. 74, Stockolm, Sweden, 1974, pp. 471–475.
[5] M. Duranton, The challenges for high performance embedded systems, in: Proceedings of the Ninth Euromicro Conference on Digital System Design, Dubrovnik, Croatia, 2006, pp. 3–7 (keynote address).
[6] E. De Kock, W. Smits, P. van der Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, K. Vissers, G. Essink, YAPI: application modeling for signal processing systems, in: Proceedings of the 37th Annual Design Automation Conference, ACM, 2000, pp. 402–405.
[7] C. Bergeron, C. Lamy-Bergor, Complaint selective encryption for H. 264/AVC video streams, in: 2005 IEEE 7th Workshop on Multimedia Signal Processing, 2005, pp. 1–4.

[8] A. Agbaria, D.-I. Kang, K. Singh, Lmpi: Mpi for heterogeneous embedded distributed systems, in: Proceedings of the 12th International Conference on Parallel and Distributed Systems, IEEE, Minneapolis, MN, 2006, pp. 79–86.
[9] P. Paulin, C. Pilkington, E. Bensoudane, Stepnp: a system-level exploration platform for network processors, IEEE Design and Test of Computers 19 (6) (2002) 17–26.
[10] J. Oh, S.W. Kim, C. Kim, OpenMP and compilation issue in embedded applications, in: Proceedings of the International Workshop on OpenMP Applications and Tools, Lecture Notes in Computer Science, vol. 2716, Springer, Toronto, Canada, 2003, pp. 109–121.
[11] H. Blume, J. von Livonius, L. Rotenberg, T.G. Noll, H. Bothe, J. Brakensiek, OpenMP-based parallelization on an mpcore multiprocessor platform – a performance and power analysis, Journal of Systems Architecture – Embedded Systems Design 54 (11) (2008) 1019–1029.
[12] D. Hommais, F. Pétrot, I. Augé, A practical tool box for system level communication synthesis, in: Proceedings of the Ninth International Symposium on Hardware/Software Codesign, ACM, Copenhagen, Denmark, 2001, pp. 48–53.
[13] E. Faure, A. Greiner, D. Genius, A generic hardware/software communication mechanism for multi-processor system on chip, targeting telecommunication applications, in: Proceedings of the 2nd International Workshop on Reconfigurable Communication-centric Systems-on-Chip, Montpellier, France, 2006, pp. 237–242.
[14] M. Gries, Methods for evaluating and covering the design space during early design development, Integration, the VLSI Journal 38 (2) (2004) 131–183.
[15] A.A. Jerraya, W. Wolf, Hardware/software interface codesign for embedded systems, Computer 38 (2) (2005) 63–69.
[16] N.S. Arora, R.D. Blumofe, C.G. Plaxton, Thread scheduling for multiprogrammed multiprocessors, Theory of Computing Systems 34 (2) (2001) 115–144.
[17] M.A. Bender, M.O. Rabin, Online scheduling of parallel programs on heterogeneous systems with applications to Cilk, Theory of Computing Systems 35 (2002) 2002.
[18] J. Bernard, J.-L. Roch, D. Traoré, Processor-oblivious parallel stream computations, in: PDP, IEEE Computer Society, 2008, pp. 72–76.
[19] E. Mohr, D.A. Kranz, R.H. Halstead Jr., Lazy task creation: a technique for increasing the granularity of parallel programs, IEEE Transactions on Parallel and Distributed Systems 2 (3) (1991) 264–280.
[20] M. Frigo, C.E. Leiserson, K.H. Randall, The implementation of the Cilk-5 multithreaded language, in: Programming Language Design and Implementation, 1998, pp. 212–223.
[21] D.P. Papadopoulos, Hood: A User-level Thread Library for Multiprogramming Multiprocessors, Ph.D. Thesis, The University of Texas at Austin, Sepember 21, 1998.
[22] D. Traoré, J.-L. Roch, N. Maillard, T. Gautier, J. Bernard, Deque-free work-optimal parallel stl algorithms, in: Euro-Par 2008 Parallel Processing, Lecture Notes in Computer Science, 2008, pp. 887–897.
[23] A. Matache, S. Dolinar, F. Pollara, Stopping rules for turbo decoders, in: JPL TMO Progress Report, 2000, pp. 42–142.
[24] M. Mattavelli, S. Brunetton, D. Mlynek, Computational graceful degradation for video sequence decoding, in: Proceedings of the International Conference on Image Processing, 1997, pp. 330–333.
[25] K. Agrawal, C.E. Leiserson, Y. He, W.-J. Hsu, Adaptive work-stealing with parallelism feedback, ACM Transactions on Computer Systems 26 (3) (2008).
[26] T. Hiraishi, M. Yasugi, S. Umatani, T. Yuasa, Backtracking-based load balancing, SIGPLAN Not 44 (4) (2009) 55–64.
[27] S. Chen, P.B. Gibbons, M. Kozuch, V. Liakovitis, A. Ailamaki, G.E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T.C. Mowry, C. Wilferson, Scheduling threads for constructive cache sharing on CMPs, in: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, San Diego, 2007, pp. 105–115.
[28] O. Certner, Z. Li, P. Palatin, O. Temam, F. Arzel, N. Drach, A practical approach for reconciling high and predictable performance in non-regular parallel programs, in: Proceedings of the Conference on Design, Automation and Test in Europe, Munich, Germany, 2008, pp. 740–745.
[29] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, C. Kozyrakis, Comparing memory systems for chip multiprocessors, in: 34th International Symposium on Computer Architecture, ACM, San Diego, California, 2007, pp. 358–368.
[30] M. Chu, R. Ravindran, S. Mahlke, Data access partitioning for fine-grain parallelism on multicore architectures, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007, pp. 369–380.
[31] B. Saglam, V. Mooney III, System-on-a-chip processor synchronization support in hardware, in: Proceedings of the Conference on Design, Automation and Test in Europe, IEEE, Munich, Germany, 2001, pp. 633–641.
[32] A. Sheibanyrad, A. Greiner, I. Miro-Panades, Multisynchronous and fully asynchronous nocs for gals architectures, IEEE Design and Test of Computers 25 (6) (2008) 572–580.
[33] P.R. Panda, N.D. Dutt, A. Nicolau, Efficient utilization of scratch-pad memory in embedded processor applications, in: Proceedings of the 1997 European Conference on Design and Test, Paris, France, 1997, pp. 7–11.
[34] F. Pétrot, P. Gomez, Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect, in: Proceedings of the Design Automation and Test in Europe, Embedded Software Forum, Munich, Germany, 2003, pp. 10182–10187.
[35] P. Feautrier, Dataflow analysis of array and scalar references, International Journal of Parallel Programming 20 (1) (1991) 23–53.
[36] P. Guironnet de Massas, F. Pétrot, Comparison of memory write policies for noc based multicore cache coherent systems, in: Proceedings of the Conference on Design, Automation and Test in Europe, Munich, Germany, 2008, pp. 997–1002.
[37] The Soclib Consortium, Soclib: an open platform for virtual prototyping of multi-processors system on chip, Technical report, 2008. <http://www.soclib.fr>.
[38] K. Fraser, T. Harris, Concurrent programming without locks, ACM Transactions on Computer Systems 25 (2) (2007).
[39] M. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems 13 (1991) 124–149.

Quentin L. Meunier received the M.S. degree in Computer Science from Ensimag (Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées), Grenoble, France, in 2007. He worked in the Verimag Laboratory as a master student in 2007 on timed models and aspect-oriented programming. He is currently a Ph.D. candidate in TIMA Laboratory, SLS Group. His current interests include multiprocessor architectures, parallel programming, and more specifically transactional memories.

Frédéric Pétrot received the Ph.D. degree in Computer Science from Université Pierre et Marie Curie (Paris VI), Paris, France, in 1994, where has been Assistant Professor in Computer Science until September 2004. From 1989 to 1996, F. Pétrot was one of the main contributors of the open-source Alliance VLSI CAD system whose research team received the French Seymour Cray award in 1994. Since 1996, he headed the work on the definition and implementation of the Disydent environment, oriented toward the specification and implementation of multiprocessor SoCs. He joined TIMA in September 2004, and holds a professor position at the Institut Polytechnique, Grenoble, France. Since 2006, he heads the System Level Synthesis Group of TIMA.



Within laboratory LIG (and previously laboratories ID-IMAG [1999–2006] and LMC-IMAG [1986–1999]), **Jean-Louis Roch** research has largely been in the interaction of parallel algorithms and their programming on parallel and/or distributed architectures: supercomputers; clusters and grid; SMP, multicore and embedded systems. He is currently heading the MOAIS team-project (2005-) of INRIA, where he studies adaptive parallel algorithms and their scheduling in the context of interactive applications and on multiprocessor system on-chips. He also work on security and fault-tolerance for global computations. He is currently assistant professor at the Ensimag, Institut Polytechnique de Grenoble, France.