Comparison of different thread scheduling strategies for Asymmetric Chip MultiThreading architectures in embedded systems

Charly Bechara and Nicolas Ventroux

CEA, LIST,
Embedded Computing Laboratory,
Gif-sur-Yvette, F-91191, FRANCE;
Email: charly.bechara@cea.fr

Daniel Etiemble

Université Paris Sud,
Laboratoire de Recherche en Informatique,
Orsay, F-91405, FRANCE;

***Abstract*—Future embedded systems will have to support multiple and concurrent dynamic compute-intensive applications. These variable workloads can be handled by an efficient asymmetric MPSoC architecture, which integrates multithreaded processors as key processing elements. In this paper, we consider an asymmetric MPSoC architecture with a centralized controller and multiple multithreaded processors, which we call A-CMT (Asymmetric Chip MultiThreading). The centralized controller can implement 2 main types of thread scheduler architectures: VSMP (Virtual Symmetric MultiProcessing), and SMTC (Symmetric Multi-Thread-Context). Each type can have a static or dynamic allocation. We show that static scheduling for the A-CMT with dynamic applications (such as connected component labeling) can become a bottleneck for the overall architecture's performance, which leverages the use of dynamic scheduling for VSMP and SMTC. The dynamic SMTC gave a maximum of 51% and 11% speedup compared to the static SMTC and dynamic VSMP respectively.**

Keywords: Embedded systems, asymmetric MPSoC, multithreaded processors, VSMP, SMTC, thread scheduling, dynamic applications

I. Introduction

The emergence of new embedded applications for mobile, telecom, automotive, digital television and multimedia domains, has fueled the demand for architectures with higher performances, more chip area and power efficiency. These complex applications are usually characterized by their computation intensive workloads, their high-level of parallelism, and their dynamism.

These types of embedded applications necessitate architectures that support a multithreaded execution environment and exploit the parallelism at the thread level (TLP), where multiple software threads can be executed in parallel on multiple execution resources. Designers are thus showing interest in System-on-Chip (SoC) paradigms composed of multiple execution resources and networks that are highly efficient in terms of latency and bandwidth. The resulting new trend in architectural design is the MultiProcessor SoC (MPSoC) [6].

Most of the embedded systems applications are dynamic, which implies that the total execution time can highly vary with respect to the input data. Thus, optimal static tasks partitioning cannot exist on these MPSoC architectures, since the tasks processing times depend on the input data and cannot be known at compile-time. The only optimal solution consists in dynamically allocating tasks according to the availability of computing resources. This is mainly achieved by a centralized controller, which handles the scheduling and allocation of tasks on multiple cores, and maintains a balanced system load by supporting the preemption and migration of tasks between the processors. Such architectures are called asymmetric MPSoC. The centralized controller can be a dedicated HW IP core or a programmable processor reserved for management of tasks and resources, such as the SCMP architecture [15].

These MPSoC architectures consist of multiple cores that can each execute one software thread at a time. However, recent studies [8] show that the processor pipeline can be stalled up to 75% of its execution time because of long latency events such as cache misses, I/O response, and data synchronization between threads. A multithreaded processor [14] provides the hardware resources and mechanisms to concurrently execute several threads on one processor in order to increase its pipeline utilization, hence the application throughput. The threads compete for the shared resources, tolerate the long latency events and increase the efficiency of the architecture. Multithreading can be applied on single-issue scalar processor architecture such as interleaved multithreading (IMT) [11] and blocked multithreading (BMT) [9], and wide-issue superscalar processors such as simultaneous multithreading (SMT) [13]. For embedded systems, hardware architects are mainly interested in scalar multithreaded cores due to their small footprint and high energy efficiency. For instance, the MIPS34K [12] and Infineon TriCore 2 [1] are two examples of commercial multithreaded IP cores for embedded systems.

Efficient thread scheduling on multiple multithreaded processors is a critical part for the overall architecture performance and might lead to severe drawbacks. In this paper, we evaluate 4 types of thread scheduling techniques for asymmetric MPSoC architectures, consisting of one

centralized controller and multiple multithreaded cores. We refer to this type of architecture as A-CMT (Asymmetric Chip MultiThreading). To study the efficiency of the schedulers regarding dynamic applications, we implement the connected component labeling application that is highly used in the embedded vision domain. This application is particularly relevant to this study in terms of dynamism, parallelism and control dependencies.

The main contributions of this paper are:

- Implementation of 4 thread scheduling algorithms for an asymmetric MPSoC architecture with multiple multithreaded processors (A-CMT)
- Performance comparison of the 4 algorithms using a dynamic application from the embedded systems domain (real-case scenario)
- Recommendation on the best thread scheduling algorithm for A-CMT architectures executing dynamic applications

This paper is organized as follows: Section II presents the A-CMT architecture used in this study, and section III discusses the 4 types of thread schedulers for A-CMT evaluated in this work. Section IV describes the modeling and simulation frameworks, while a case study from the embedded vision domain that shows the advantages and limitations of each of the thread scheduler architectures is conducted in section V. And finally, section VI concludes this paper by discussing the present results along with future works.

II. A-CMT Structure

As mentioned earlier, we consider in our study an asymmetric MPSoC architecture, which consists of a centralized controller and multiple multithreaded processors connected to a shared memory. We call this architecture A-CMT, which stands for Asymmetric Chip MultiThreading, and it is described in Figure 1.

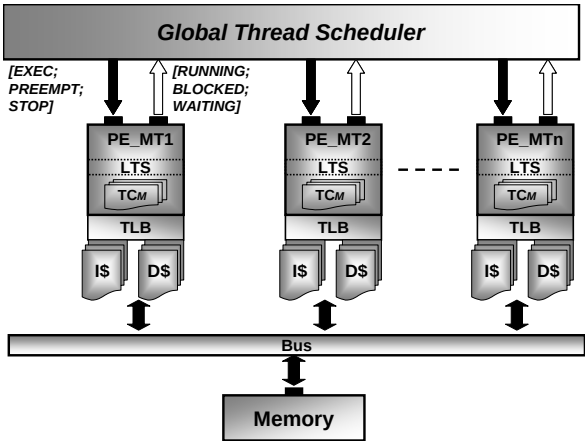


Figure 1. A-CMT architecture consisting of a centralized controller or global thread scheduler, and multiple multithreaded processors, all sharing a memory

This architecture is composed of 3 main parts:

1) *PE_MT system:* It consists of multiple multithreaded cores. Each core is a scalar in-order processor. It can process multiple Thread Contexts (TC) concurrently, where each TC is a virtual processor. In this work, we consider the case of 2 TCs per multithreaded core, which is suitable for embedded systems requirements. A Local Thread Scheduler (LTS) synchronizes the execution of the tasks on multiple TCs according to the PE_MT's multithreading policy. Since it is a scalar in-order processor, only one instruction is allowed to be processed from one task at a time. For instance, an IMT core processes the instructions in a round-robin manner between the available TCs, while a BMT core switches between the instructions of the available TCs whenever one is stalled on a long latency event, such as a cache miss. Each TC state is sent to the centralized controller. The TC state can be either *running* normally, *blocked* on cache miss or I/O, or *waiting* for an execution demand. Based on these values, the controller has a more global view on all the cores' status and can perform the right scheduling decision. Each PE_MT has a shared TLB for all the TCs for proper virtual to physical address translation, and it is connected to a L1 Instruction memory cache (I$) and Data memory cache (D$). In our architecture, the L1$ is segmented per TC in order to limit cache interferences.
2) *Centralized Controller:* It is a dedicated programmable processor that holds all the information of the application tasks in a special local memory. It runs a scheduling algorithm that schedules and allocates the tasks on the PE_MTs. The allocation decision is based on each TC status and the implemented thread scheduling strategy. It sends an *execution*, *preemption* or *stop* task demand on the corresponding TC. The reactivity of the controller is a key metric for low scheduling overheads. In section III, we will show the different scheduling strategies that are implemented in the controller and their overhead cost.
3) *Memory system:* It consists of a 2-level memory hierarchy: a private L1$ per PE_MT and a shared memory. They are connected by a bus that transfers the memory requests of all the PE_MTs to the shared memory. We assume that the shared memory has enough space to store all the application's tasks code and data prior to execution.

III. Thread scheduling strategies for A-CMT

The objective of a thread scheduler is to keep busy all the underlying execution resources and balance the load perfectly between them. It holds the information of all the SW threads that can be executed on the processors in a *runqueue*. For the case of a multicore system and a SMP OS such as Linux SMP, the scheduler creates a *runqueue* per core. Tasks are migrated periodically from one runqueue to another whenever

a workload imbalance occurs. This works fine with monothreaded cores. However, for multithreaded cores, it is not clear which scheduling technique fits better: whether to assign one runqueue per multithreaded core (VSMP) or one runqueue per thread context (SMTC), the objective is the same, keeping all the multithreaded cores active.[1] A multithreaded core is active if it has at least one active TC. For each scheduler architecture, we implement static and dynamic allocation.

A. VSMP

VSMP or Virtual SMP is an OS scheduler architecture that creates one *runqueue* per core (see Figure 2). If there is one TC per core (monothreaded processor), the scheduler converges to normal SMP. But in case of multiple TCs per core (multithreaded processor), only one *runqueue* is assigned to all the TCs. Then, it is up to the LTS to guarantee an efficient dispatching of the tasks to the free TCs.

The main advantage of VSMP is its rapid deployment. Only small modifications to the SMP OS need to be done. However, the scheduler does not have a global view of the workload balance between the TCs and the cores, which might be penalizing in some cases. Consider for example 2 PE_MTs with 4 TCs each, if PE_MT1 has 3 active TCs and PE_MT2 has 1 active TC, then the VSMP scheduler will treat both multithreaded processors equally, since both of them are active.

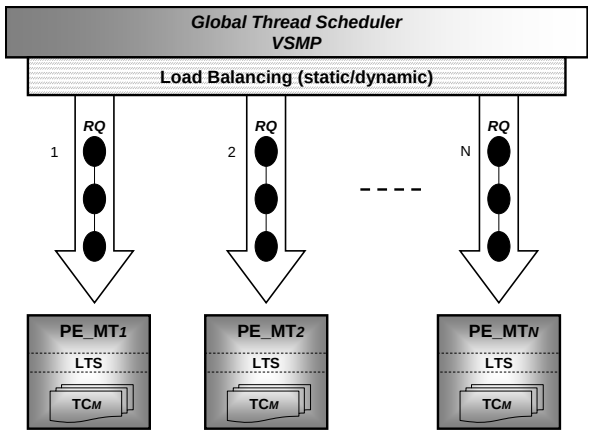


Figure 2. VSMP scheduler architecture

1) Static VSMP: For static VSMP, a task is allocated on a *runqueue* based on its identifier using the modulo operator. No task is allowed to migrate to other runqueues.

2) Dynamic VSMP: For dynamic VSMP, the scheduler scans the execution status of all the PE_MTs. If a multithreaded core is active and another one is free, it migrates a task from the active to the free *runqueue*. However, as stated earlier, the scheduling decision does not take into consideration the exact load of each PE_MT.

[1]We adopt the same terminologies used for Linux SMP ported to MIPS 34K [2]: VSMP (Virtual Symmetric MultiProcessing), and SMTC (Symmetric Multi-Thread-Context)

B. SMTC

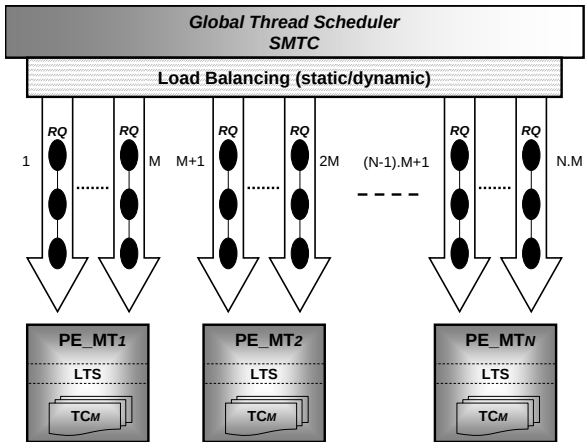


Figure 3. SMTC scheduler architecture

SMTC or Symmetric Multi-Thread-Context is an OS scheduler architecture that creates one *runqueue* per TC (see Figure 3). The scheduler has a more global and correct view of the real physical hardware. Depending on the TC state, the scheduler is able to know which PE_MT is active and by how much load, which facilitates the global workload balancing. This will relieve the LTS from doing local task allocation and concentrate only on its scheduling policy (interleaved, blocked, etc...). Since more execution states information are available, the scheduling time might take a little longer than in VSMP but this is not critical due to the possible gain we can have.

1) Static SMTC: For static SMTC, tasks are allocated on each TC runqueue based on its identifier using the modulo operator, and no load balancing is allowed. This implies that the LTS has no local scheduling role, since the tasks are already predefined where they will execute. This can be penalizing, since all the TCs are treated equally as a virtual processor which might lead to severe load imbalance (see Section V-B).

2) Dynamic SMTC: For the dynamic SMTC scheduler, the native SMP scheduler code needs to be modified and rethought. At the beginning of a scheduling cycle, the controller receives the execution state of all the TCs. Then, it executes the scheduling algorithm that is decomposed into 3 main parts: sorting, allocation, and verification.

The first phase creates a sorting list of the tasks that are ready to be allocated and executed. The sorting decision depends on the task priority and execution state. For example, a blocked task is put at the end of the sorting list. Then, the first NB_PE tasks are chosen to be allocated, where NB_PE is the maximum number of TCs available in the architecture. For instance, 4 PE_MTs with 2 TCs each has NB_PE equal to 8. The second phase allocates the tasks on the *runqueue* of each

TC. Here, the scheduling algorithm has 2 different views of the A-CMT architecture: virtualized mode and non-virtualized mode. In the *virtualized mode*, the execution state of all the TCs of one PE_MT are grouped together in order to form a common architectural state of the PE_MT. A PE_MT is active if at least one TC is active, and an A-CMT architecture is executing efficiently if all the PE_MTs are active. Accordingly, ready tasks are allocated on the corresponding TCs runqueue that turns a PE_MT into active. If all the PE_MTs has at least one active TC and there are still ready tasks in the sorting list, then the scheduling algorithm switches to the *non-virtualized mode*. In this case, a ready task is allocated on a *runqueue* of a free TC.

The final phase verifies if the multithreaded processors are well-balanced. For example, consider a system of 2 PE_MTs with 4 TCs each. If PE_MT1 has 3 active TCs and PE_MT2 has 1 active TC, then the dynamic SMTC scheduler will allow the migration of tasks from runqueue TC2 of PE_MT1 to runqueue TC1 of PE_MT2. This scenario is not possible for the VSMP scheduler.

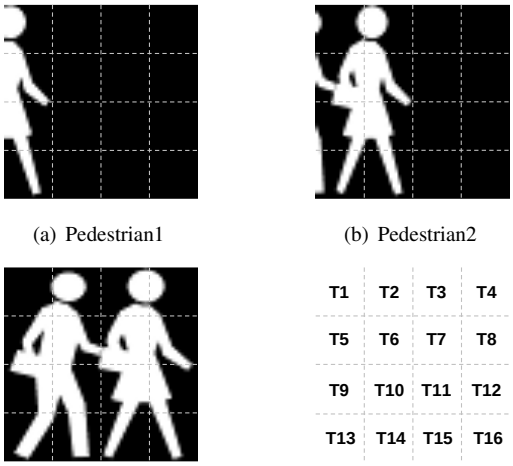
IV. Experimental setup

The A-CMT architecture is implemented in the SESAM framework [16], [17], which is a SystemC framework for modeling and exploration of asymmetric MPSoC architectures.

Initially, SESAM infrastructure consists of several SystemC instruction-set simulators (ISS) for the MIPS1 R3000, MIPS32, SPARC and other monothreaded processor architectures. Recently, we supported SESAM with a cycle-accurate multithreaded ISS based on MIPS1 ISA [4]. It implements the blocked multithreading protocol and it has 2 thread contexts. In fact, as implementing more than 2 TCs for a small footprint RISC processor would almost double the processor area, it would be better to duplicate the number of multithreaded processors instead of the number of TCs per multithreaded processor. For instance, the MIPS 1004K [3] and TriCore 2 [1] implements 2 TCs per PE_MT.

The centralized controller sees each ISS as one execution resource. Thus, we added virtualization support to the SESAM framework, so that the centralized controller is aware of the TCs in each multithreaded ISS. In SESAM, a parallelized application is cut manually into several SW tasks. Their control and data dependencies are represented in a Control Data Flow Graph (CDFG), which is described manually with a simple dedicated assembly language. Each transition represents an execution constraints that imposes the task execution order. Then, a parser generates the binary from the CDFG file, which is fed to the centralized controller. Each SW task is a standalone program and the binaries are generated by the MIPS gcc 4.2.3 cross-compiler.

As for the memory system, we consider that a L1$ hit takes 1 cycle and a memory access time takes a minimum of 6 cycles in case of no bus contentions. Otherwise, the access time increases depending on the number of cores accessing the shared memory simultaneously.



(a) Pedestrian1 (b) Pedestrian2

(c) Pedestrian3 (d) Sub-image to task number

Figure 4. Figure a) b) c) 3 images of 2 pedestrians crossing the road. Figure d) shows the sub-images decomposition and their corresponding task number

V. Case Study

To measure the A-CMT performance for dynamic embedded applications, we consider an application called ADAS (Advanced Driver Assistance Systems). It consists of a camera installed in the car that detects humans on the roads, in order to detect a pre-crash situation. This is a critical application for automotive systems and is particularly relevant to this study in terms of dynamism, parallelism and control dependencies.

In ADAS, one part of the obstacle detection process is the connected component labeling algorithm. The labeling algorithm transforms a binary image into a symbolic image, so that each connected component is uniquely labeled based on a given heuristic. It detects unconnected regions in binary images. Various algorithms have been proposed [7] [10], but we have chosen an algorithm using contour tracing technique [5]. The initial algorithm is parallelized by creating independent tasks with control dependencies explicitly represented in a CDFG. To get multiple independent tasks, we cut the image into sub-images and applied the algorithm on each sub-image. Then, we carried out successively a vertical and a horizontal fusion of labels in analyzing frontiers between sub-images, and finally we constructed the corresponding tables between labels and changed in parallel all labels into sub-images. As input images, we used a 128x128 pixel image, cut into 16 8x8 sub-images. This implies that the maximum parallelism is *16*. The input images are a sequence of 3 images taken at different time intervals. They show 2 pedestrians crossing the road (Figure 4(a),4(b),4(c)), and they are close to the car (about 10 meters). The labeling algorithm is implemented on each image.

For all the experiments, the number of PE_MTs varies between 1 and 8, where each PE_MT has 2 TCs. The L1 I$ and D$ size is fixed to 2KB, which gives a cache miss rate around 10% for the connected component labeling application. In fact,

since the blocked multithreading policy is implemented, all the TCs will execute only if there are enough pipeline stalls (i.e. cache misses).

A. Dynamism of the application

The computation requirement differs for the 3 images as shown in Figure 5. Pedestrian3 image takes about **3 times** more processing than pedestrian1 image. Pedestrian1 image has 25% of its sub-images executing the labeling code, since the others are black sub-images (non-balanced workload). Similarly, pedestrian2 image has 50% (semi-balanced workload) and pedestrian3 has 100% (fully-balanced workload). This behavior reveals the dynamism of the connected component labeling application with respect to the input data.

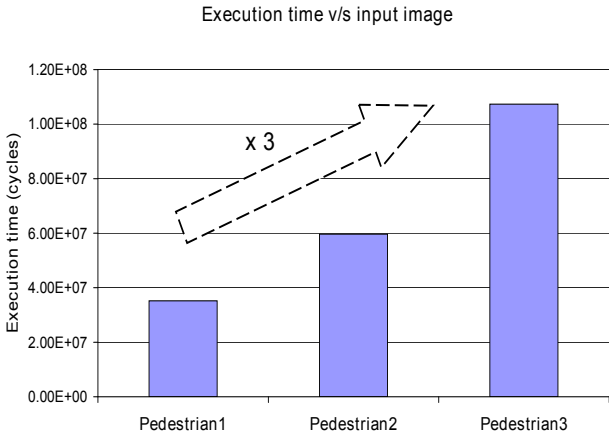


Figure 5. Dynamic behavior of labeling application when executing the 3 pedestrian images

B. Static v/s dynamic thread scheduling

In this experiment, we compare the static and dynamic algorithms of the VSMP and SMTC thread scheduling architectures, shown in Figure 6(a) and 6(b) respectively. In these figures, we plot the number of execution cycles of the static and dynamic algorithms for the 3 pedestrian images. The input image is cut into 16 sub-images, and the labeling tasks of each sub-image are allocated first horizontally then vertically. The sub-images identifiers are set from 1 to 16 respectively as shown in Figure 4(d). For example, in the pedestrian1 image (Figure 4(a)), the tasks [T1,T5,T9,T13] contain pixels that need to be processed by the labeling algorithm, which implies more processing times.

The first observation is related to the type of the input image. We can clearly notice that for both VSMP and SMTC, and for a balanced workload (pedestrian3 image), the difference between static and dynamic is not significant (less than 10%). However, when the workload is more unbalanced (pedestrian1 and pedestrian2 images), the dynamic algorithm reaches a maximum speedup of 40% and 51% compared to the static algorithm, for VSMP and SMTC respectively. In real-case scenarios, we expect on average a semi-balanced workload similar to pedestrian2 image.

For the VSMP scheduler, there is one runqueue per PE_MT. This implies that the execution time for static and dynamic VSMP should be similar for 1 PE_MT and independent of the input image, which is confirmed in Figure 6(a). As for 8 PE_MTs, the performance is also similar for static and dynamic VSMP. In fact, VSMP allocates 2 tasks on each runqueue in the same way for static and dynamic algorithm. The LTS of each PE_MT is responsible of dispatching these 2 tasks on one of its 2 TCs. If one TC finishes execution before the other, the PE_MT is still considered as *active*, and the dynamic VSMP cannot balance the load between the runqueues since it does not see the actual workload per PE_MT. However, for 2 and 4 PE_MTs, the dynamic VSMP outperforms the static VSMP. The maximum speedup reaches 40% for the pedestrian1 image, and goes down to 36% and 7% for pedestrian2 and pedestrian3 images respectively. For instance, let's take the configuration of 4 PE_MTs and pedestrian1 image. If PE_MT1{TC1}, PE_MT2{TC2}, PE_MT3{TC3}, and PE_MT4{TC4}; then TC1=[T1,T5,T9,T13]; TC2=[T2,T6,T10,T14]; TC3=[T3,T7,T11,T15]; TC4=[T4,T8,T12,T16]. Thus, it can be clearly seen that all the heavy computation tasks are assigned to TC1 runqueue for the static VSMP scheduler. On the other hand, the dynamic VSMP is able to move those tasks to other free runqueues and balance the load effectively.

For the SMTC scheduler, there is one runqueue per TC (in our case 2 runqueues per PE_MT). This explains the speedup for 1 and 8 PE_MTs configurations as shown in Figure 6(b). In fact, the dynamic SMTC scheduler is able to see the exact occupation rate of each TC and balance the workload between the runqueues to exploit the multithreaded processor performance. For example, let's consider the case of 1 PE_MT{TC1,TC2} with pedestrian1 image: all the heavy computation tasks are allocated on TC1 runqueue in the static SMTC version. This means that TC2 runqueue will be processed much faster than TC1, and the remaining tasks on TC1 will not be migrated in the static version, which is not the case for dynamic SMTC. As for 2 and 4 PE_MTs, the dynamic SMTC reaches a maximum speedup of 51% for the pedestrian1 image compared to the static SMTC. To understand better the reason for this large performance difference, let's consider the case for 2 PE_MTs with pedestrian1 image. If PE_MT1{TC1,TC3} and PE_MT2{TC2,TC4}; then TC1=[T1,T5,T9,T13]; TC2=[T2,T6,T10,T14]; TC3=[T3,T7,T11,T15]; TC4=[T4,T8,T12,T16]. Again, in static SMTC, the heavy computation tasks are assigned to TC1 runqueue, which dispatches the tasks to only one thread context of PE_MT1. On the other hand, the dynamic SMTC scheduler is able to perform better load balancing and gives a speedup of 51% for this configuration.

C. VSMP v/s SMTC

In Figure 6(c), we compare the dynamic algorithm of VSMP and SMTC for the 3 pedestrian images. In all the configurations, the dynamic SMTC has a better performance than

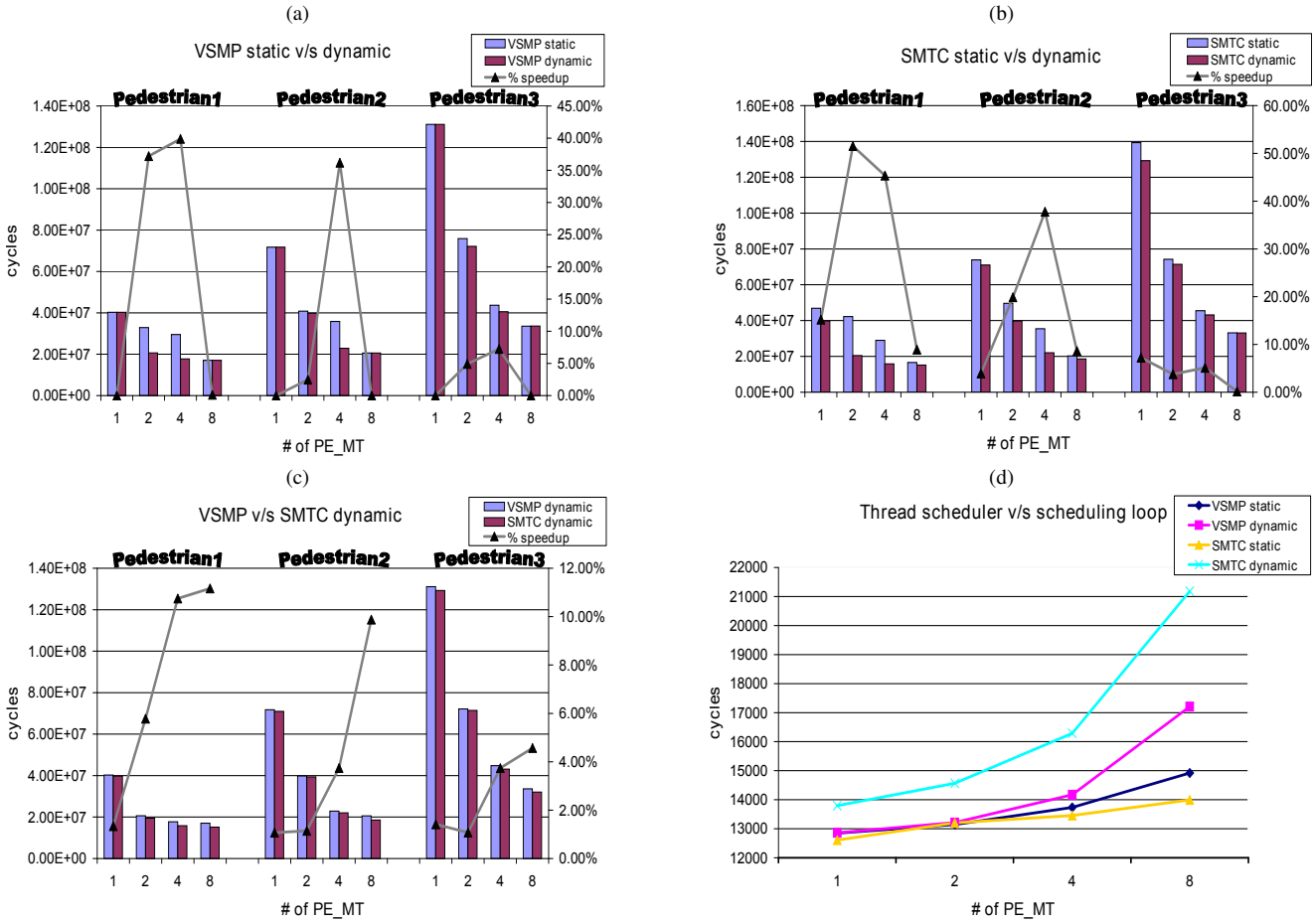


Figure 6. Performance comparison of the different thread scheduling strategies: a) VSMP static v/s dynamic. b) SMTC static v/s dynamic. c) VSMP dynamic v/s SMTC dynamic. d) Thread scheduler v/s number of clock cycles per scheduling loop. The speedup line in the first 3 figures is the speedup with respect to the same configuration.

dynamic VSMP. The speedup varies between 1% and 11%. Again, the speedup is more important for non-balanced and semi-balanced workloads, since the runqueues contain tasks with different computation requirements. This necessitates a more optimal load balancing from the global thread scheduler, which favors the dynamic SMTC on dynamic VSMP. In addition, the speedup is higher for the cases with large number of multithreaded processors (PE_MT = 8). In fact, when the number of multithreaded processors increases, the complexity of finding the optimal scheduling decision also increases. This is due to the fact that the scheduling decision for multiple multithreaded processors is different and more complex than monothreaded processors. It requires an effective and reactive global thread scheduler for proper load balancing between the runqueues. Hence, dynamic SMTC gives superior performance on dynamic VSMP, and this difference would be more important if the number of TCs per PE_MT is bigger than 2.

D. Scheduling overhead

Finally, in Figure 6(d), we compare the complexity of the 4 types of thread schedulers. The results show the average number of cycles taken to complete a scheduling loop. As expected, the static versions take less time to finish a scheduling loop compared to their corresponding dynamic versions. One clear observation is that the time to complete a scheduling loop for the SMTC dynamic is much longer than VSMP dynamic, especially when the number of multithreaded processors increases (around 4000 clock cycles difference when considering 8 PE_MTs). The difference is expected to increase more when the number of TC per PE_MT is bigger. This result is not surprising, since the dynamic SMTC algorithm has one runqueue per TC, thus performing more tests in order to choose the best TC's runqueue to allocate the SW task (see Section III-B2). In fact, the SMTC has a complexity of O(NxM), while VSMP is O(N), where N is the number of multithreaded processors and M is the number of TCs per PE_MT. But, as we saw previously from the results, the scheduling overhead does not impact the performance, since in an asymmetric CMT architecture, the global thread scheduler executes in parallel to the computation. On the other hand, in a symmetric approach, the scheduler executes on the same processor as the computation and hence needs to finish the scheduling loop as fast as possible. This implies that the centralized scheduler in an asymmetric CMT architecture

can implement more complex scheduling algorithms without performance drop. For instance, the scheduling overhead with respect to the effective execution time of the processors reaches a maximum of 10% for 8 PE_MTs and for all the scheduling strategies.

VI. Conclusion

This paper has presented a comparison between 4 different thread scheduler architectures for asymmetric MPSoC architectures, consisting of one centralized controller and multiple multithreaded processors (2 Thread Contexts) with local segmented L1$ memories and a shared memory. The studied MPSoC architecture is called A-CMT, which stands for Asymmetric Chip MultiThreading, and it is used for embedded dynamic applications.

We studied the VSMP and SMTC schedulers suitable for multiple multithreaded processors. Both schedulers were implemented with a static and dynamic allocation of the tasks on the runqueues. The dynamic scheduling algorithm has proven its efficiency and superiority in performance compared to the static algorithm, for applications with high degree of parallelism and dynamism, such as the connected component labeling. A real-case scenario that applies the labeling algorithm on 3 images of 2 pedestrians crossing the road was conducted. The input images have different workloads. For instance, the dynamic VSMP and SMTC gave a maximum speedup of 40% and 51% compared to their corresponding static versions. Finally, the dynamic SMTC has proven to be the best thread scheduler for A-CMT architecture with a maximum speedup of 11% compared to dynamic VSMP.

For future enhancements, we aim to extend the number of thread contexts per multithreaded processor and explore the efficiency of the A-CMT architecture for high-end server applications.

References

[1] Infineon tricore2. http://www.infineon.com/.

[2] Linux mips 34k: http://www.linux-mips.org/wiki/34k.

[3] Mips 1004k. http://www.mips.com/products/cores/32-64-bit-cores/mips32-1004k/.

[4] C. Bechara, N. Ventroux, and D. Etiemble. A tlm-based multithreaded instruction set simulator for mpsoc simulator environment. In *3rd Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO 2011), Held in conjunction with the 6th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, January 2011.

[5] C. C. F. Chang and C. Lu. A Linear-Time Component-Labeling Algorithm Using Contour Tracing Technique. *Computer Vision and Image Understanding*, 93(2):206–220, 2004.

[6] A. A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chips*. 978-0-12-385251-9. Elsevier, 2005.

[7] I. H. K. Suzuki and N. Sugie. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, 89(1):1–23, 2003.

[8] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[9] J. Kreuzinger and T. Ungerer. Context-switching techniques for decoupled multithreaded processors. volume 1, pages 248 –251 vol.1, 1999.

[10] L. Lacassagne and B. Zavidovique. Light speed labeling: efficient connected component labeling on risc architectures. *Journal of Real-Time Image Processing*, pages 1–19, 2009. 10.1007/s11554-009-0134-0.

[11] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: a multithreading technique targeting multiprocessors and workstations. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 308–318, New York, NY, USA, 1994. ACM.

[12] MIPS. Programming the MIPS32® 34K Core Family. Technical report, MIPS Technology, 2005.

[13] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 533–544, New York, NY, USA, 1998. ACM.

[14] T. Ungerer, B. Robic, and J. Silc. Multithreaded processors. *The Computer Journal*, 45:320–348, 2002.

[15] N. Ventroux and R. David. The scmp architecture: A heterogeneous multiprocessor system-on-chip for embedded applications. *Eurasip*, 2009.

[16] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, G. Blanc, C. Bechara, and R. David. Sesam: An mpsoc simulation environment for dynamic application processing. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1880 –1886, July 2010.

[17] N. Ventroux, T. Sassolas, R. David, G. Blanc, A. Guerre, and C. Bechara. Sesam extension for fast mpsoc architectural exploration and dynamic streaming applications. In *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, pages 341 –346, sept. 2010.