AHDAM: an Asymmetric Homogeneous with Dynamic Allocator Manycore chip

Charly Bechara[1], Nicolas Ventroux[1], and Daniel Etiemble[2]

[1] CEA, LIST, Embedded Computing Laboratory, Gif-sur-Yvette, F-91191, FRANCE; charly.bechara@cea.fr
[2] Université Paris Sud, Laboratoire de Recherche en Informatique, Orsay, F-91405, FRANCE;

Abstract. The future high-end embedded systems applications are characterized by their computation-intensive workloads, their high-level of parallelism, their large data-set requirements, and their dynamism. Those applications require highly-efficient manycore architectures. In response to this problem, we designed an asymmetric homogeneous with dynamic allocator manycore architecture, called AHDAM chip. AHDAM chip exploits the parallelism on all its granularity levels. It implements multithreading techniques to increase the processors' utilization. We designed an easy programming model and reused an automatic compilation and application parallelization tool. To study its performance, we used the radio spectrum sensing application from the telecommunication domain. On a simulation framework, we evaluated sequential and parallel versions of the application on 2 platforms: single processor, and AHDAM chip with a variable number of processors. The results show that the application on the AHDAM chip has an execution time 574 times faster than on the single-processor system, while meeting the real-time deadline and occupying 51.92 mm^2 at 40 nm technology.
Keywords: Manycore, asymmetric, multithreaded processors, dynamic applications, embedded systems

1 Introduction

During the last decades, the computing systems were designed according to the CMOS technology push resulting from Moore's Law, as well as the application pull from ever more demanding applications [1]. The emergence of new embedded applications for mobile, telecom, automotive, digital television, communication, medical and multimedia domains has fuelled the demand for architectures with higher performances (order of TOPS), more chip area and power efficiency. These complex applications are usually characterized by their computation-intensive workloads, their high-level of parallelism, their large data-set requirements and their dynamism. The latter implies that the total application execution time can highly vary with respect to the input data, irregular control flow, and auto-adaptivity. Typical examples of dynamic algorithms are 3D rendering, high definition (HD) H.264 video decoder, and connected component labeling [2].

The parallelism can be exploited on multiple granularities, such as instruction level (ILP), loop level (LLP), and thread level (TLP). Those massively parallel high-end embedded applications, which can have more than 1000 parallel threads, require highly efficient microprocessor architectures. With the limits of ILP [3] and the low transistor/energy efficiency of superscalar processors for embedded systems applications, the chip manufacturers are increasing the overall processing power by integrating additional CPUs or "cores" to the microprocessor package. Such microprocessor chip architectures for the embedded systems world are called *MPSoC*. An MPSoC with large number of cores is called a *manycore* architecture. These architectures need to exploit all types of parallelism in a given application.

Unfortunately, the existing MPSoC/manycore architectures offer only partial solutions to the power, chip area, performance, reliability and dynamism problems associated with the embedded systems. For instance, an optimal static partitioning on an MPSoC cannot exist since all the tasks processing times depend on the input data that cannot be known off-line. [4] and [5] show that the solution consists in dynamically allocating tasks according to the availability of computing resources. Global scheduling maintains the system load-balanced and supports workload variations that cannot be known off-line. Only an asymmetrical approach can implement a global scheduling and efficiently manage dynamic applications. An asymmetric MPSoC architecture consists of one (sometimes several) centralized or hierarchized control core, and several homogeneous or heterogeneous cores for computing tasks. The control core handles the tasks scheduling. In addition, it performs load balancing through task migrations between the computing cores when they are homogeneous. The asymmetric architectures have usually an optimized architecture for control. This distinction between control and computing cores renders the asymmetric architecture more transistor/energy efficient than the symmetric architectures. However, one main drawback of asymmetric architectures is their scalability. The centralized core is not able to handle more than a specific threshold number of computing cores due to reactivity reasons.

In this paper, we present the AHDAM chip, an asymmetric manycore architecture that tackles the challenges of future massively-parallel dynamic embedded applications. Its architecture permits to process applications with large data sets by efficiently hiding the processors' stall time using multithreaded processors. Besides, the AHDAM chip has an easy programming model as it will be shown in this paper. The main contributions of this paper are:

- System design (architecture + programming model) of an efficient asymmetric manycore chip architecture for the embedded systems.
- Architecture evaluation using a significant application from radio telecommunication domain (radio sensing).

This paper is organized as follows: Section 2 introduces the AHDAM chip architecture, its system environment, its programming model, its different components functionality, their interoperability, and the execution model. The evaluation of the architecture using an embedded application from the radio telecom-

munication domain, and its performance speedup compared to a monothreaded core is done in section 3. And finally, section 4 concludes the paper by discussing the present results along with future works.

2 AHDAM chip

AHDAM chip stands for Asymmetric Homogeneous with Dynamic Allocator Manycore chip. It is used as an on-chip accelerator component for high-end massively parallel dynamic embedded applications. Depending on the computation requirements, it can be used as a shared accelerator for multiple host CPUs, or a private accelerator for each host CPU. The host CPU is running an operating system or a bare-metal application. When a host CPU encounters a massively-parallel application, it sends an execution demand to AHDAM chip and wait for its acknowledgment. Then, the host CPU offloads the massively-parallel application to AHDAM chip. The application is already decomposed into concurrent tasks. AHDAM chip exploits the parallelism at the thread level (TLP) and loop level (LLP).

In this section, we illustrate AHDAM chip's programming model that supports the control-flow and streaming execution models (section 2.1). Then, we describe the overall architecture as well as the functionalities and interoperabilites between the hardware components in section 2.2. A typical execution model will be presented.

2.1 Programming model

The programming model for AHDAM chip architecture is specifically adapted to dynamic applications and global scheduling methods. It is based on a streaming programming model. The chip's asymmetry is tackled on 2 levels: a fine-grain level and a coarse-grain level. The proposed programming model is based on the explicit separation of the control and the computing parts. As depicted in Figure 1, each sequential application is parallelized semi-automatically using the *PAR4ALL* tool from HPC Project [6]. It is manually cut into different tasks through pragmas from which explicit execution dependencies are extracted (TLP). Then, the generated parallel application follows a second path in *PAR4ALL*, where OpenMP pragmas are inserted at the beginning of possibly parallelized 'for-loop' blocks (fine-grain). In fact, OpenMP [7] is a method of loop parallelization (LLP) whereby the master thread forks a specified number of slave threads, and a task is divided among them. Then, the child threads run in parallel, with the runtime environment allocating threads to different cores. The *PAR4ALL* tool supports AHDAM chip Hardware Abstraction Layer (HAL) for proper tasks generation. *PAR4ALL* generates as output the computing tasks and the control task that are extracted from the application, so as each task is a standalone program. The greater the number of independent and parallel tasks that are extracted, the more the application can be accelerated at runtime, and the application pipeline balanced.

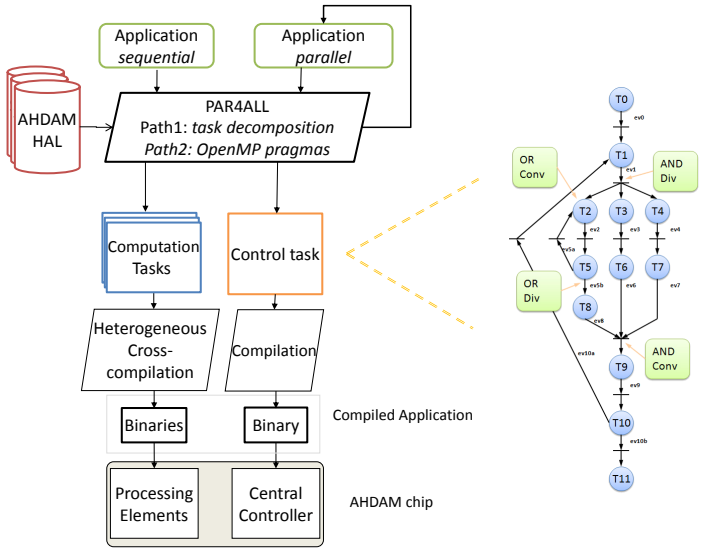

Fig. 1. AHDAM programming model and an example of a typical CDFG control graph

The control task is a Control Data Flow Graph (CDFG) extracted from the application (Petri Net representation), which represents all control dependencies between the computing tasks (coarse-grain). The control task handles the computing task scheduling and activations. A specific compilation tool is used for the binary generation from the CDFG generated by the *PAR4ALL* tool.

For the computing tasks, a specific HAL is provided to manage all memory accesses and local synchronizations, as well as dynamic memory allocation and management capabilities. A special on-chip unit called MCMU (Memory Configuration and Management Unit) is responsible for handling these functionalities (more details in section 2.2). With these functions, it is possible to carry out local control synchronizations or to let the control manager taking all control decisions. Concurrent tasks can share data through local synchronizations handled by the MCMU (streaming execution model). Each task is defined by a task identifier, which is used to communicate between the control and the computing parts. A task suspends/resumes its execution based on data availability from other tasks. It follows the producer/consumer execution model (streaming). When a data is produced by Task A, then Task B resumes its execution. When data is consumed by Task B, then it suspends its execution. Each task has the possibility to dynamically allocate or deallocate buffers (or double buffers) in the shared memory space through specific HAL functions. An allocated buffer is released when a task asks for it and is the last consumer. A buffer cannot be released at the end of the execution of the owner task. A dynamic right management of buffers enables a dataflow execution between the tasks: it is handled by the MCMU.

Once each application and thread has been divided into independent tasks, the code is cross-compiled for each task. For heterogeneous computing resources, the generated code depends on the type of the execution core.

2.2 Architecture description

The AHDAM chip architecture separates control from computing tasks. This separation rends the architecture more transistor/energy efficient, since the tasks are executed on dedicated resources. AHDAM chip is composed of 3 main units: Memory units, control unit, and computation units, as shown in Figure 2. AHDAM chip has M Tiles, where a Tile represents a computation unit. In the following sections, we will describe in more details the functionality of each unit.

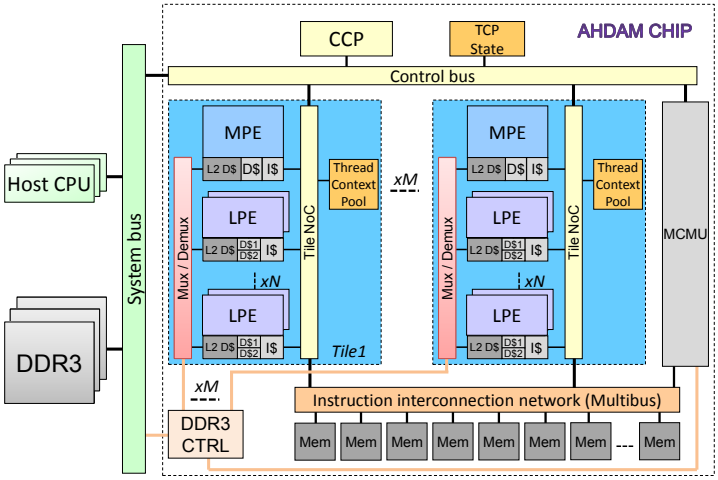

Fig. 2. AHDAM chip architecture

Memory units The AHDAM chip memory hierarchy is composed of a separated L2 instruction memory and data cache memory.

The instruction memory is a shared on-chip multi-banked SRAM memory that stores the codes of the tasks. The code size for all the tasks is known statically for a given application, thus the instruction memory size can be well dimensioned. In addition, the instruction memory is a shared memory, which is suitable for inter-tile task migration and load-balancing. Besides, it is implemented as a multi-banked memory instead of a single-banked multiple Read/Write ports memory, which is proven to be more efficient according to CACTI 6.5 tool [8]. In addition to a better area occupation, the multi-bank memory generates less

contention per bank when multiple Tiles are accessing simultaneously the instruction memory. This happens when the instruction codes for the tasks are stored in different memory banks.

On the other hand, since we cannot know in advance the application data set size, we implement a L2 data cache memory instead of an on-chip SRAM memory. Cache memories have a bigger area and are less area/energy efficient than SRAM memories. But caches facilitate the programmability since the memory accesses to the external DDR3 memory are transparent to the programmer and independent from the data set size. This eliminates the need for explicit data prefetching using DMA engines, which hardens the tasks decomposition and synchronization as it happens with the IBM Cell processor [9] for instance. All the L2 data cache memories are connected to an on-chip DDR controller, which transfers the data memory access requests to the off-chip DDR3 memories.

A special unit called MCMU (Memory Configuration and Management Unit) handles the memory configuration for the tasks. It divides the memory into pages. In addition, MCMU is responsible of managing the tasks' creation and deletion of dynamic data at runtime, and synchronizing their access with other tasks. There is one allocated memory space per data. A data identifier is used by tasks to address them. Each task has a write exclusive access to a data buffer. Since all the tasks have an exclusive access to data buffers, the data coherency problems are eliminated without the need for specific coherency mechanisms. A data access request is a blocking demand, and another task can read the data when the owner task releases its right. Multiple readers are possible even if the memory latency will increase with the number of simultaneous accesses.

The *Instruction interconnection network* connects the M Tiles to the multi-banked instruction memory. It is a **multibus**. According to the author [10], the multibus occupies less die area that other types of NoCs for small to medium interconnections, and has less energy consumption and memory access latency. The shared on-chip instruction memory is the last level of instruction memory. As we will see later in section 2.3, the execution model assumes that the instructions are already prefetched in the instruction memory.

Control unit In the AHDAM chip, the CCP (Central Controller Processor) controls the task prefetching and execution. The application CDFG is stored in dedicated internal memories. The CCP is a programmable solution that consists of an optimized processor for control, which is a small RISC 5-stage, in-order, and scalar pipeline core. Thus, the RTOS functionalities are implemented in software. In addition, the CCP has special interfaces from receiving/sending interruption demands to the computation units.

Computation units The AHDAM chip supports M Tiles. The CCP views a Tile as 1 computation unit. But actually, a Tile has one MPE (Master Processing Element) and N LPEs (Loop Processing Element). In addition, it has a special scratchpad memory called *Thread Context Pool* that stores the thread contexts to be processed by the LPEs. The *Thread Context Pool* represents the tasks

runqueue per Tile, thus AHDAM chip has M runqueues. The occupation status of all the Tiles' *Thread Context Pool* are updated in a special shared memory unit called the *TCP state*. The *TCP state* is shared by all the Tiles.

The MPE is the Master PE that receives the execution of a coarse-grain task or master thread from the CCP. It is implemented as a monothreaded processor with sufficient resources (ALUs, FPUs, etc...) for executing the tasks' serial regions. On the other hand, the LPE or Loop PE, is specialized in executing child threads that represent loop regions. The LPEs are implemented as blocked multithreaded VLIW processors with 2 hardware thread contexts (TC). In fact, the blocked multithreaded processor increases the LPE's utilization by masking the long access to the off-chip DDR3 memory that stalls the processors.

Each MPE and LPE has a private L1 I$, L1 D$, and L2 D$. For the multithreaded LPE, the L1 I$ is shared by both TCs, while the L1 D$ is segmented per TC. In this way, we privilege the execution of 2 child threads from the same parent thread, while limiting their interferences on the data memory level. The *Tile NoC*, which is a set of multiple busses interconnecting all the units to each other and to the external world (control and memory busses), is responsible of forwarding the cores' request accesses to the corresponding external unit. However, for the memory data accesses, the requests are grouped by a special MUX/DEMUX unit that forwards the data request to the DDR controller, then to the off-chip DDR3 memory. The *Tile NoC* provides one serial connection of the Tile to the external world, which eases the implementation of the Control and Instruction busses.

2.3 Execution model

In this section, we describe a typical execution model sequence in the AHDAM chip. At the beginning, the AHDAM chip receives an application execution demand from an external host CPU through the *System bus*. The CCP handles the communication. It fetches the application task dependency graph (CDFG), stores it in its internal memory, and checks the next tasks ready to run to be pre-configured by the MCMU. When the MCMU receives a task pre-configuration demand from the CCP, it configures the shared instruction memory space and allocates the necessary free space, then it fetches the tasks instruction codes from the off-chip DDR3 memory using an internal DMA engine, and finally it creates internally the translation tables. At this stage, the CCP is ready to schedule and dispatch the next tasks to run on available computation units through the *Control bus*.

Each task has serial regions and parallel regions. The parallel regions are the parallelized loop codes using a fork-join programming model such as OpenMP pragmas. For instance, let us consider the code example shown in Figure 3. It consists of 3 serial regions (S1,S2,S3) and 2 parallel regions (P1,P2). The thread execution is processed in 4 steps: 1) executing the serial region 2) forking the child threads 3) executing the child threads in parallel 4) joining the child threads.

Fork: The MPE executes the serial region of the task (S1). When it encounters a loop region using OpenMP pragmas (P1), the MPE executes a scheduling

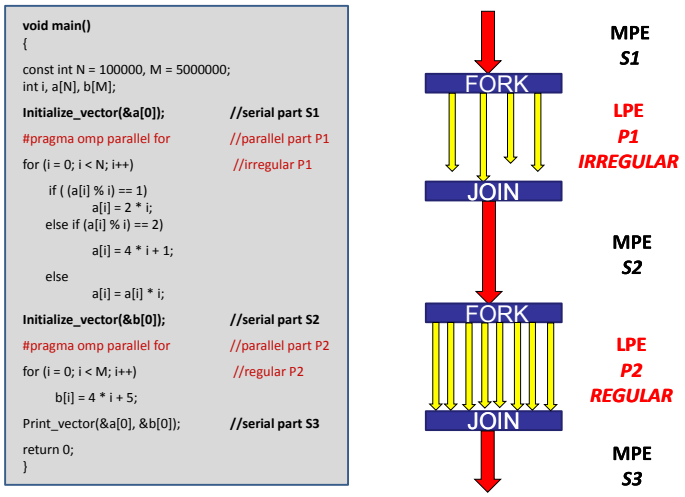

Fig. 3. A task code example of serial and parallel regions using OpenMP pragmas

algorithm that uses a heuristic to fork the exact number of child threads in the appropriate Tiles' *Thread Context Pool*. The scheduling algorithm is part of a modified OpenMP runtime. The heuristic determines the maximum number of parallel child threads required to execute the loop as fast as possible based on: 1) the data set size 2) the number of cycles to execute one loop iteration 3) the Tiles' *Thread Context Pool* occupation using the shared *TCP State* memory 4) the cost of forking threads in the local and other Tiles. If possible, the algorithm favors the local *Thread Context Pool* since the fork and join process are done faster by avoiding the access to multiple busses. However, in some cases, the local *Thread Context Pool* is full or not sufficient while the ones in other Tiles are empty. Therefore, the local MPE has the possibility of forking the child threads in others *Thread Context Pool* by verifying their availability using the shared *TCP state* memory. This can be the case for the parallel region P2 in Figure 3.

Execute: Then, each LPE (Loop PE) TC executes one child task instance from the local *Thread Context Pool* until completion. Forked parallel child threads are executed in a *farming model* by the LPEs. As soon as a LPE is idle, it spins on the local *Thread Context Pool* semaphore trying to fetch another child thread context. This type of execution model reduces the thread scheduling time and improves the LPEs occupation rate. In addition, it optimizes the execution of irregular for-loops. In fact, some for-loops have different execution paths that render their execution highly variable as shown in parallel region P1 in Figure 3. This scheduling architecture resembles the SMTC (Symmetric Multi-Thread Context) scheduling model, which has been shown to be the best scheduling architecture for multiple multithreaded processors [11].

In AHDAM, we implement a fork-join model with synchronous scheduling: the master thread forks the child threads, then waits to join until all the child threads have finished their execution. Therefore, during the execution of the parallel child threads, the MPE is in a dormant mode and is not preemptable by the CCP. There are 2 advantages from using this execution model: 1) the LPEs have a full bandwidth to the memory and are not disturbed by the MPE execution 2) easier 'join' process.

Join: When a child thread finishes execution, it sends a message to the corresponding MPE. The MPE waits until all the child threads have finished execution to join the process and continue execution of the serial region (S2).

3 Evaluation

In this part, we provide a case study scenario in order to evaluate the AHDAM chip performance. The AHDAM chip is simulated in a modified SESAM framework [12], which is a SystemC framework for modeling and exploration of asymmetric MPSoC architectures. SESAM supports the AHDAM programming model already discussed in section 2.1. It also has a wide range instruction-set simulators (ISS) including monothreaded MIPS1 and MIPS32, and cycle-accurate multithreaded MIPS1 [13]. The latter implements the blocked multithreading protocol and has 2 thread contexts. In this framework, MPEs are implemented as monothreaded MIPS32 24K with FPU, while LPEs are implemented as 2-threaded 3-way VLIW processor (1 ALU, 1 FPU, 1 ld/st). In fact, the performance results of the monothreaded 3-way VLIW are extracted from Trimaran simulator [14] due to its high simulation speed compared to the RTL model, then the results are injected in SESAM simulator. The CCP is a 32-bit monothreaded 5-stage RISC processor similar to MIPS1 R3000.

We choose an application that meets the future high-end embedded applications requirements as discussed in section 1. It is a radio spectrum sensing application from the telecommunication domain. This component, which can be found in cognitive radios, is developed by Thales Communications France (TCF). The radio spectrum sensing application sweeps the overall radio spectrum to detect unused spectrums. If an unused spectrum is found, it establishes a communication. We conducted a profiling on a modified version of the application, which provides a high degree of scalability for platform testing (number of cores, etc...) and is not fully optimized for sensing processing. This application has been developed within the SCALOPES project. The application is characterized by its high computation requirements and its adaptive reconfiguration (**dynamism**). The application supports different execution modes. For our evaluation, we choose the following options: high-sensitivity (frequency sample 102.4 MHz), 6 buffers, 100 ms buffer size every 1 sec. This gives us a computation requirement of *75.8 GOPS*, a data set of *432 MB*, and a real-time deadline of *6 seconds*. In addition, we examined the hot spots in the code where most of the application time is spent, and we noticed that 99.8 % of the loop regions can be parallelized by OpenMP.

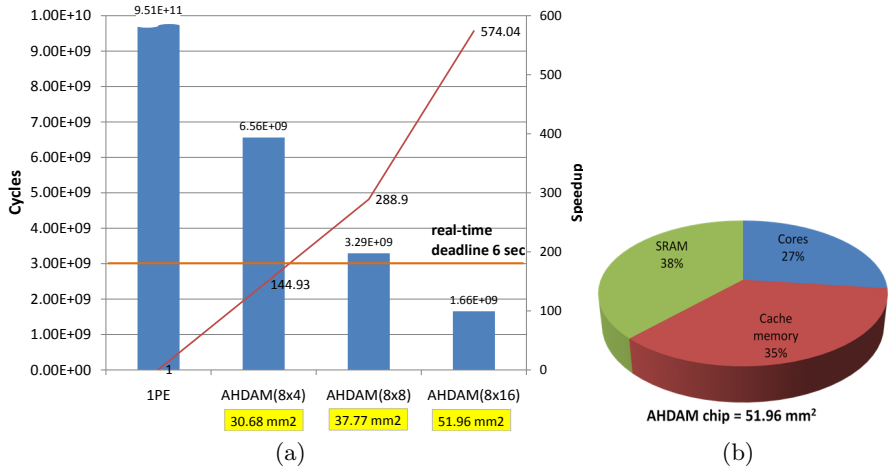

Fig. 4. a) Execution time of radio-sensing application on 1 PE v/s AHDAM chip with 8 Tiles and 4/8/16 LPEs on 6 buffers. The real-time deadline is 6 seconds b) Surface repartition of AHDAM with 8 Tiles and 16 LPE/tile at 40 nm technology

Initially, the radio sensing application is built to run sequentially on a mono-threaded processor. The task level parallelism is explicitly expressed by inserting specific pragmas. Then, *PAR4ALL* cuts the application in a set of tasks according to these pragmas, generates communication primitives to implement a double buffer streaming processing, and the corresponding CDFG control graph. For this paper, 19 tasks are generated. Once independent tasks are generated, *PAR4ALL* identifies netloops and inserts OpenMP pragmas. The loop parallelism is detected during runtime depending on the resources occupation. Also some loops are irregular, which means a variable execution time between the child threads.

We run the radio-sensing application on 2 processing systems running at 500 MHz:

- Sequential version: 1 MIPS32 24K processor with a FPU, and a sufficient on-chip memory for data and instructions (432 MB). The memory access time to the on-chip memory is 10 cycles, as well as the L2$ memory. The processor has 4KBI and 8KBD L1$, and a 32KB L2$.
- Parallel version: AHDAM chip that is configured with 8 Tiles and 4/8/16 LPEs per Tile. The MPE has 4KBI and 8KBD L1$, while the LPE has 1KBI and 2KBD (1KB per TC) L1$, and a 32KBD L2$. The on-chip instruction memory is equal to 1.5 MB, which is the total size of the tasks' instructions and stack memories. The memory access to the on-chip instruction memory takes 10 cycles, as well as the L2 D$. For the off-chip DDR3 memory, the access time is 50 cycles.

In Figure 4, we compare the execution time of these processing systems. The results show that the AHDAM chip with 4 LPE/tile has a speed-up of 145 compared to a single-processor system. And since the application has lot of LLP, the acceleration goes up to 574 for 16 LPE/tile. In fact, Trimaran compiler optimizations contribute with the LLP exploitation to this high speed-up factor. It is clear that only AHDAM(8x16) with 136 processors is able to meet the real-time deadline constraints of 6 seconds for the radio-sensing application. In addition, we estimated the chip size (processors + memories) at 40 nm technology of both processor systems giving the radio-sensing application conditions. The cache memories and SRAM memories are estimated using CACTI 6.5 tool. As for the processors, we synthesized the multithreaded VLIW LPE and the CCP, while the MPE core area is given at MIPS website. AHDAM(8x4) with 40 processors has an estimated die area of 30.67 mm^2, while AHDAM(8x16) with 136 processors is 51.92 mm^2, which is only 69.2% bigger. The surface repartition of AHDAM(8x16) is shown in Figure 4(b). We can notice that the computing cores take 27% of the overall die area, which is quite a good number compared to recent MPSoC architectures. In fact, the key design parameter taken in AHDAM design is to reduce the size of the on-chip memory and integrate instead more efficient processors for computation.

4 Conclusion

This paper has presented an asymmetric homogeneous with dynamic allocator manycore architecture, called AHDAM. The AHDAM chip is designed to tackle the challenges and requirements of future high-end massively parallel dynamic embedded applications. The AHDAM chip units are chosen to increase the transistor/energy efficiency.

We presented the AHDAM chip programming model and its automatic compilation tool that takes as input a sequential application, then decomposes it into multiple parallel independent tasks (TLP), and finally inserts OpenMP pragmas at the loop level (LLP). To study its performance, we used the radio spectrum sensing application from the telecommunication domain. We simulated the execution of sequential and parallel versions of the application on 2 platforms: a single processor with sufficient on-chip memory for the application data set, and the AHDAM chip with a variable number of processors. The results show that the application on the AHDAM chip with 136 processors has an execution time 574 times faster than on the single-processor system, while meeting the real-time deadline and occupying 51.92 mm^2. The speed-up is almost scalable with respect to the number of LPEs since the application has lot of LLP.

For future enhancements, we would like also to compare AHDAM with other MPSoC architectures excluding the GPU architecture. In fact, with AHDAM chip, we are targeting applications that are highly dynamic with lot of TLPs. This execution model is not adequate with the GPU execution model, thus not comparable with AHDAM chip. In addition, we aim to explore AHDAM runtime environment and the heuristics for forking the child threads between the Tiles.

Acknowledgements

Part of the research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 100029. The authors would like to thank Florian Broekaert from Thales Communications France for providing the radio-sensing application. Also, we would like to thank Beatrice Creusillet, Ronan Keryel and Pierre Villalon from HPC Project for providing the *PAR4ALL* compilation tool.

References

1. Marc Duranton, Sami Yehia, Bjorn De Sutter, Koen De Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, Nacho Navarro, Alex Ramirez, Olivier Temam, and Mateo Valero. *The HiPEAC Vision*. HiPEAC network of excellence , 2010.
2. Lionel Lacassagne and Bertrand 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.
3. D.W. Wall. Limits of instruction-level parallelism. In *Int'l Conf. on Architectural Support for Programming Languages and OperatingSystems (ASPLOS)*, Santa Clara, USA, April 1991.
4. M. Bertogna, M. Cirinei, and G. Lipari. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, April 2008.
5. N. Ventroux and R. David. The SCMP architecture: A Heterogeneous Multiprocessor System-on-Chip for Embedded Applications. *Eurasip*, 2009.
6. HPC Project: PAR4ALL tool: http://hpc-project.com/pages/par4all.htm. online.
7. OpenMP: http://www.openmp.org. online.
8. Naveen Muralimanohar and Rajeev Balasubramonian. CACTI 6.0: A Tool to Model Large Caches.
9. M. W. Riley, J. D. Warnock, and D. F. Wendel. Cell Broadband Engine processor: Design and implementation. *IBM Journal of Research and Development*, 51(5):545 –557, sept. 2007.
10. A. Guerre, N. Ventroux, R. David, and A. Merigot. Hierarchical Network-on-Chip for Embedded Many-Core Architectures. In *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, pages 189 –196, may 2010.
11. C. Bechara, N. Ventroux, and D. Etiemble. Comparison of different thread scheduling strategies for Asymmetric Chip MultiThreading architectures in embedded systems. In *14th Euromicro conference on Digital System Design (DSD 2011)*, Oulu, Finland, September 2011.
12. 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.
13. C. Bechara, N. Ventroux, and D. Etiemble. Towards a Parameterizable cycle-accurate ISS in ArchC. In *IEEE International Conference on Computer Systems and Applications (AICCSA)*, Hammamet, Tunisia, May 2010.
14. Bhuvan Middha, Anup Gangwar, Anshul Kumar, M. Balakrishnan, and Paolo Ienne. A Trimaran based framework for exploring the design space of VLIW ASIPs with coarse grain functional units. pages 2–7, 2002.