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*
[b] *LIG, INP Grenoble and INRIA*

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.

Key words: MPSoC Architectures, Concurrent Programs, Work Stealing, Hardware/Software Codesign, Design Space Exploration

[*] Corresponding author. Tel.:+33 4 76 57 48 70; Fax:+33 4 76 57 49 81.
Email addresses: Quentin.Meunier@imag.fr (Quentin Meunier), Frederic.Petrot@imag.fr (Frédéric Pétrot), Jean-Louis.Roch@imag.fr (Jean-Louis Roch)

1. Introduction

Even though in the past years many if not all embedded system 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.

Most of the algorithms used in the devices that are MPSoCs based are currently essentially implemented 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]. 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[6], light versions of Corba[7], OpenMP[8, 9], 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 [7] that introduces hardware support to Corba-like communication and efficient SMP task managment and [10, 11] that define specific hardware IP and software APIs to automate the mapping of process networks with different kinds of FIFOs. [12] provides a very broad survey on the general codesign approaches, whereas [13] 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[14, 15]. We focus on a specialization of work-stealing based algorithms, denoted in the following AWS (Adaptive Work-Stealing) [16]. 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* [17, 18, 19, 20], 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 [21] or video decoding [22].

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 behaviour to drive the definition of both a hardware architecture and the run-time library that 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 that 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 run times 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 [23]: whenever a processor runs out of work, it steals work from a randomly chosen processor.

From work stealing reference implementation Cilk [18], 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, Plaxton and Blumofe [14] showed that for any parallel program, the time T_p on p identical processors verifies with high probability (w.h.p.): $T_p \leq O(\frac{T_{seq}}{p} + T_\infty)$ 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 [14] and for processors with variable speeds [15]. In particular, the number of steal requests is $O(pT_\infty)$ 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 $O(p.T_\infty)$ 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 [20], 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" [20] creates a new task which is assigned to the theft processor. Similarly, in the *Tascell* framework [24], 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 [24].

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 [18]. Considering CMPs, [25] 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 [26]. These results are obtained on a predefined 4 cores platform and do not explore alternatives implementations of data placement and synchronization.

In [16] 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 behaviour. This work however does not encompass the embedded field nor architecture properties and remains mainly theoretical.

[27] 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[28] 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 fix 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 Figure 1, the platform is an interconnection of CPU sub-systems. All CPU sub-systems 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[29]. 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 [30], 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.

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

Table 1: Simulated platforms characteristics

Number of processors	$p = \{1, ..., 16\}$
Number of memory banks	$p + 3$
Processor model	SPARC-V8 with FPU
Data cache size	$16Kb$
Data block size	8 words (32 bytes)
Instruction cache size	$16Kb$
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

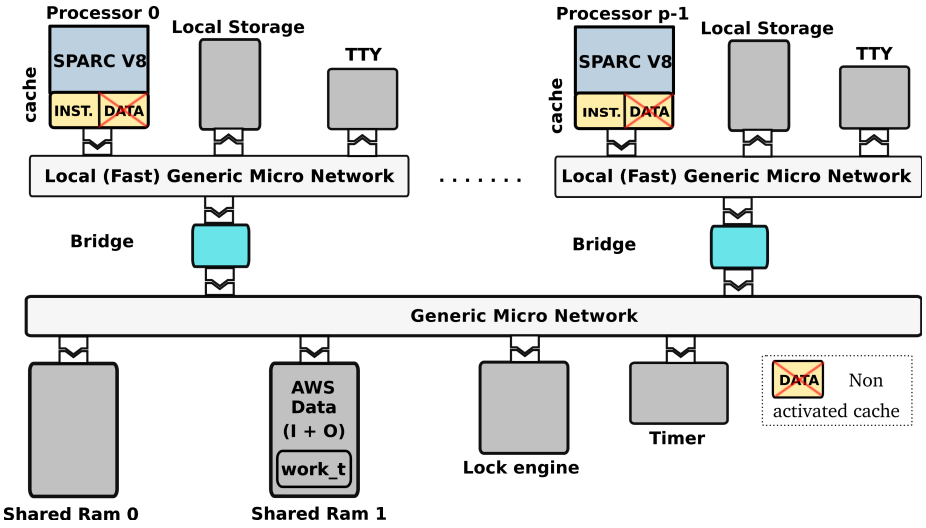


Figure 1: Basic architecture schematic view

a dedicated interface [31]), but this wasn't explored as being a limitation of processor models available in the simulation environment. However, thanks to the local crossbar latency, the timing behavior of both 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 speed-up synchronization.

3.2. Operating System and Task Assignment

We use the Decentralized Scheduling (DS) configuration of a lightweight kernel called Mutek[32], 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, in order to evaluate the overhead of work-stealing with respect to classical standard approaches on embedded systems, the template has to enable calibration at the cycle level of applications for which optimal parallelization 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 $\frac{n}{p}$.

Besides, locally on each hardware processing unit, the processing on each element can be achieved either by a generic or by a dedicated optimized component for the unit. Such local choice is implicitly managed by work stealing which manages the execution of the applications template on the hardware platform. Since work stealing is a fully distributed algorithm, independent on one hand from the number, the configuration and frequencies of processors, and on the other hand from the granularity of the application, it enables codesign by global calibration of the platform.

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 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. 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.

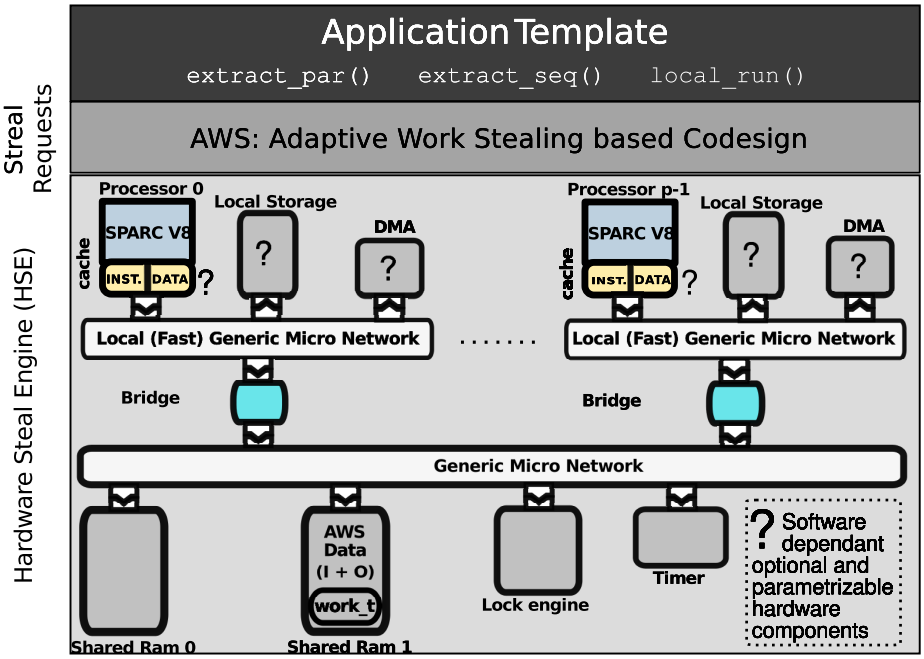


Figure 2: AWS: adaptive work-stealing based codesign

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 [20] which is based on lazy task creation. A non idle processor *j* performs the computation of its assigned task; when a steal request from a processor *i* occurs on *j*, a new task is created corresponding to a ready part of the remaining work on *j*; then this task is assigned to processor *i* which starts its execution. The operation that constructs the description of the task (inputs) is named `extract_seq()` in the following; it is implemented at the application level. Thus, effective task creation is delegated to the application: AWS only manages the steal requests on the idle processors and the local execution of the local work on a non idle one.

Both operations are dependent: a successful steal operation leads to an `extract_par()` on the victim which modifies the remaining work. This synchronization between the steal requests and the local work is managed at the work-stealing level. To enable a wait-free implementation, the local work on a non-idle processor is performed by a block of operations. These blocks are assumed wait-free and their definition is delegated at the application level: a function named `extract_par()` serially iterates through the local work until its completion.

Finally, while structuring the implementation of the work-stealing strategy, it is to be noted that this on-line coupling of two algorithms, a sequential one (from `extract_seq()` operations) and a parallel one (from `extract_par()` operations), is not restrictive. On a first hand, recursive parallelism may be implemented at the application level by managing, inside the local work, a collection of the future tasks to be stolen; of course, memory constraints have then to be considered at the application level. On the other hand, the scheduling of ready tasks on each processing unit is well known to have a crucial impact on the performances. For instance, in [25], the classical oldest-first ready task strategy (OWS) is compared to the parallel depth-first strategy (PDF) that globally schedules tasks in a way that tracks the sequential execution. Considering several applications, while OWS is generally more attractive than PDF with respect to the number of steal requests, it compares poorly to PDF in terms of L2-cache miss on architectures with cache-sharing between processors [25]. Yet, the delegation of `extract_par()`/`extract_seq()` operations enables to tune the global scheduling at the application level.

The main drawback of the proposed design is that for a complex application, all induced synchronizations must be managed in the application code. However, this is not necessarily a problem: several data processing applications, such as the ones considered in the previous applications template, do not require complex synchronizations. Besides, a higher level library based on a wait-free implementation can efficiently manage synchronizations at the application level on top of AWS [14, 24].

4.2. Adaptive Work-Stealing Implementation

Based on previous design choices, this paragraph presents AWS implementation and application programming interface 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.

```
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) {    /* micro-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 micro 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);
```

Figure 3: Core of the AWS algorithm.

The adaptive work-stealing algorithm is presented in Algorithm 4.2. After a successful steal request, a processor gets a work w and executes two nested loops: the steal-loop and the micro-loop. In the micro-loop, the processor executes a function called `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 a function called `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 gets out from the micro-loop. It scans all the public `work_t` structures until it finds a non-empty one and then executes the `extract_par()` function which extracts some work

from the victim work v in its own public `work_t` structure w. The processor then enters the micro-loop again. When all the public `work_t` structures are empty, the whole work has been computed.

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. For the case where the input data is a table, Figure 4 illustrates the node structure which

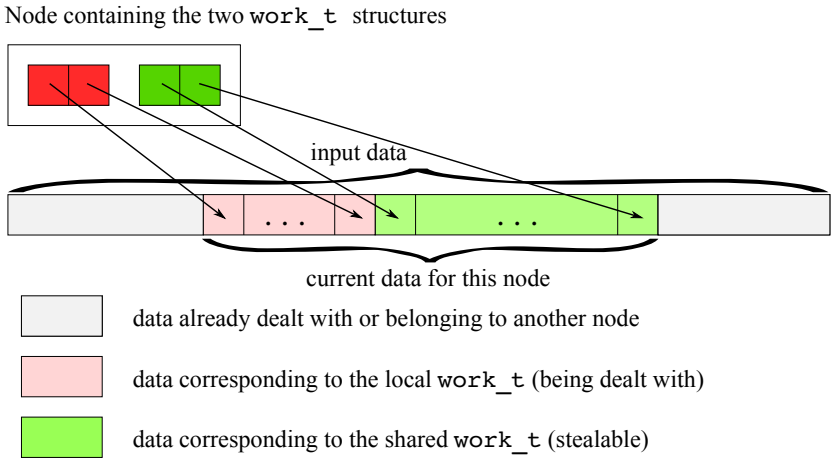


Figure 4: Data contained in a node

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. To bound the scheduling overhead, the `extract_seq()` function must extract log of the remaining amount of work and `extract_par()` must extract a fraction, generally half, of the remaining work. By a theoretical analysis of the size of the chunks for `extract_par()` and `extract_seq()` operation, the implementation guarantees asymptotic time optimality of stream computations while being processor-oblivious (i.e. it does not depend on the number of processors [16]). Initially, the input data is pre-allocated between the processors. After a successful steal operation, the `extract_par()` extracts a half of the remaining interval of the victim. The `extract_seq()` extracts the first $\log_2$ 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 that are static control programs [33]. Moreover, we will first consider a low complexity and fine grained 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 $\tau_{min} \leq \tau \leq \tau_{max}$. In the following section, we consider that the instruction caches can hold the whole application and we restrict the analysis to the data caches.

4.3. 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 processor verifies $M_{seq} \leq p \times M_p^{PAR} \leq M_{seq} + p$. 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

$$T_p^{PAR} \simeq \frac{T_{seq}}{p}. \tag{1}$$

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

$$\frac{T_{seq}}{p} \leq T_p^{PAR} \leq \frac{\tau_{max}}{\tau_{min}} \frac{T_{seq}}{p}. \tag{2}$$

4.4. 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 $\log_2$, we have that $T_\infty = O\left(\log_2 \frac{T_{seq}}{p}\right)$. Moreover, due to the cyclic search of a victim processor, the total number of steal operations is $\#S = O(p \times T_\infty)$. w.h.p. and in the worst case

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

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: $M_{seq} \leq p \times M_p^{AWS} \leq M_{seq} + 2\#S$. 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:

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

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 (Figure 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.

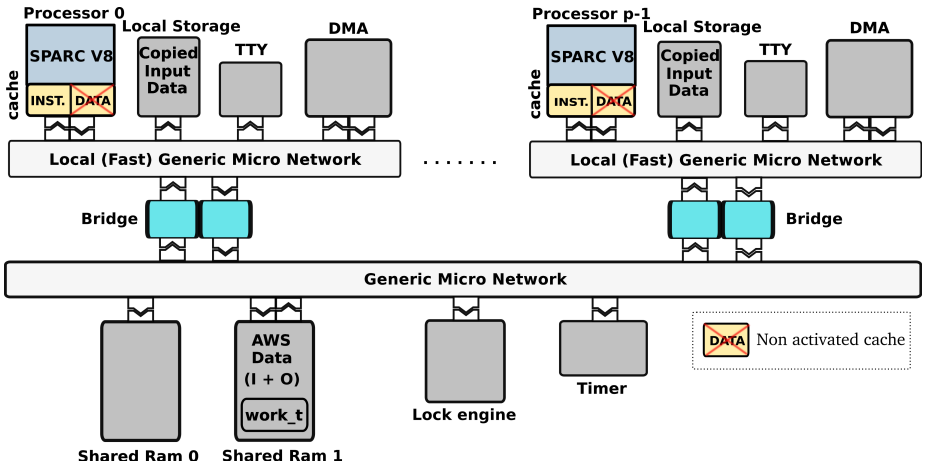


Figure 5: Architecture with DMAs

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 transfered 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 (`local_run()`). The alternative strategy is to program the DMA at the beginning of 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 (Figure 6) detailed in in [34].

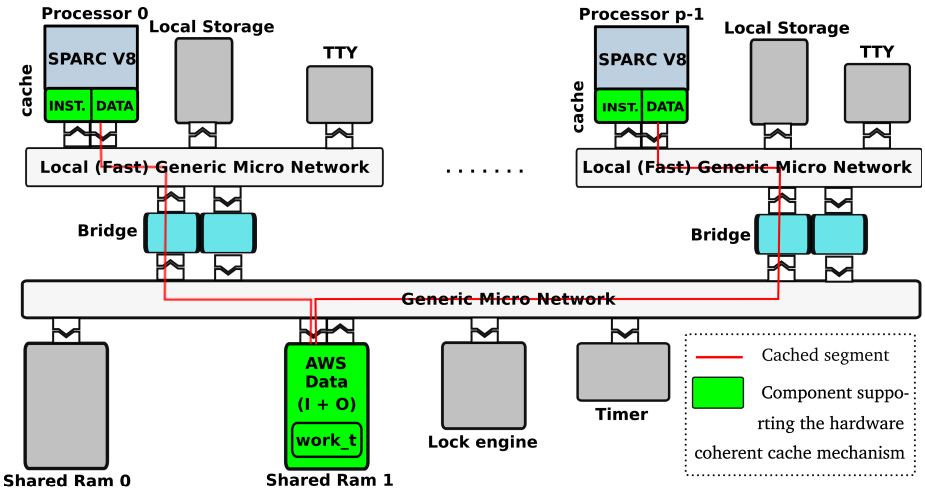


Figure 6: Architecture with coherent caches

5.3. Using Caches and DMAs

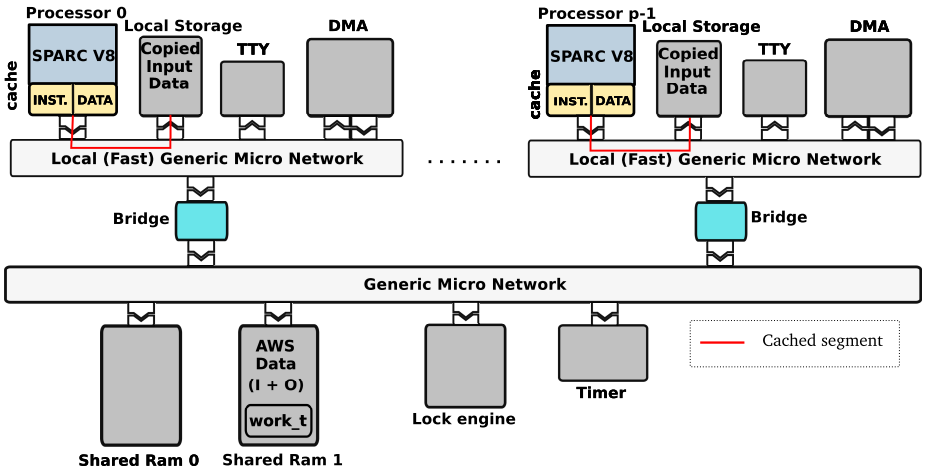


Figure 7: Architecture with caches and DMAs

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 (Figure 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 that 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.

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 [15]). 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 Figure 1 with shared values uncached), that provides reference performance measures,
- the architecture with coherent caches.

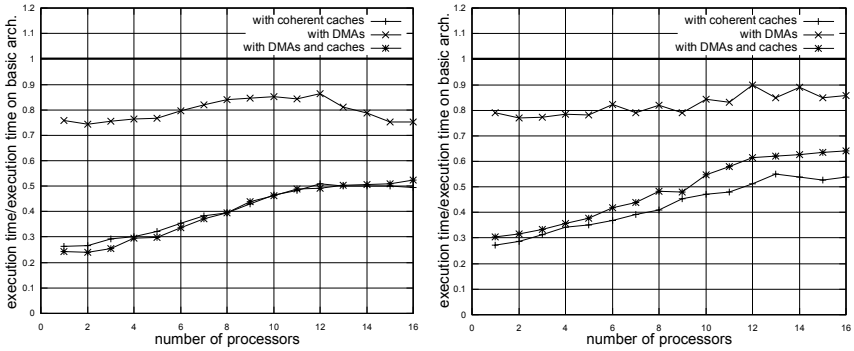
6. Results and Analysis

The simulations were done using the Cycle-Accurate Bit-Accurate (CABA) SystemC models of the SOCLib[35] 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 4.2, the number of elements extracted via the `extract_seq()` function must be $O(\log(m))$, where m is the number of remaining elements.

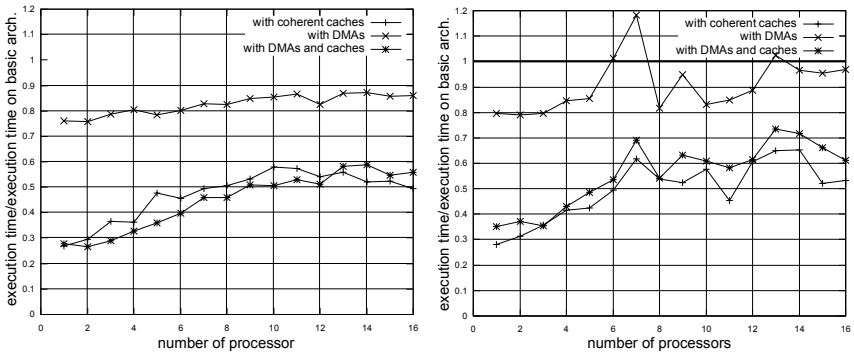
6.1. Comparing the Execution Times on Different Architectures

The Figures 8(a) and 8(b) show 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.



(a) PAR (b) AWS

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



(a) PAR (b) AWS

Figure 9: Normalized execution times on the different architectures for 1 to 16 processors with 10k elements

The first noticeable information is that the simulation times obtained with AWS and PAR are close for the 3 configurations, but a bit less regular with AWS compared to PAR. This can be explained by the traffic due to the final synchronisation in AWS. In fact, depending on the order of all the requests sent in this phase ($O(p^2)$), 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 both PAR and AWS for this architecture. 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 synchronizations overheads, which represent a higher percentage of the total execution time with many processors.

The Figures 9(a) and 9(b) show 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 at a too fine granularity, even with optimized hardware support.

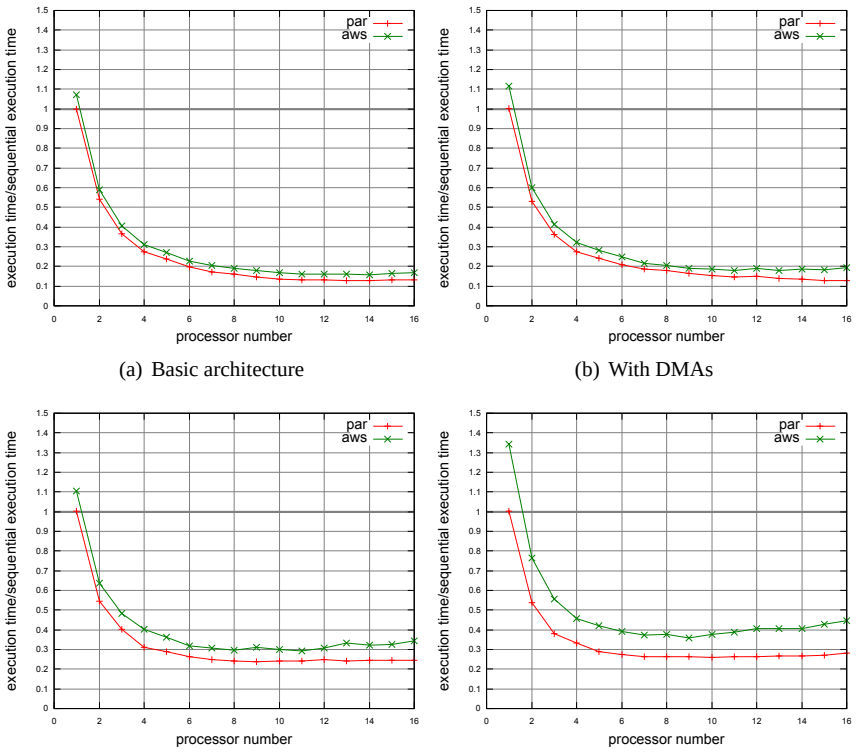
6.2. Comparing Sequential and Parallel Execution Times on a Given Architecture

Figure 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 7 for the slowest architecture and only 4 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 Figure 11.

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



(a) Basic architecture (b) With DMAs

(c) With coherent caches (d) With DMAs and caches

Figure 10: Normalized execution times on the different architectures for 1 to 16 processors with 100k elements

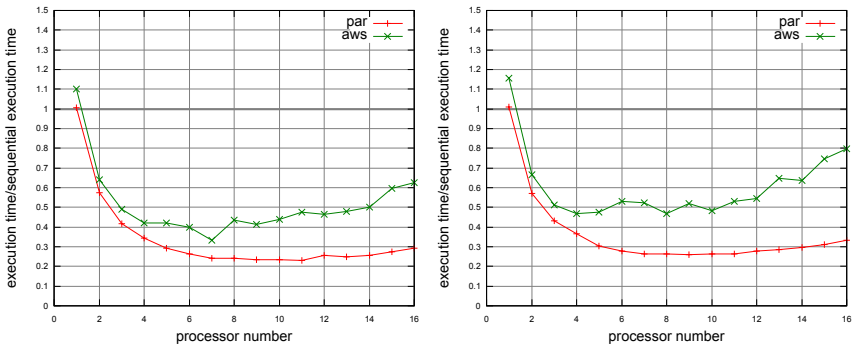
around 4, while AWS execution times on the fastest architectures (DMAs + caches and coherent caches) quickly go beyond the sequential execution time.

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.

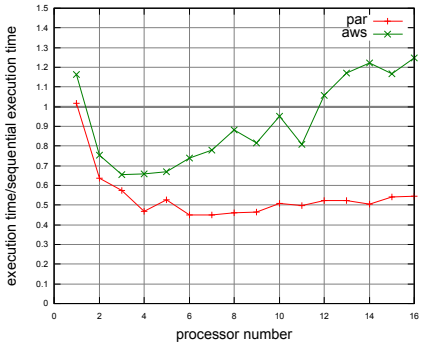
6.3. Distributed Locks and Work Structures

On Figure 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

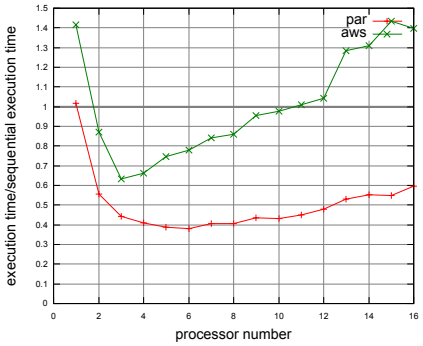


(a) Basic architecture

(b) With DMAs



(c) With coherent caches



(d) With DMAs and caches

Figure 11: Normalized execution times for different architectures for 1 to 16 processors with 10k elements

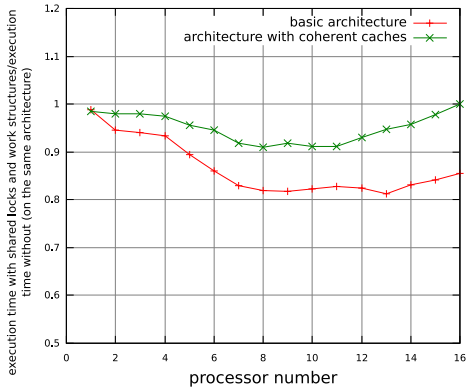


Figure 12: Normalized execution times with distributed locks and work structures, on two architectures

time-savings since we are in a case in which there are almost no steals. 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. Comparison of Theoretical and Practical AWS Overhead

As seen in Equation 3, the theoretical AWS overhead is quadratic in the number of processors when the processor on which work is to be stolen are traversed cyclically. Figure 13 plots the overheads (*i.e.* the sum of the cycles spent on all processors minus the sequential time) for the synthetic application with 100k elements and on the basic architecture. On the same figure is also plotted a quadratic regression of the data, thus showing that the expected theoretical results are confirmed by our experiments.

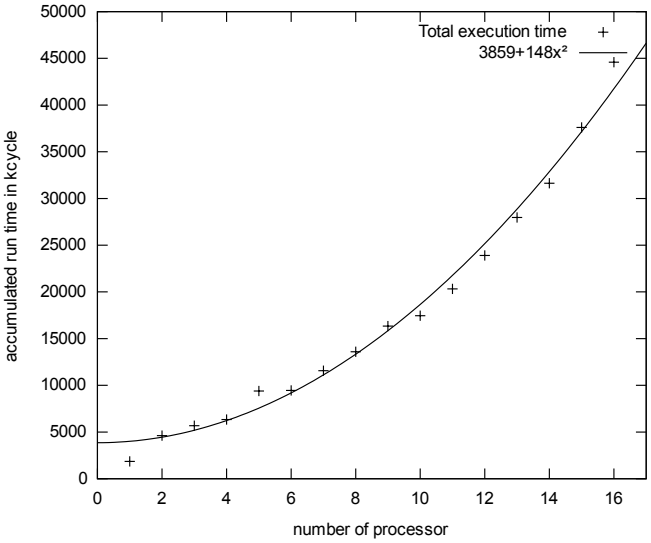


Figure 13: AWS overhead behavior

6.5. 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*[27].

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.

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.

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 714x244.

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

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 6 frames on 4 processors are presented in Table 7.2. 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 Figure 14. This graph shows the average decoding time for 4 frames. As with 4 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.

Frame	Processing Time on		
	Basic PAR	Basic AWS	Enhanced AWS
1	6 512	6 514	5 559
2	6 527	6 518	5 571
3	6 529	6 531	5 576
4	6 532	6 531	5 578
5	6 529	6 527	5 571
6	6 525	6 523	5 567

Table 2: TNR Execution Times (in KCycles)

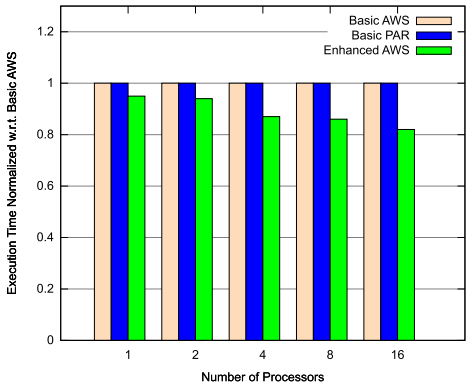


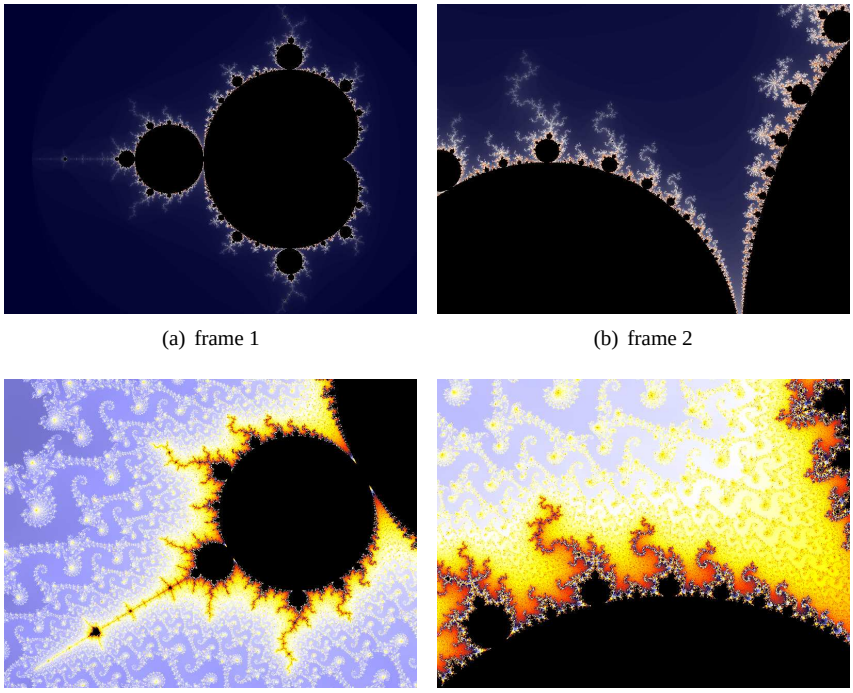
Figure 14: TNR Execution Time for the decoding of 4 frames, Normalized w.r.t. Basic AWS

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 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 don't know in advance for which pixels the computation will quickly diverge or at the opposite converge. For the simulation, we ran the computation of 4 frames (represented Figure 15) on 4 processors, and for one of these frames, we ran the computation on 1 to 16 processors. The parameters of the application are presented in Table 7.3.



(a) frame 1 (b) frame 2

(c) frame 3 (d) frame 4

Figure 15: Frames Computed in Simulation

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 3: Parameters of the Frame Computations

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.

7.4. Mandelbrot Results

The computation results for 4 processors are shown in table 7.4.

Frame	Processing time with PAR	Processing time with AWS	Speedup relative to PAR	Number of steals	% of pixels stolen
1	5,212	2,958	1.76	34	29.8
2	11,995	6,139	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

Table 4: Mandelbrot Results on 4 processors (times in Kcycles)

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 generalisation, 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 (Figure 16). 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, Figure 17 shows the computation times of frame #4 on 1 to 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 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

(a) frame 1 (b) frame 2

(c) frame 3 (d) frame 4

Figure 16: Visual representation of the work on the processors for the Mandelbrot Frames. Legend: ■ Proc. 0, ■ Proc. 1, ■ Proc. 2, ■ Proc. 3.

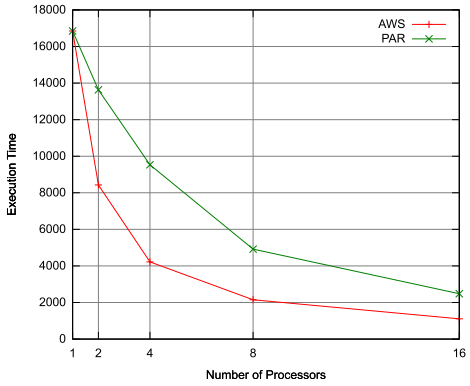


Figure 17: Execution Time for the computation Mandelbrot frame #4

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[36, 37] to AWS in order to take advantage of the properties of this type of algorithms, while keeping the efficiency of AWS parallelization.

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. B. 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, Proc. of the IEEE 85 (3) (1997) 366–390.

[4] G. Kahn, The semantics of a simple language for parallel programming, in: Proc. of Information Processing 74, Stockolm, Sweden, 1974, pp. 471–475.

[5] M. Duranton, The challenges for high performance embedded systems, in: Proceedings of the 9th Euromicro Conference on Digital System Design., Dubrovnik, Croatia, 2006, pp. 3–7, keynote address.

[6] 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.

[7] P. Paulin, C. Pilkington, E. Bensoudane, Stepnp: A system-level exploration platform for network processors, IEEE Design & Test of Computers 19 (6) (2002) 17–26.

[8] 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, Vol. 2716 of Lecture Notes in Computer Science, Springer, Toronto, Canada, 2003, pp. 109–121.

[9] 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.

[10] 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.

[11] 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.

[12] M. Gries, Methods for evaluating and covering the design space during early design development, Integration, the VLSI Journal 38 (2) (2004) 131–183.

[13] A. A. Jerraya, W. Wolf, Hardware/software interface codesign for embedded systems, Computer 38 (2) (2005) 63–69.

[14] N. S. Arora, R. D. Blumofe, C. G. Plaxton, Thread scheduling for multiprogrammed multiprocessors., Theory of Computing Systems 34 (2) (2001) 115–144.

[15] 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.

[16] J. Bernard, J.-L. Roch, D. Traoré, Processor-oblivious parallel stream computations, in: PDP, IEEE Computer Society, 2008, pp. 72–76.

[17] 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.

[18] 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.

[19] D. P. Papadopoulos, Hood: A user-level thread library for multiprogramming multiprocessors, Ph.D. thesis, The University of Texas at Austin (Sep. 21 1998).

[20] 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.

[21] A. Matache, S. Dolinar, F. Pollara, Stopping rules for turbo decoders, in: JPL TMO Progress Report, 2000, pp. 42–142.

[22] 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.

[23] K. Agrawal, C. E. Leiserson, Y. He, W.-J. Hsu, Adaptive work-stealing with parallelism feedback, ACM Transactions on Computer Systems 26 (3).

[24] T. Hiraishi, M. Yasugi, S. Umatani, T. Yuasa, Backtracking-based load balancing, SIGPLAN Not. 44 (4) (2009) 55–64.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] A. Sheibanyrad, A. Greiner, I. Miro-Panades, Multisynchronous and fully asynchronous nocs for gals architectures, IEEE Design & Test of Computers 25 (6) (2008) 572–580.

[31] 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.

[32] F. Pétrot, P. Gomez, Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect, in: Proc. of the Design Automation and Test in Europe, Embedded Software Forum, Munich, Germany, 2003, pp. 10182–10187.

[33] P. Feautrier, Dataflow analysis of array and scalar references, International Journal of Parallel Programming 20 (1) (1991) 23–53.

[34] 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.

[35] The Soclib Consortium, Soclib: an open platform for virtual prototyping of multiprocessors system on chip, Tech. rep., [Online]. Available: http://www.soclib.fr (2008).

[36] K. Fraser, T. Harris, Concurrent programming without locks, ACM Transactions on Computer Systems 25.

[37] M. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems 13 (1991) 124–149.