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

ΑΝΤΩΝΙΑΔΗΣ ΣΤΑΥΡΟΣ ΑΕΜ:8279

Contents

Introduction

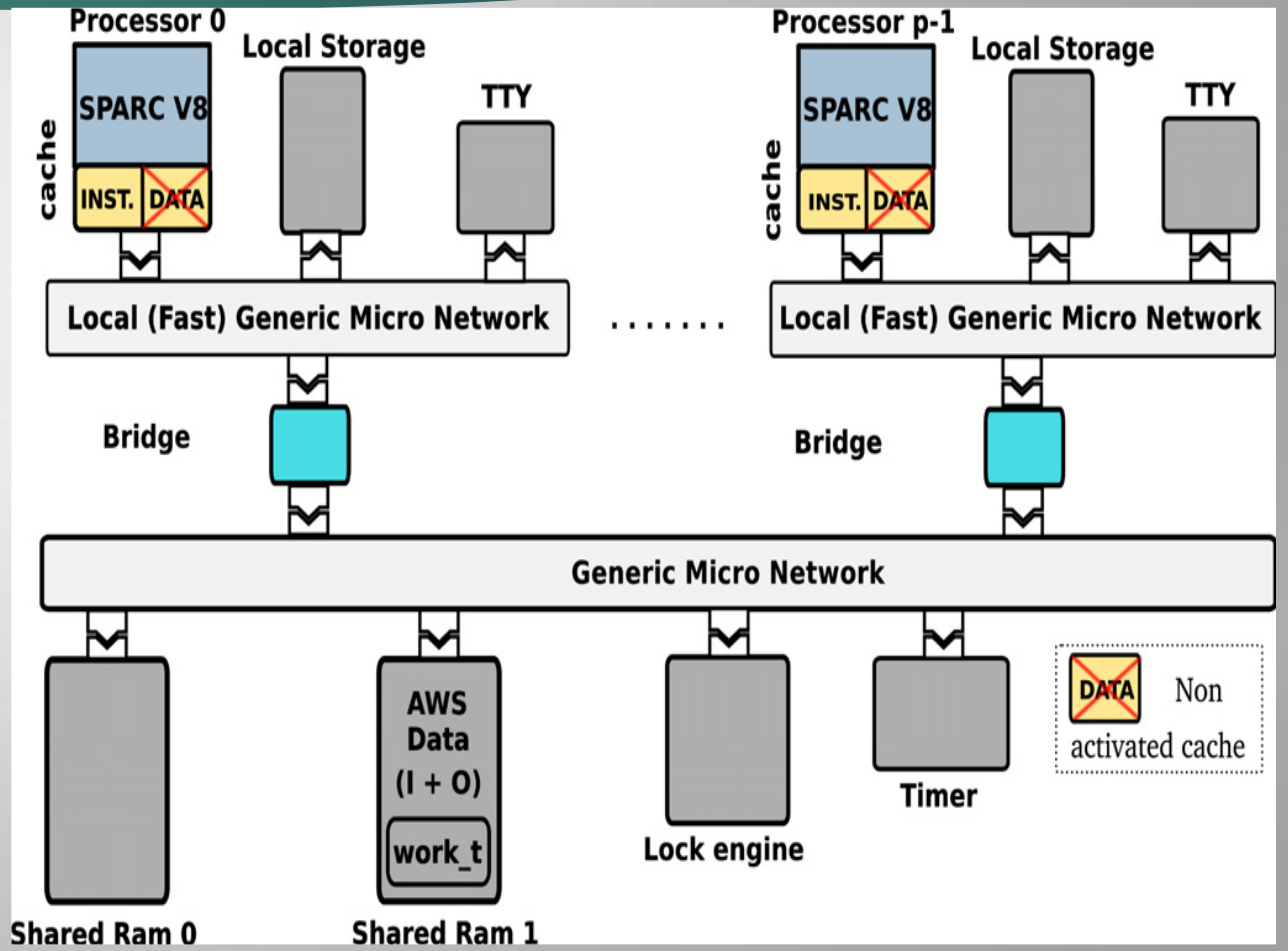
- AWS: Based on the principle that each processor executes its own task until it becomes idle, and then steals a fraction of the remaining work on a randomly chosen busy processor.

- PAR: Based on the pre-partitioning of the input into chunks of equal size.

- 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.

Hardware

- 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: root(2)n Cycles for n interfaces
- Local NoC latency: 1 Cycle



Operating System-Task Assigment

We use the Decentralized Scheduling (DS) configuration of a lightweight kernel called Mutek, which provides an implementation of the POSIX pthreads for shared memory multiprocessor machines.

- DS: each processor has its own scheduler and that's the reason why in this configuration contentions are limited
- SMP: all processors share a single scheduler structure to perform task selection

Selected applications template

Our choice of a template for the applications has been motivated by three points.

I. we must be able to accurately evaluate the overhead of work-stealing as compared to static optimal parallelization when it is known

II. 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)

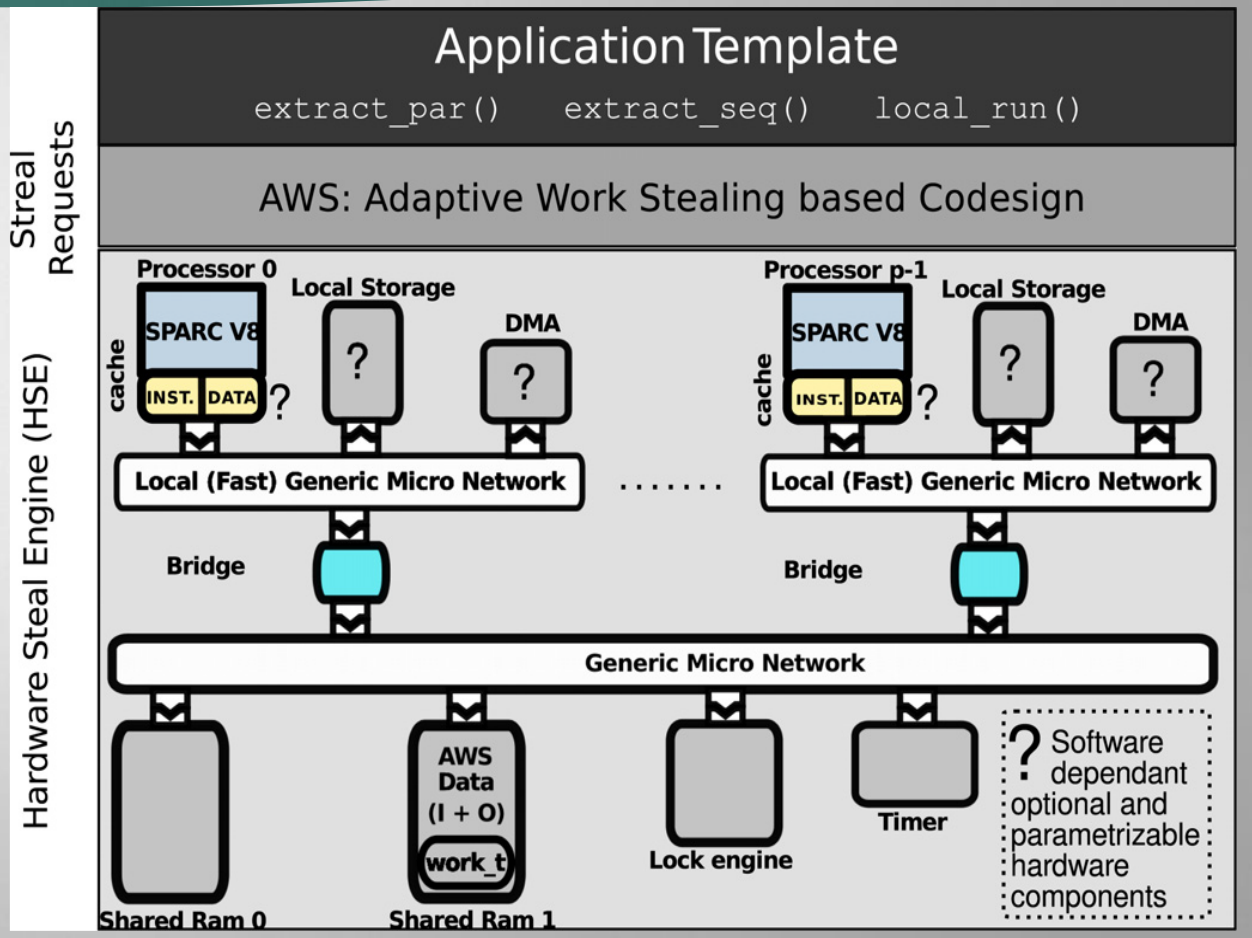
III. it should enable a theoretical analysis of the implementation of the work-stealing in order to have feedback on the experimentations

Synthetic Template

Codesign based on work-stealing: AWS

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.



AWS interface for data processing

- 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 a ✖ log(remaining data)
- the local_run() function, defining how to process a piece of data
- the ratio a of local extraction, if used in the extract_seq() function
- the threshold under which a steal operation fails

```
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.

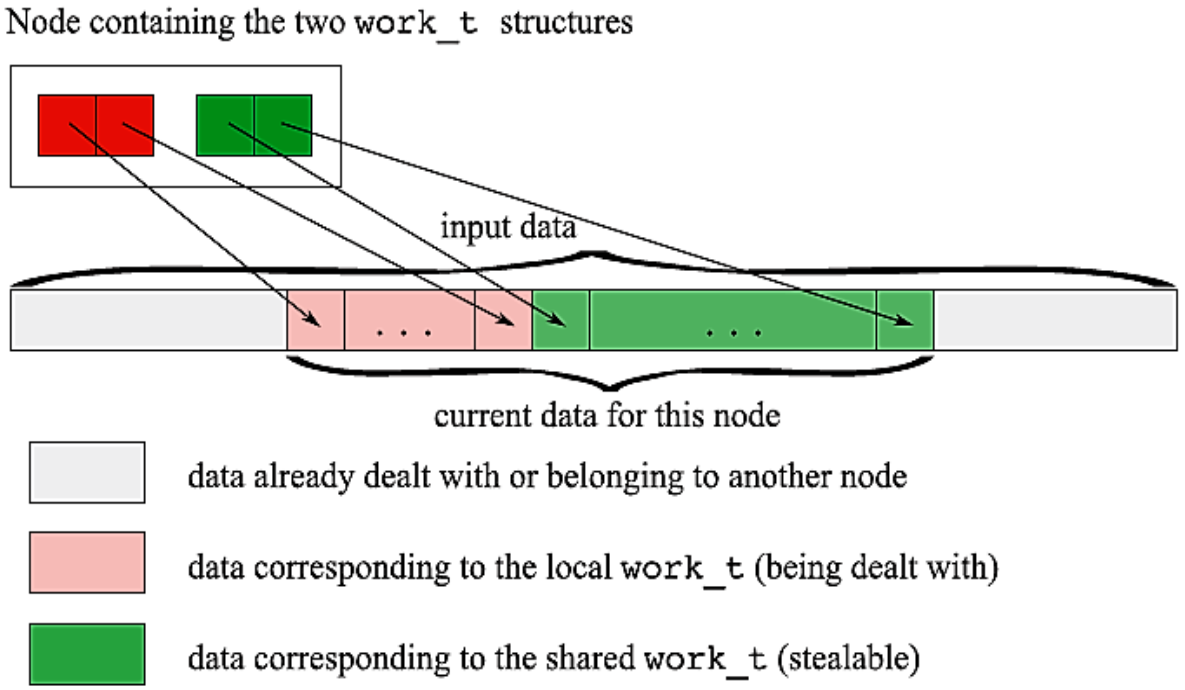
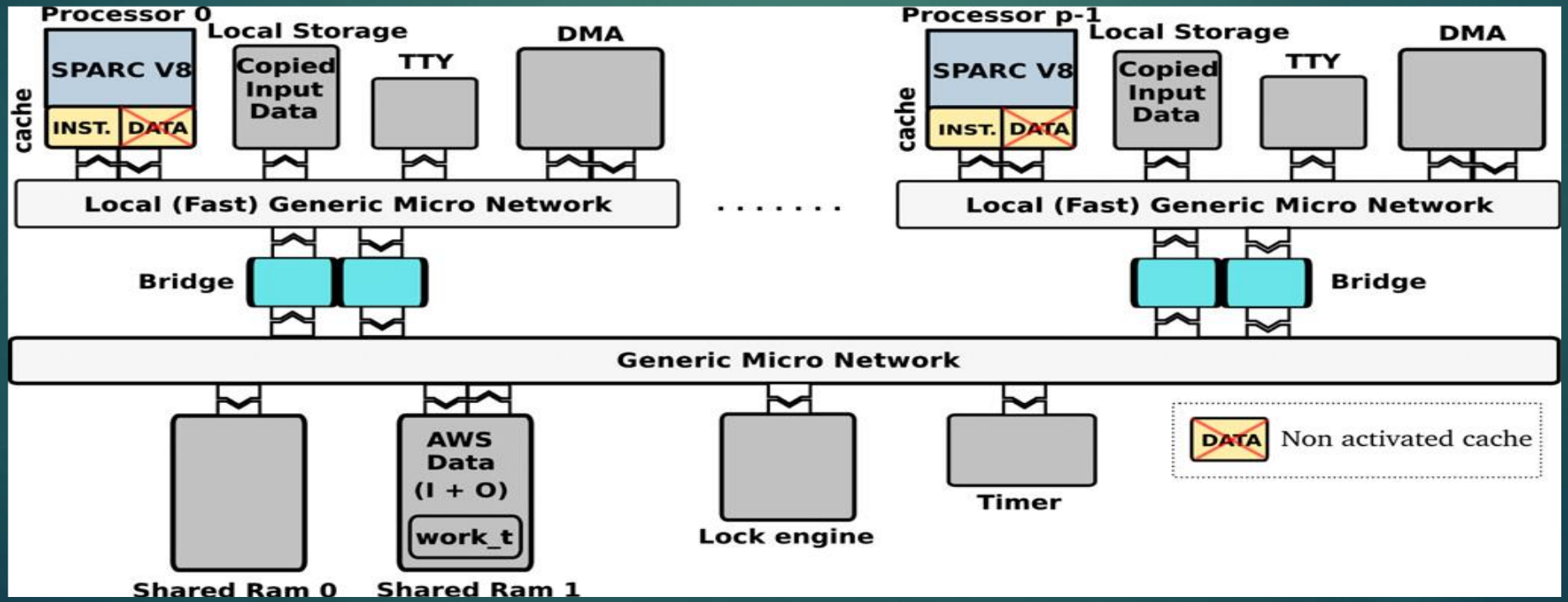


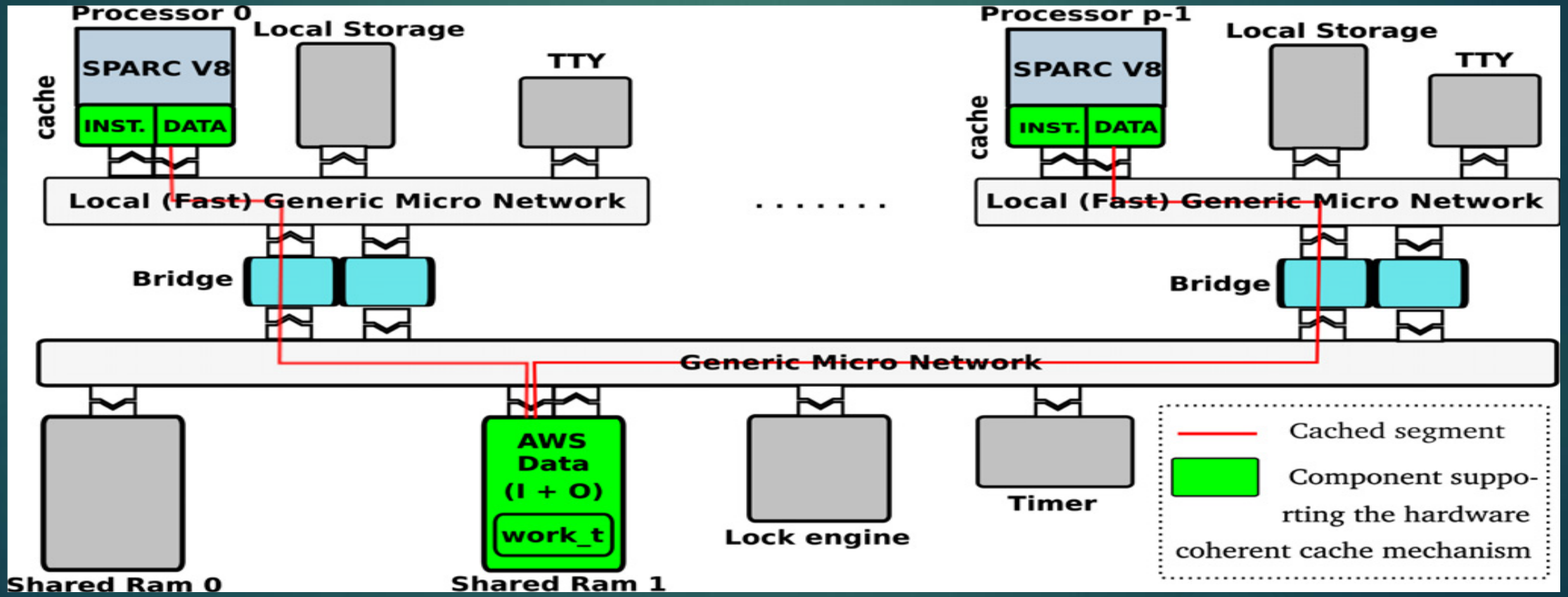
Fig. 4. Data contained in a node.

Hardware/software codesign for AWS implementation

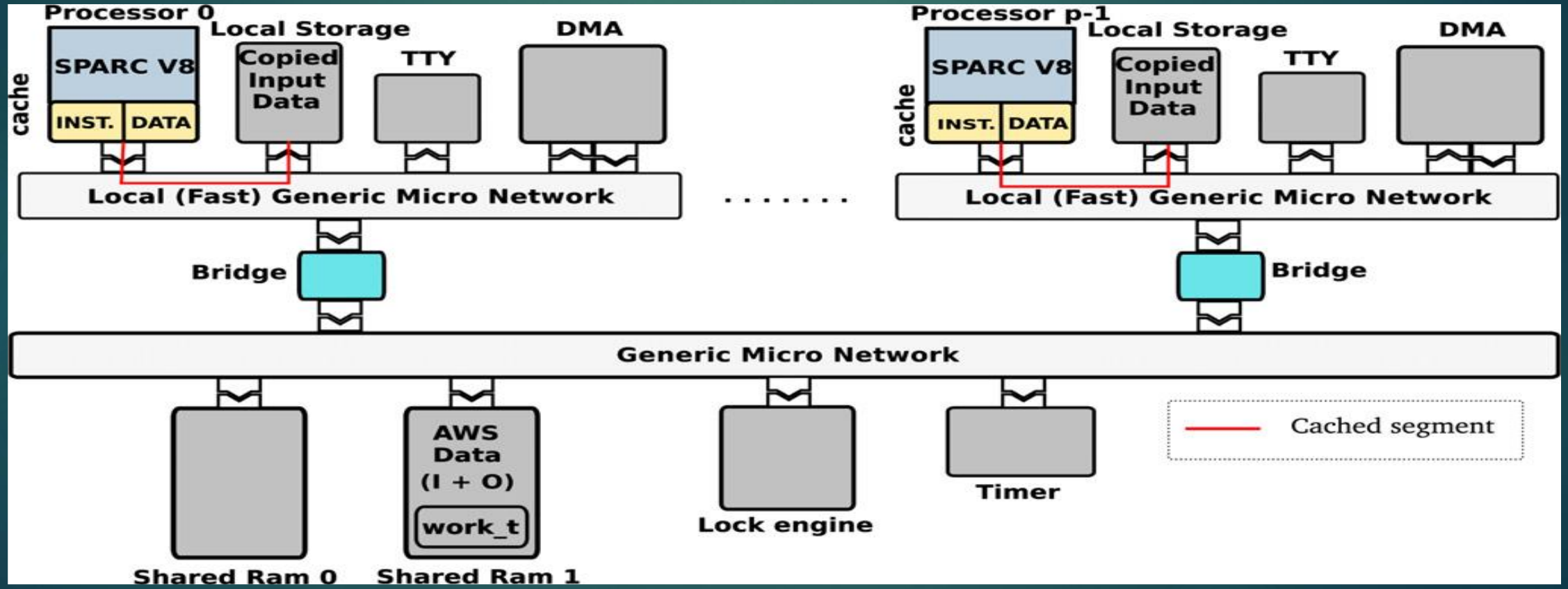
-Using DMAs



-Using Caches



-Using Caches and DMAs



-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

Results and analysis

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

-Comparing the execution times on different architectures for 100k

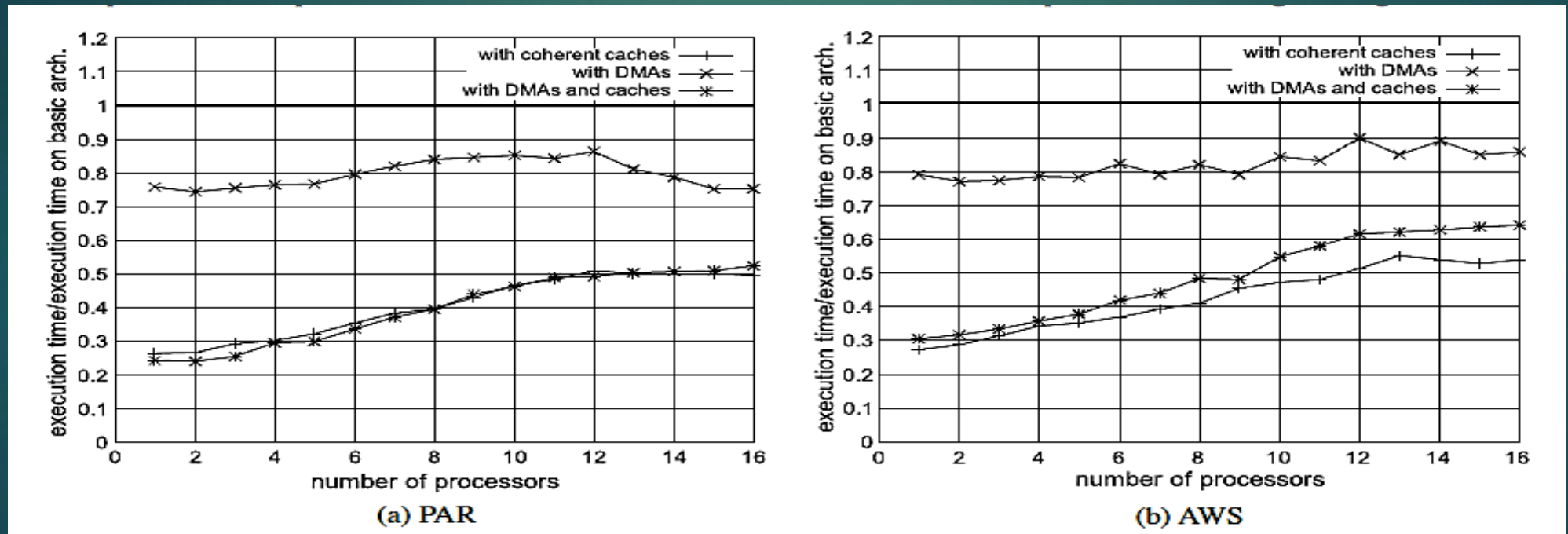
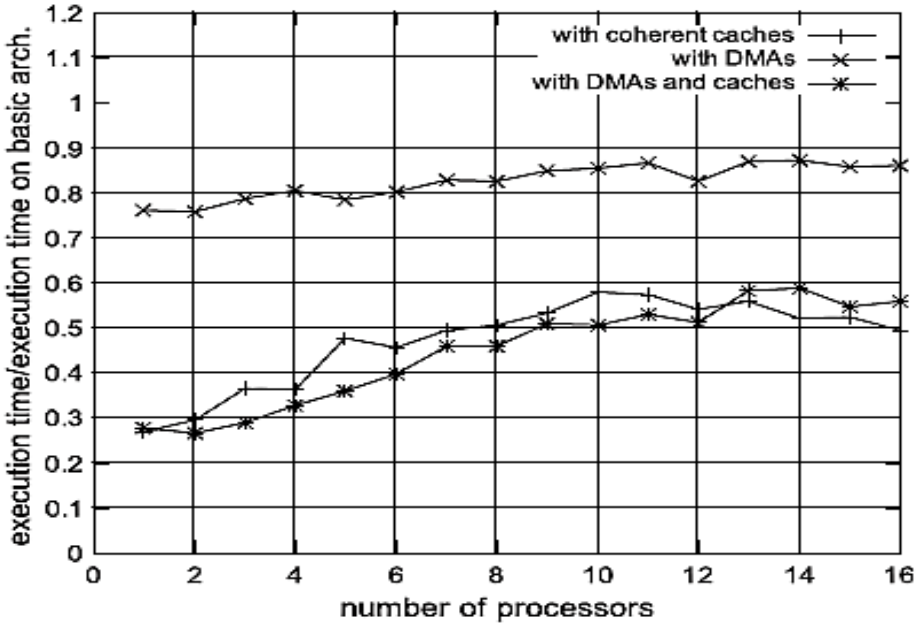


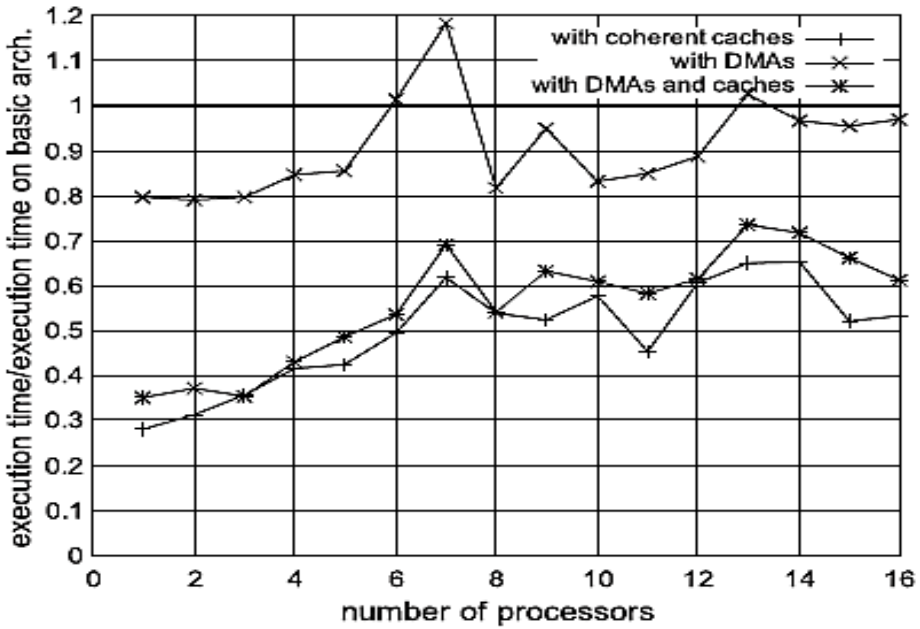
Fig. 8. Execution times on the different architectures for 1–16 processors, normalized w.r.t. the times on the basic architecture, with 100k elements

-Comparing the execution times on different architectures for 10k

Q. Meunier et al. / Journal of Systems Architecture 56 (2010) 392–406



(a) PAR



(b) AWS

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

-Comparing sequential and parallel execution times on a given architecture for 100k elements

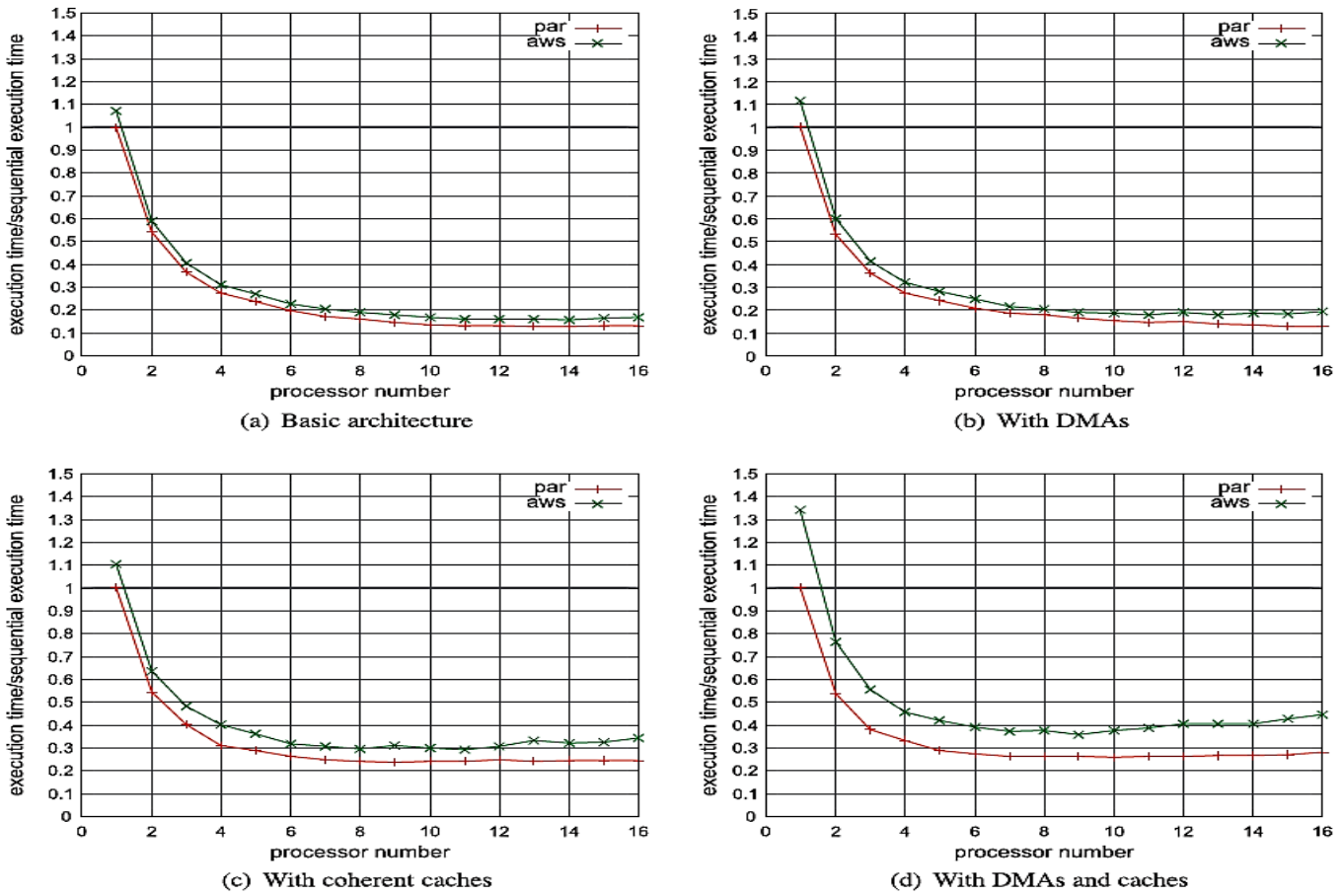


Fig. 10. Normalized execution times on the different architectures for 1–16 processors with 100k elements.

-Comparing sequential and parallel execution times on a given architecture for 10k elements

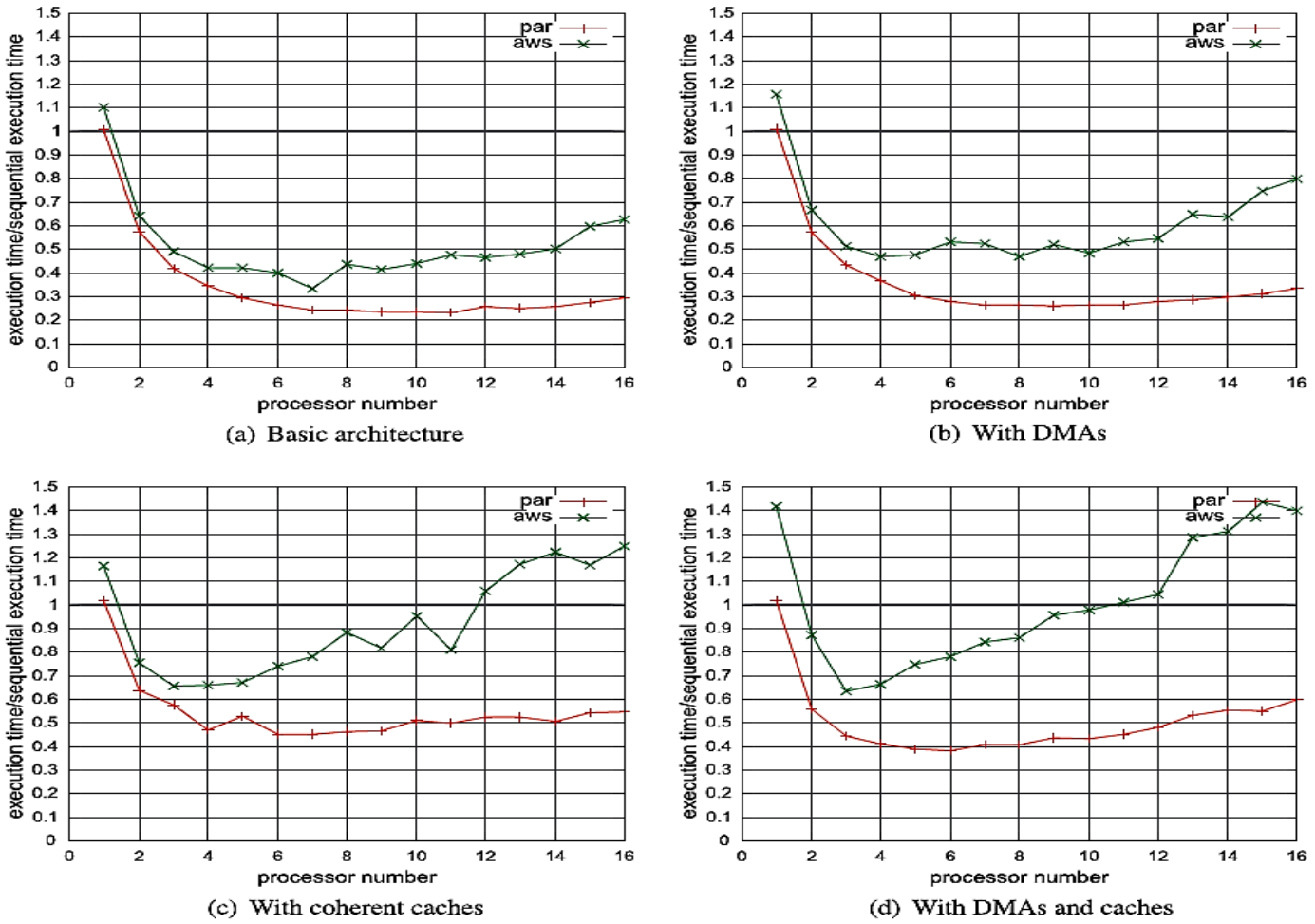


Fig. 11. Normalized execution times for different architectures for 1–16 processors with 10k elements.

-Distributed locks and work structures

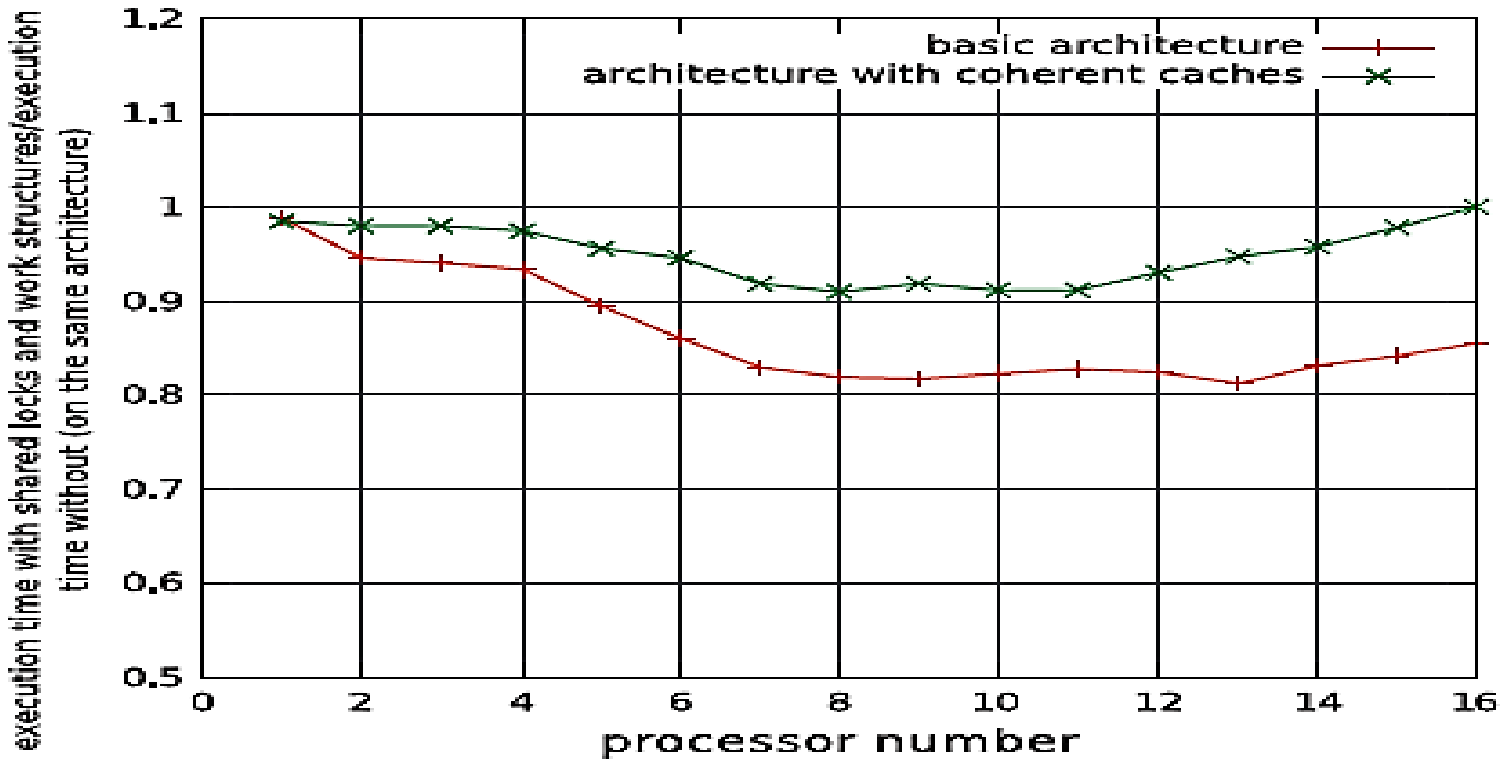


Fig. 12. Normalized execution times with distributed locks and work structures, on two architectures.

Performances on two real applications

1. A Luma and Chroma Temporal Noise Reduction (TNR) application
2. The computation and drawing of pictures from the Mandelbrot set

1. 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

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

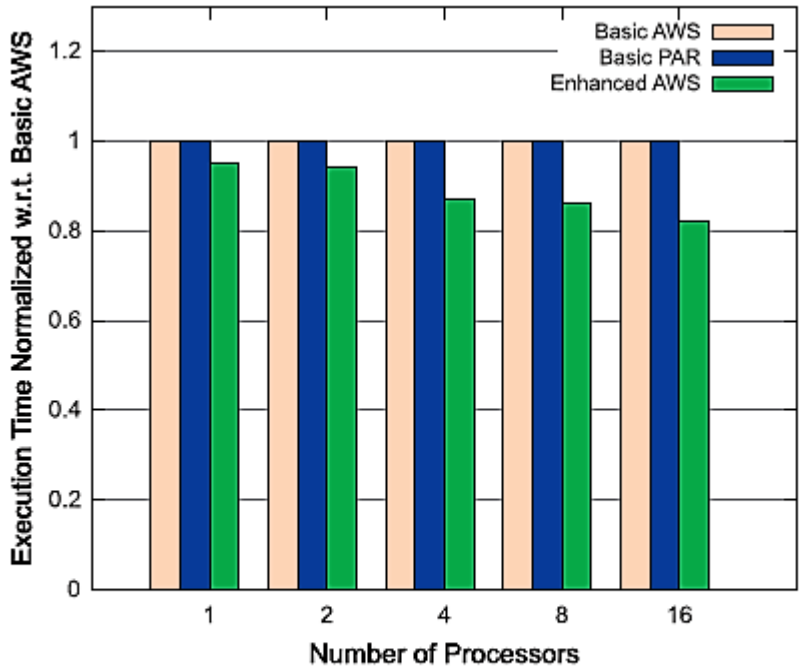


Fig. 13. TNR Execution Time for the decoding of four frames, normalized w.r.t. basic AWS.

2. 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.

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

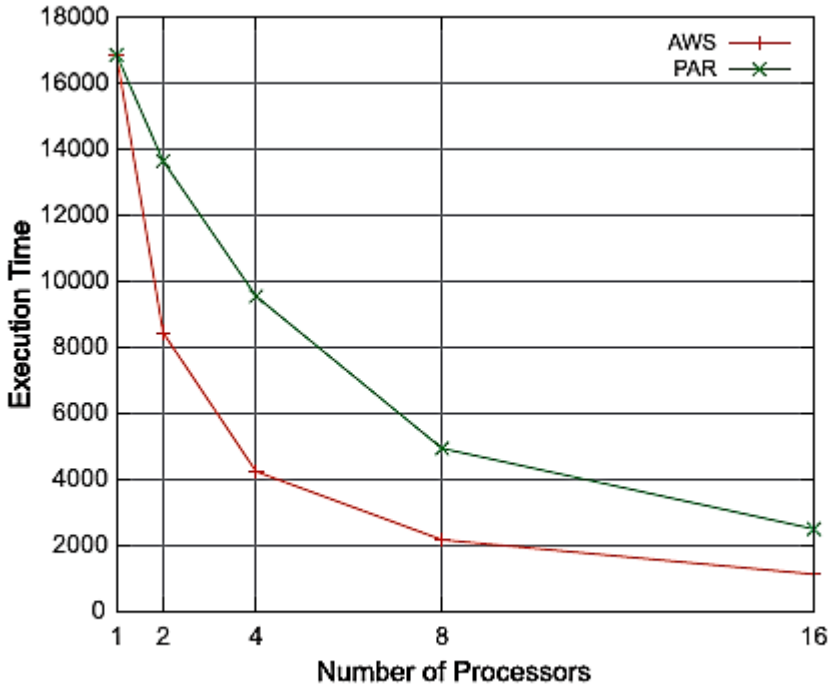


Fig. 16. Execution Time for the computation Mandelbrot frame #4.

ΕΥΧΑΡΙΣΤΩ ΓΙΑ ΤΟ ΧΡΟΝΟ ΣΑΣ