UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MATEUS BECK RUTZIG

A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Efficient ILP and TLP Exploitation

Thesis presented in partial
fulfillment of the requirements
for the degree of Doctor of
Computer Science

Prof. Dr. Luigi Carro
Advisor

Porto Alegre
January/2012

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Beck Rutzig, Mateus

A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Efficient ILP and TLP Exploitation/Mateus Beck Rutzig – Porto Alegre: Programa de Pós-Graduação em Computação, 2012.

119 p.:il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2012. Orientador: Luigi Carro.

1.Sistemas Multiprocessados 2.Arquiteturas Reconfiguráveis 3.Sistemas Embarcados I. Carro, Luigi II. Título

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Alvaro Freitas Moreira
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic and Logic Unit
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuits
BT	Binary Translation
CAD	Computer Aided Design
CCA	Configurable Compute Array
DIM	Dynamic Instruction Merging
DSP	Digital Signal Processor
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
ILP	Instruction Level Parallelism
TLP	Thread Level Parallelism
IPC	Instructions per Cycle
RAW	Read After Write
RFU	Reconfigurable Functional Unit
RPU	Reconfigurable Processor Unit
SIMD	Single Instruction – Multiple Data

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

As the number of embedded applications is increasing, the current strategy of several companies is to launch a new platform within short periods, to execute the application set more efficiently, with low energy consumption. However, for each new platform deployment, new tool chains must come along, with additional libraries, debuggers and compilers. This strategy implies in high hardware redesign costs, breaks binary compatibility and results in a high overhead in the software development process. Therefore, focusing on area savings, low energy consumption, binary compatibility maintenance and mainly software productivity improvement, we propose the exploitation of Custom Reconfigurable Arrays for Multiprocessor System (CReAMS). CReAMS is composed of multiple adaptive reconfigurable systems to efficiently explore Instruction and Thread Level Parallelism (ILP and TLP) at hardware level, in a totally transparent fashion. Conceived as homogeneous organization, CReAMS shows a reduction of 37% in energy-delay product (EDP) compared to an ordinary multiprocessing platform when assuming the same chip area. When a variety of processor with different capabilities on exploiting ILP are coupled in a single die, conceiving CReAMS as a heterogeneous organization, performance improvements of up to 57% and energy savings of up to 36% are showed in comparison with the homogenous platform. In addition, the efficiency of the adaptability provided by CReAMS is demonstrated in a comparison to a multiprocessing system composed of 4-issue Out-of-Order SparcV8 processors, 28% of performance improvements are shown considering a power budget scenario.

Keywords: Multiprocessors, Reconfigurable Architectures, Instruction and thread level parallelism.

1 INTRODUCTION

Industry competition in the current wide and expanding embedded market makes the design of a device increasingly complex. Nowadays, embedded systems are in a transition process from closed devices to a world in which the products have to run applications, previously unforeseen at design time, during their whole life cycle. Thus, companies are always enhancing the repository of applications to sustain their profit even after the product has been sold. Current cell phones, a clear example of devices that explore today´s convergence, are capable of downloading applications during the product life cycle. Android, Google´s software framework, in less than three years of existence offers 380,297 applications for downloading, while Apple´s platform, iOS, has three times as many applications as Android. Apple becomes the most valuable company of United States of America after four years of iPhone. The customers are attracted to have in their devices more and more applications such as games, text editors and VoIP communication interfaces.

However, most embedded products are mobile and hence battery-powered. Hardware designers should cope with well-known design constraints such as energy consumption, chip area, process costs and processing capability. The strategy to embed different applications during the product life cycle produces new design challenges, which makes embedded platforms development even more difficult. Thus, the current embedded system design is not only constrained by the existing applications requirements. To reach a wider market, one should carefully conceive the design to cope with the requirements of the wide software repository that will be developed even after the product deployment.

The fast deployment of embedded applications dynamically enlarges the range of different types of code that the platform should execute. Consequently, the life cycle of modern embedded products is getting increasingly small since the hardware platform was not originally built to handle such software heterogeneity. A few years ago, cell phones manufacturers launched a major product line per year, what was suitable to supply the performance required by the new applications launched during this period. The life cycle of a cell phone has shortened to achieve the requirements of the new applications (HENKEL, 2003), which implies in less revenue per new design due to the reduced product lifetime. However, companies try hard to stretch their product lines to amortize the costs and to increase the profits per design. Typically, companies use the natural life cycle of the applications in the market as a strategy to stretch the product life cycle and to avoid costs with frequent hardware redesigns. The application life cycle is divided into three phases (BRANDAO e WYNN, 2008): introduction, growth and

maturity. During the introduction phase, which reflects the time when the application is launched in the market (e.g. a new video decoding standard such as H.264). Due to the doubts about the consumer acceptance, the logical behavior of a new application is described using well-known high-level software languages (e.g. C++, Java and .NET), supported by the platform tool chain, which could possibly cause overload in some parts of the underlying platform. The general-purpose processor would be responsible for executing the new application. In this life cycle step, companies still avoid hardware costs, since the target platform is the very same of the previous product, or very close to it. After market consolidation, the growth and maturity phase start, thanks to the widespread use of the application in different products. At this time, a redesign of the hardware platform is mandatory to shrink the gap between the application and the hardware achieving better energy/performance execution.

Generally, two approaches are used to supply the efficient execution of the latest embedded application. In the first approach, new instructions are added to the original instruction set architecture (ISA) of the platform. This approach aims at solving performance bottlenecks created by massive execution of certain application parts with well-defined behavior. For instance, this used to be the scenario of the last generation of embedded systems. After the profiling and evaluation phase, parts of applications that contain a similar behavior are implemented as specialized hardwired instructions. These instructions will extend the processor ISA to assist a delimited range of applications (GONZALEZ, 2000). Since multimedia applications and digital filters are massively used in the embedded systems field, current ARM processors have implemented DSP to efficiently execute, in terms of energy and performance, these kinds of applications.

A second technique uses a more radical approach to close the gap between the hardware and the embedded application. Application Specific Instruction Set Processors (ASIPs) is a technique used to implement the entire logic behavior of the application in hardware. ASIP development can be considered better design solution than ISA extensions, since it provides higher energy savings and performance. Nowadays, such an approach is widely explored by the leading companies of the market. Open Multimedia Application Platform (OMAP), designed by Texas Instruments, comprises one or two ARM processors that are surrounded by several ASIPs (communication, graphics and audio standards), each one of them with its particular architectural characteristics to efficiently execute a restricted type of software. Companies usually employ ASIPs to obey the demand for new applications in the shortened design time scenario and to reach the performance requirements imposed by the market.

However, the use of ASIPs causes frequent platform redesign that besides increasing costs in hardware deployment also affects software development process. While it is difficult to develop applications for the current platform where one can find up to 21 ASIPs. Such difficulty will increase in the coming decade since it is expected that 600 different ASIPs will be needed to cover the growing convergence of applications for the embedded devices (SEMICONDUCTORS, 2009). To soften such a complexity, hardware companies (e.g. OMAP and Nvidia) provide particular tool chains to support the software development process. This tools chain makes the implementation details of the platform transparent to the software designers, even in the presence of a great number of ASIPs. However, each release of a platform relies on tool chain modifications, since it must be aware of the existence of the underlying ASIPs. Thus, changes on both software and hardware are mandatory when ASIPs are employed

to supply the energy and performance efficiency demanded for the current embedded platforms.

Despite the great advantages shown in the employment of some ISA extensions and ASIPs, such approaches rely on frequent hardware and software redesigns, which go against the current market trend on stretching the life cycle of a product line. These strategies attack only a very specific application class, failing to deliver the required performance while executing applications for those behaviors that have not been considered at design time. In addition, both ISA extensions and ASIP employed in the current platforms only explore instruction level parallelism (ILP). Aggressive ILP exploitation techniques no longer provide an advantageous tradeoff between the amount of transistors added and the extra speedup obtained (MAK, 1991) .

Due to the aforementioned reasons, the foreseen scenario dictates the need for changes in the paradigm of the hardware platform development for embedded systems. Many advantages can be obtained by combining different processing elements into a single die. The execution time can clearly benefit since several different parts of the program could be executed concurrently in processing elements. In addition, the flexibility to combine different processing elements, in terms of performance, appears as a solution to the heterogeneous software execution problem. The hardware developers can select the set of processing elements that best fit with the heterogeneity running in their designs.

Multiprocessing systems provide several advantages, and three of them highlight among all: performance, energy consumption and validation time. The life cycle of these devices has halved in comparison with products of last decade. Validation time appears as an important consumer electronics constraint that should be carefully handled. Researches explain that 70% of the design time is spent in the platform validation (ANANTARAMAN, SETH, *et al.*, 2003), thus being an attractive point for time-to-market optimization. Considering this subject, the use of multiprocessing system softens the hard task to shrink time-to-market. Commonly, such an approach is built by the combination of validated processing elements that are aggregated into a single die as a puzzle game. Since each puzzle block reflects a validated processing element, the remaining design challenge is to assemble the different blocks. Actually, the designers should select a communication mechanism to connect the entire system, which eases the design process by the use of standard communication mechanisms such as buses or network on chips (NoC).

Multiprocessing systems introduce a new parallel execution paradigm aiming to overcome the performance barrier created by the limits of instruction level parallelism. Nowadays, the software team should manually detect the parts of the program that could be executed in parallel. The hardware team is only responsible for the encapsulation process of a certain processing elements and for the communication infrastructure. When considering ILP exploitation, the complexity on extracting the parallelism moves to software team for multiprocessing chips, since there is no strong methodology that can support automatic software parallelization. The software team is responsible for the non-trivial task of spawning and distributing the code among the processing elements. Due to this reason, software productivity arises as the hardest challenge in a multiprocessing system design, since the applications should be launched as fast as possible to supply the demand of the market. The binary code of these applications should be as generic as possible to provide compatibility among different products and

platforms. In addition, the communication infrastructure should be efficient enough to smooth the latency of the inter-thread communication.

The four quadrants plotted in the Figure 1 show the strengths and the weaknesses of the existing hardware strategies used to design a multiprocessor platform. This figure considers the organization and the architecture of the multiprocessing platforms. The main strategy used for leader companies in the market is building embedded platforms as illustrated in the lower left quadrant of the Figure 1. Such strategy could be area inefficient, since it relies on the employment of a particular ASIP to efficiently cover the execution of software with a restricted behavior in terms of ILP and TLP. Each release of a platform will not be transparent to the software developers, since together with a new platform, a new version of its tool chain with particular libraries and compilers must be provided. Besides the obvious deleterious effects on software productivity and compatibility for any new hardware upgrade, there will also be intrinsic costs of new hardware and software developments for every new product.

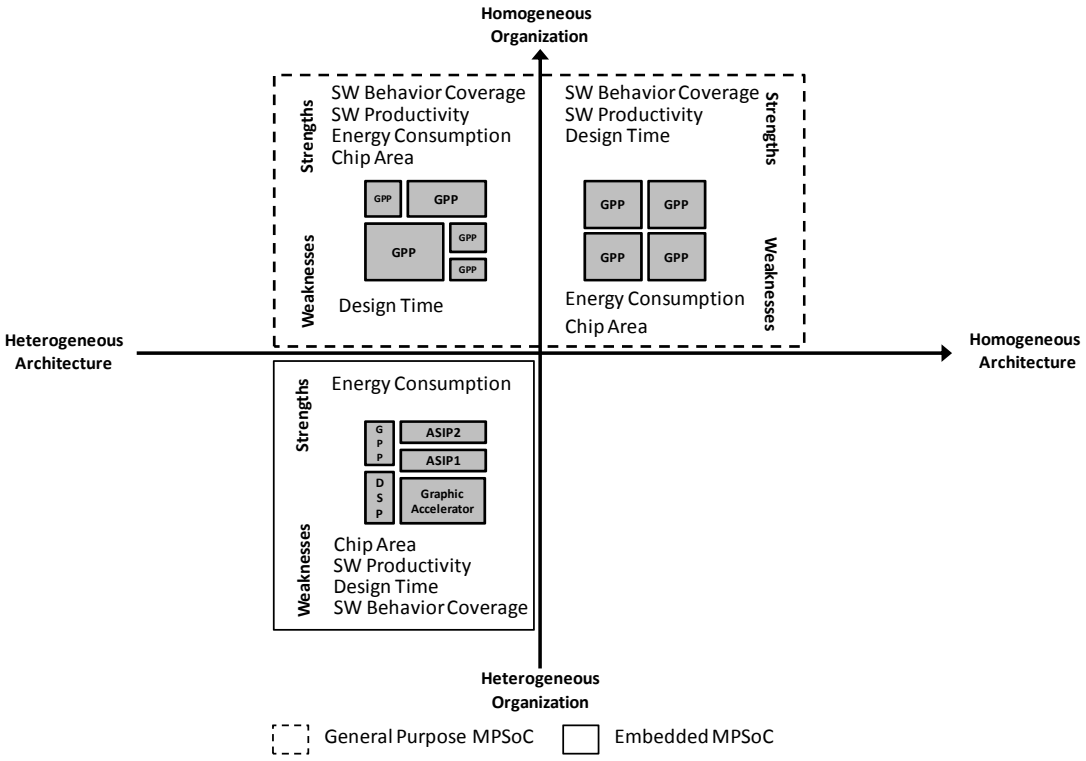


Figure 1. Different Architectures and Organizations

On the other hand, the upper right quadrant of the Figure 1 illustrates the multiprocessing systems that are composed of multiple copies of the same processors, in terms of architecture and organization. Typically, such strategy is employed in general-purpose platforms where performance is mandatory. However, energy consumption is also getting relevant in this domain (e.g. it is necessary to reduce energy costs in datacenters). In order to cope with this drawback, the homogeneous architecture and heterogeneous organization, shown in the upper left quadrant of the Figure 1, has been emerging to provide better energy and area efficiency than the other two aforementioned platforms. This approach brings the cost of higher design validation time, since many different organizations of processors are used. However, it has the advantage of implementing a unique ISA, so the software development process is not

penalized. It is possible to generate assembly code using the very same tool chain for any platform version maintaining full binary compatibility for the already developed applications. However, the scheduling of the threads appears as an additional challenge when heterogeneous organization approach is used. Threads that have different levels of instruction level parallelism should be assigned to processors with different performance capabilities.

Software partitioning is a key feature in multiprocessing environments. A computational powerful multiprocessing platform becomes useless if threads of a certain application show significant load unbalance ratio. Usually, it is given by the poor quality of the software partitioning, or by the nature of the application that does not provide a minimum thread level parallelism to be explored. Amdahl's law shows that the speedup of a certain application is limited by its sequential part. Therefore, if an application needs 1 hour to execute, being 5 minutes sequential (almost 9% of entire application code), the maximum speedup provided for a multiprocessing system is 12 times, no matter how many processing elements are available.

Figure 2 shows the performance of some well-known embedded applications that were split in threads using a traditional shared-memory parallel programming language. As can be seen, the performance of these applications does not scale as the number of processors increases, when executed on a multiprocessor system composed of multiple copies of five-stage pipeline RISC processors. In the best case of the examples, even overlooking inter-thread communication costs, a speedup of nine times is achieved when 64 processors are used. Clearly, these embedded applications are good examples of Amdahl's law, demonstrating that multiprocessing systems can fail to accelerate applications that have a meaningful sequential part. Since there is a limit of TLP for most applications (BLAKE, DRESLINSKI, *et al.*, 2010), standalone TLP exploitation does not provide the energy and performance optimization demanded for current embedded designs.

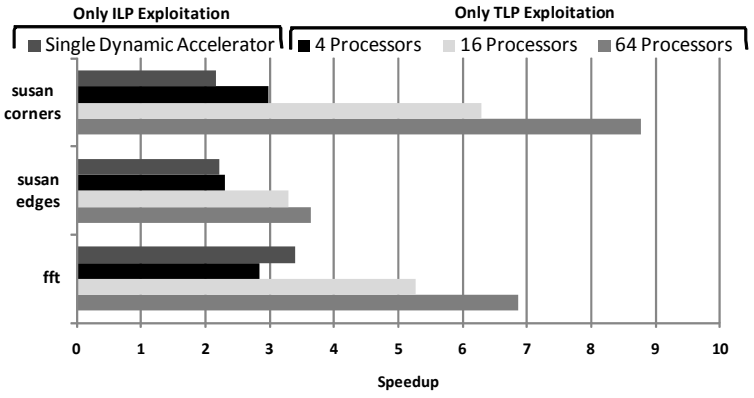


Figure 2. Speedup of homogeneous multiprocessing systems on embedded applications

The ideal platform would have the hardware benefits from the third quadrant of Figure 1, with the ease for software development of the second quadrant, without any cost associated to new hardware development. It means that, the physical structure of the hardware could be homogenous, since chip area is no longer a drawback for billion transistor technologies. Nevertheless, it is mandatory that the costs, such as power and energy consumption, virtually must behave as a heterogeneous organization. However, this can only be achieved if the available hardware has the ability to be tuned for each different application or even program phase on the fly. Dynamic reconfigurable

architectures have already shown to be very attractive for embedded platforms, since they can adapt the fine grain parallelism exploitation (i.e. at instruction level) to the application requirements at run-time (CLARK, KUDLUR, *et al.*, 2004) (LYSECKY, STITT e VAHID, 2004). However, besides having restricted thread level parallelism, embedded applications also exhibit limits of instruction level parallelism. Thus, gains in performance when such exploitation is employed tend to stagnate, even if a huge amount of resources is available in the reconfigurable accelerator. The "Single Dynamic Accelerator" bars of the Figure 2 illustrate the above claim. This assumption considers the performance of a dynamic reconfigurable architecture with an area equivalent to sixteen five-stage pipeline RISC processor. Although it is faster than a RISC processor executing a single thread, outperforming four processors on running the Fast Fourier Transform (FFT). The standalone ILP exploitation of the single dynamic accelerator does not provide an advantageous trade-off between area and performance if compared to the multithreaded version of the remaining benchmarks. Summarizing, this figure indicates that ILP as well as TLP alone do not provide meaningful area-performance tradeoff, considering a heterogeneous software environment.

The state-of-art of the multiprocessing systems with both ILP and TLP exploitation is very divergent, if one considers the complexity of the processing element. At one side of the spectrum, there are multiprocessing systems composed of multiple copies of simple cores to better explore coarse grain parallelism of highly thread-based applications (HAMMOND, HUBBERT, *et al.*, 2000) (ANDRE, BARROSO, *et al.*, 2000). At the other side, there are multiprocessor chips assembled with few complex superscalar/SMT processing elements, to explore applications where ILP exploration is mandatory. There is no consensus on the hardware logic distribution in a multiprocessing environment to explore the best of ILP and TLP together regarding a wide range of application classes. Considering the wide range of instruction level parallelism that current applications exhibit, there is a large design space to explore by creating platforms composed of processors with different capabilities on exploiting ILP. Despite the technology allows the encapsulation of billion transistors in a single chip, area could be saved and the performance of the homogeneous platform could be maintained by exploiting the diversity of computational capabilities of the heterogeneous organization. Such strategy relies on scheduling algorithm that would correlate the intrinsic characteristics of the threads, such as load unbalance and ILP, with the computational capability of the available processors. (KUMAR, FARKAS, *et al.*, 2003) (KUMAR, JOUPPI e TULLSEN, 2006)

Summarizing, an ideal multiprocessing system for embedded devices should be composed of replication of generic processing elements that could adapt to the particularities of the applications, throughout at the product life cycle. This platform should emulate the behavior, in terms of performance and energy, of the ASIPs that are successfully employed in the current embedded platforms. At the same time, in contrast to such platforms, the use of the same ISA for all processing elements is mandatory to increase software productivity by avoiding time spent on tool chain modifications, and to maintain the binary compatibility to the already developed applications. This ideal platform would be able to attack efficiently the whole spectrum of application behaviors: those that contain dominant thread level parallelism and those single threaded applications. However, the platform should be conceived as a heterogeneous organization to provide a best fitting between the heterogeneous characteristics that the applications exhibit and the necessary processing capability to execute them. Moreover,

the number of processing elements should be careful investigated. Since the overall system performance could be affected by inter-thread communication costs with the growth of processing elements. This way, the hypothesis is that by using such strategy one could reach a satisfactory tradeoff in terms of energy, performance and area, without extra software and hardware costs.

1.1 Contributions

Considering all motivations discussed before, the first goal of this work is focused on reinforcing, by the use of an analytical model, that the employment of a standalone level of parallelism exploration does not provide a meaningful energy-performance tradeoff when a heterogeneous application environment is handled. In addition, this study gives some clues about the ratio of hardware deployment in multiprocessing chips, in terms of fine and coarse grain parallelism exploitation, to achieve a balanced architecture in terms of area and performance. A Network-on-Chip is also modeled to investigate the impact of inter-thread communication latency over the gains obtained by thread level parallelism exploitation.

In this scenario, we propose a platform based on Custom Reconfigurable Arrays for Multiprocessor System (CReAMS), by merging two different architectural concepts: reconfigurable architectures and multiprocessing systems. In the first step of this work, CReAMS is built as homogeneous on both architecture and organization. However, it virtually behaves as a homogeneous architecture with a heterogeneous organization. Thanks to its dynamic adaptive hardware, coupled to each basic processor, CReAMS takes advantage of the flexibility provided by the reconfigurable architecture.

This system is capable of transparently explore (no changes in the binary code are necessary at all) the fine-grained parallelism of the individual threads, offering much greater ability to adapt to the ILP demands of the applications, while at the same time it makes the most of the available thread parallelism. The coarse-grained parallelism exploitation does not rely on special tools employment since it is explored by well-known application programming interfaces (e.g. OpenMP and POSIX threads), making CReAMS execution independent of any particular software partitioning process. Thus, dynamically and in a transparent fashion it is possible to balance the best of both thread and instruction parallelism levels. This way, any kind of code, from those that present high TLP and low ILP to those that are exactly the opposite are accelerated. CReAMS achieves performance improvement, providing less energy consumption, but with the software productivity of a multiprocessor device based on homogeneous architecture. In addition, a single tool chain is used for the whole platform and for any new version launched, with full binary compatibility.

Aiming at showing the potential of CReAMS platform on adapting to a wide range of software behaviors, we selected applications from general purpose (e.g. SPEC OMP2001), parallel (e.g. Splash2) and embedded benchmark suites (e.g. MiBench). The experimental setup was supported by simulation using the SparcV8 ISA model supplied by Simics instruction set accurate simulator (MAGNUSSON, CHRISTENSSON, *et al.*, 2002). CReAMS measurements were obtained through replication of cycle accurate simulators that model the behavior of the basic processing element of CReAMS, named as Dynamic Adaptive Processor (DAP). The cycle accurate simulators precisely calculate threads synchronization, as barriers and locks. CReAMS implements thread communication through shared-memory mechanism and, as already cited, supports the well-known application programming interfaces, which makes the thread spawning

process transparent to the hardware. Performance improvement and energy savings were demonstrated when comparing CReAMS to ordinary multiprocessing system composed of multiple copies of pipelined SparcV8 processors when considering the same chip area for both designs.

Since very interesting performance and energy results were obtained, considering CReAMS as homogeneous organization platform, aiming at reducing the area occupied by CReAMS, we investigated the advantages of using DAPs with different processing capabilities, taking advantage of the heterogeneous organization. One of the motivations for such design space exploitation is the diversity of instruction level parallelism available in a heterogeneous application workload. Some threads may have larger amount of instruction level parallelism than others, which can be exploited by a DAP that can issue many instructions per cycle.

However, the powerful DAP could be assigned to execute a certain thread that requires tiny ILP exploitation, consuming more power than a simpler core that would better matched to the characteristics of such thread. This wrong thread assignment could cause load-unbalanced execution, significantly affecting the overall execution time. Thus, DAPs with different processing capabilities bring a diversity of ILP opportunities to explore, opening room to achieve larger area savings and less power consumption than the homogeneous organization strategy. However, it also brings a need for a thread scheduling strategy that matches to the performance requirements of a certain thread to maintain the performance shown by the homogeneous DAPs. Thus, we developed a simple thread scheduling algorithm only to prove the need for a dynamic thread scheduling strategy when heterogeneous organizations are employed. The scheduling strategy assigns threads to DAPs with different ILP exploitation capabilities considering the number of executed instructions. As all DAPs have the same instruction set in the heterogeneous environment, the transparency offered by the homogeneous CReAMS in the software development process is not affected, which maintains the same software productivity.

Chapter 2 presents the related work, discussing issues related to reconfigurable architectures, multiprocessing systems based on heterogeneous architecture. We also describe the contribution and main novelty of this work, when comparing against these other studies. In Chapter 3, we first discuss, using an analytical model, the potential of standalone exploitation of the instruction and thread level parallelism. We model a multiprocessing architecture composed of several simple and homogeneous cores, and we compare it to the modeling of a superscalar architecture in terms of performance and energy. The impact of the communication infrastructure is also analytically modeled. After that, in the Chapter 4, we present the structure of the CReAMS platform. Chapter 5 shows the methodology and tools employed to gather the results. The performance, energy and area results regarding the homogeneous organization of CReAMS are demonstrated in this Chapter. After, results considering CReAMS conceived as heterogeneous organization are shown. Finally, the performance of CReAMS is compared to a 4-issue Out-Of-Order SparcV8 multiprocessor. Chapter 6 discusses the future works and concludes this work.

2 RELATED WORK

In this chapter, we review traditional works that explore reconfigurable fabric to accelerate single-threaded applications. After, we show some approaches that use multiprocessing systems in the commercial and academic field. Finally, the characteristics of many researches that employ reconfigurable architecture in a multiprocessing environment are shown. At the end of this section, we analyze our approach, linking its similarities/dissimilarities with the other researches that use the same strategy.

2.1 Single-Threaded Reconfigurable Systems

Although there is no common criteria over the classification of the single-threaded reconfigurable system, careful study with respect to coupling, granularity and reconfiguration type is presented in (HAUCK e COMPTON, 2002).

In a reconfigurable architecture design, the choice of the coupling between the reconfigurable data path and the basic processor is crucial for performance. As can be seen in Figure 3, tightly coupled is the classification given for the reconfigurable fabric implemented as an additional functional unit (FU) of the processor. As the communication between both elements occurs only inside the chip, its high throughput is a benefit over the loosely coupled reconfigurable fabric. There are many sub-classifications of loosely coupled fabrics. When the fabric is classified as co-processor, the data path is implemented outside the chip, as shown in Figure 3. Design constraints guides the coupling employment, when there is not enough silicon area to store the reconfigurable fabric, loosely coupled architectures are used, where an external bus is responsible for the communication between processor and reconfigurable fabric. Attached is the coupling strategy that connects the reconfigurable fabric between cache memory and the I/O interface. The communication cost is high, however, lower than the standalone strategy that connects the reconfigurable data path to the I/O interface.

The size and complexity of the basic reconfigurable elements is referred to as the block's granularity. For example, one could build a reconfigurable fabric as replications of one-bit width adders as a basic reconfigurable element. However, 32-bits width adder could be encapsulated as a black box building a coarser basic reconfigurable element. This latter design provides lower reconfiguration flexibility than the former, since usage of adders that need less than 32-bits width would always occupy a basic element. On the other hand, a simple controller is required as the granularity becomes coarser, so fewer bits are used to reconfigure the whole fabric.

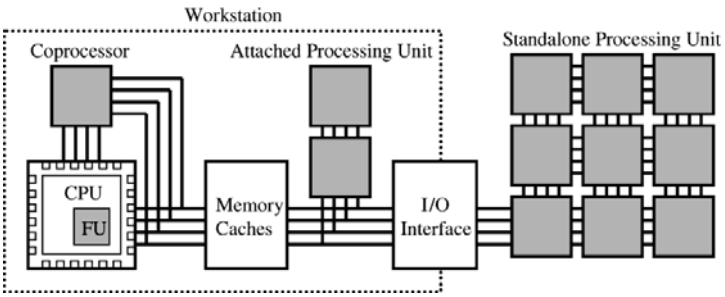


Figure 3. Coupling setups (HAUCK e COMPTON, 2002)

Static reconfiguration is exploited by several researches as strategy to extract, at a compile time, the most suitable parts of the application code to efficiently execute in the reconfigurable fabric. This strategy avoids any kind of execution time task by adding compilation phase to discover the suitable parts of the application code. However, it breaks the binary compatibility since it relies on some kind of source code modification. In addition, the time-to-market constraint can be affect as a new compilation phase is inserted.

Many successful reconfigurable fabrics employ static reconfiguration. Processors like Chimaera (HAUCK, FRY, *et al.*, 2004) have a tightly coupled reconfigurable array in the processor core, working as an additional functional unit, limited to combinational logic only. This simplifies the control logic and diminishes the communication overhead between the reconfigurable array and the rest of the system. Look-up-tables are used as a basic reconfigurable block, which lead to high reconfiguration costs, as in memory footprint as well as in reconfiguration time. The GARP machine (WAWRZYNEK, 1997) is a MIPS compatible processor with a loosely coupled reconfigurable array. The communication is done using dedicated move instructions, as one also employs look-up-tables as basic reconfigurable blocks the same design costs of Chimaera are produced by this approach.

Piperench (GOLDSTEIN, SCHMIT, *et al.*, 2000) proposes a pipeline-based reconfigurable fabric attached to the processor to reduce the reconfiguration/execution time of FPGAs. This approach use a technique, named as virtualization, to reduce area costs of the reconfigurable fabric. The upper side of this Figure (Figure 4(a)) shows an example of a Piperench execution without the virtualization technique. In this case, the application was divided in 5 parts and takes 7 cycles to be configured and executed, since no parallelism in the configuration/execution process is provided. The lower side of the Figure 4 shows the virtualization technique, 3 cycles are needed to execute the same application. The reuse of the same data path stage at different periods is the key factor to achieve high performance with low area.

More recently, new reconfigurable architectures, very similar to the dataflow approaches, were proposed. For instance, the TRIPS is based on a hybrid von-Neumann/dataflow architecture that combines an instance of coarse-grained, polymorphous grid processor cores with an adaptive on-chip memory system (SANKARALINGAM, NAGARAJAN, *et al.*, 2004) . To better explore the application parallelism and utilize the available resources, TRIPS uses three different modes of execution, focusing on instruction-, data- or thread level parallelism. Wavescalar (SWANSON, 2007), in turn, totally abandons the program counter and the linear von-Neumann execution model that could limit the amount of exploited parallelism. The major difference between this approach and the conventional systems is that there is no

central processing unit at all, which is replaced by many distributed processing nodes. In agreement with the previous examples, one can also refer to Molen (VASSILIADIS, WONG, *et al.*, 2004). All cited approaches still rely on static reconfiguration to achieve code optimization and better resource utilization on applying reconfigurable logic.

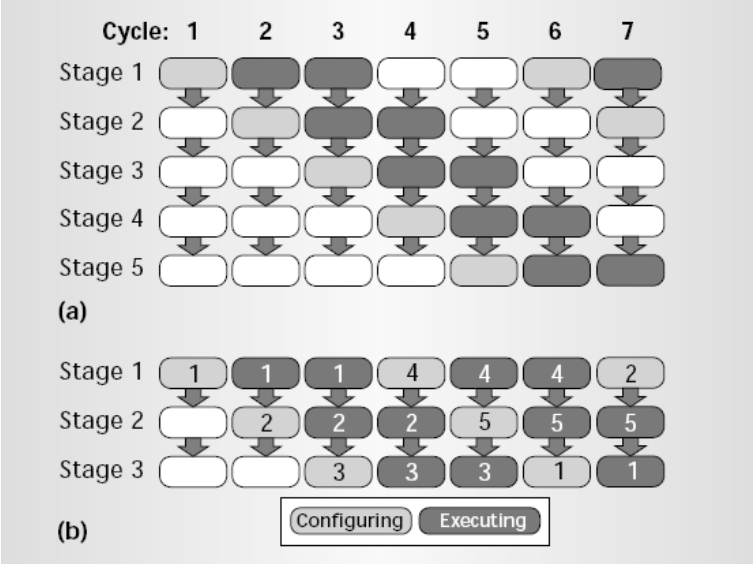


Figure 4. Virtualization process of Piperench (GOLDSTEIN, SCHMIT, *et al.*, 2000)

Concerned about the overheads created by the static reconfiguration process, Stitt (LYSECKY, STITT e VAHID, 2004) had a pioneering work on proposing the dynamic detection strategy to reconfigurable fabrics. The employment of dynamic detection techniques does not rely on code recompilation, providing software compatibility and maintaining the device time-to-market. Stitt et al. (LYSECKY, STITT e VAHID, 2004) presented the Warp Processing, which is based on a system that does dynamic partitioning using reconfigurable logic. Performance improvements are shown on applying such a technique to a set of popular embedded system benchmarks. It is composed of a microprocessor to execute the application software, another microprocessor where a simplified CAD algorithm runs, local memory and a dedicated simplified FPGA.

In (CLARK, KUDLUR, *et al.*, 2004) the Configurable Compute Array (CCA), which is a coarse-grained array tightly coupled to an ARM processor, is proposed. The feeding process of the CCA involves two steps: the discovery of which sub graphs are suitable for running on the CCA, and their replacement by microops in the instruction stream. Two alternative approaches are presented: static, where the sub graphs for the CCA are found at compile time, and dynamic. Dynamic discovering assumes the use of a trace cache to perform sub-graph discovery on the retiring instruction stream at run-time.

Even applying dynamic techniques Warp Processing and CCA present some drawbacks, though. First, significant memory resources are required for the kernels transformation. In the case of the Warp Processing, the use of an FPGA presents long latency and consumed area, being power inefficient. In the case of the CCA, some operations, such as memory accesses and shifts, are not supported at all. Then, usually

just the very critical parts of the software are optimized, limiting their field of application.

In (BECK, RUTZIG, *et al.*, 2008), Beck proposes a coupling of a reconfigurable system together with a special binary translation (BT) technique implemented in hardware, named Dynamic Instruction Merging (DIM). DIM is designed to detect and transform instruction groups for reconfigurable hardware execution. Therefore, this work proposes a complete dynamic nature of the reconfigurable array: besides being dynamic reconfigurable, the sequences of instructions to be executed on it are also detected and transformed to a data path's configuration at run-time.

As can be observed in Figure 5, this is done concurrently while the main processor fetches other instructions (Step 1). When a sequence of instructions is found, a binary translation is applied to it (Step 2). Thereafter, this configuration is saved in a special cache, and indexed by the memory address of the first detected instruction (Step 3).

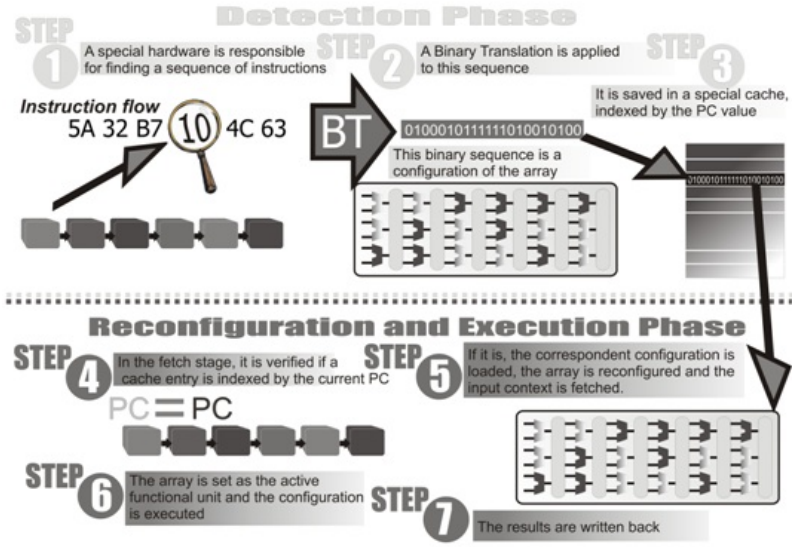


Figure 5. How the DIM system works (BECK, RUTZIG, *et al.*, 2008)

The next time the saved sequence is found (Step 4), the dependence analysis and the translation are no longer necessary: the BT mechanism loads the previously stored configuration from the special cache, the operands from the register file and memory (Step 5), and activates the reconfigurable hardware as functional unit (Step 6). Then, the array executes that configuration in hardware (including write back of the results) (Step 7), instead of ordinary (not translated) processor instructions. Finally, the PC is updated, in order to continue the execution. This way, repetitive dependence analysis for the same sequence of instructions throughout program execution is avoided.

The reconfigurable data path is tightly coupled to the processor, working as another ordinary functional unit in the pipeline. It is composed of coarse-grained functional units, as arithmetic and logic units and multipliers. A set of multiplexers are responsible for the routing. Because of the small context size and simple structure, the use of a coarse-grained data path is more suitable for this kind of dynamic technique. In this technique, both DIM engine and reconfigurable data path are designed to work in parallel to the processor and do not introduce any delay overhead or penalties for the critical path of the pipeline structure.

All the works explained in this subsection show the potential of transforming parts of the software to reconfigurable logic execution. Both dynamic and static approaches are still limited to optimize single-threaded applications, which narrow its field of application since, nowadays, due to the limited instruction level parallelism (MAK, 1991) the performance will not increase at the same pace as the number of functional units increases as well.

2.2 Multiprocessing Systems

In the nineties, sophisticated architectural features that exploit instruction level parallelism, like aggressive out-of-order instruction execution, provided higher increase in the overall circuit complexity than performance improvements. Therefore, as the technology reached an integration of almost a billion of transistors in a single die in this decade, researchers started to explore thread level parallelism by integrating many processors in a single die.

In the academic field, several researches address chip-multiprocessing subject. Hydra (HAMMOND, HUBBERT, *et al.*, 2000) was one of the pioneering designs that integrated many processors within a single die. The authors argue that the cost in hardware of extracting parallelism from a single-threaded application is becoming prohibitive, and advocate the use of software support to extract thread level parallelism to allow hardware to be simple and fast. In addition, they discourage a complex single processor implementation in a billion-transistor design, since the wire delay increases as the technology scaling that makes the handling of long wires complex in pipeline-based designs. For instance, in the Pentium 4 design, the long wires distance adds two pipeline stages to the floating-point pipeline, so the FPU has to wait two whole clock cycles for the operands to arrive from the register file (PATTERSON e HENNESSY, 2010).

The Hydra Chip Multiprocessor is composed of multiple copies of the same processors being homogeneous on both architecture and organization point of views. Hydra implementation contains eight processors being each of them capable of issuing two instructions per cycle. The choice for simple processor organization provides advantages over multiprocessing systems composed of complex processor since, besides allows a higher operating frequency of the chip, achieves larger number of processor in the same area. A performance comparison among Hydra design, 12-issue superscalar processor and 8-thread 12-issue simultaneous multithreading processor shown promising results for applications that could be parallelized into multiple threads, since Hydra uses relatively simple hardware than the compared architectures. However, disadvantages appear when applications contain code that cannot be multithreaded, Hydra is then slower than the compared architectures, because only one processor can be targeted to the task, and this processor does not have strong ability to extract instruction level parallelism.

Piranha (ANDRE, BARROSO, *et al.*, 2000), as Hydra, invests on the coupling of many simple single-issue in-order processors to massive explore thread level parallelism of commercial database and web server applications. The project makes available a complete platform composed of eight simple processor cores along with a complete cache hierarchy, memory controllers, coherence hardware, and network router all onto a single chip running at 500 MHz. Results around web server applications show that Piranha outperforms an aggressive out-of-order processor exploitation running at 1 GHz by over a factor of three times. As Hydra, the authors explicit declare that Piranha is a wrong design choice if the goal is to achieve performance improvements in applications

that have lack of sufficient thread-level parallelism due to the simple organization of their processors.

Tullsen (KUMAR, TULLSEN, *et al.*, 2004) demonstrates that there can be great advantage on providing a diversity of processing capabilities within a multiprocessing chip, allowing that architecture to adapt to the application requirements. A heterogeneous organization and homogeneous ISA multiprocessing chip is assembled with four different processors organization, each one with its particular power consumption and instruction level parallelism exploitation capability. To motivate the use of such an approach, a study over the SPEC2000 benchmark suite was done. It shows that applications have different execution phases and they require different amount of resources in these phases. On that account, several dynamic switching algorithms are employed to examine the limits of power and performance improvements possible in a heterogeneous multiprocessing organization environment. Huge energy reductions with little performance penalties are presented by only moving applications to a better-matched processor.

For almost ten years now, multiprocessing systems are increasingly getting the general-purpose processor marketplace. Intel and AMD have been using this approach to speed up their high-end processors. In 2006, Intel has shipped its multiprocessor chip based on homogeneous architecture strategy. Intel Core Duo is composed of two processing elements that make communication among themselves through an on-chip cache memory. In this project, Intel has thought beyond the benefits of such a system employment and created an approach to increase the process yield. A new processor market line, named Intel Core Solo, was created aiming to increase the process yield by selling even Core Duo dies with manufacturing defects. In this way, Intel Core Solo has the very same two-core die as the Core Duo, but only one core is defect free.

Recently, embedded processors are following the trend of high-end general-purpose processors coupling many processing elements, with the same architecture, on a single die. Early, due to the hard constraints of these designs and the few parallel applications that would benefit from several GPP, homogeneous multiprocessors were not suitable for this domain. However, the embedded software scenario is getting similar to a personal computer one due to the convergence of the applications to embedded device already discussed in the beginning of this work. ARM Cortex-A9 processor is the pioneer to employ homogeneous multiprocessing approach into embedded domain, coupling up to four Cortex-A9 cores into a single die. Each processing element uses powerful techniques for ILP exploration, as superscalar execution and SIMD instruction set extensions, which closes the gap between the embedded processor design and high-end general-purpose processors.

Texas Instrument strategy better illustrates the embedded domain trend to use multiprocessor systems. This heterogeneous architecture handles in hardware most widely used applications on embedded devices like multimedia and digital signal processing. In 2002, Texas Instruments has launched in the market an Innovator Development kit (IDK) targeting high performance and low power consumption for multimedia applications. IDK provides an easy design development, with open software, based on a customized hardware platform called open multimedia applications processor (OMAP). Since its launch, OMAP is a successful platform being used by the embedded market leaders like Nokia with its N90 cell phones series, Samsung OMNIA HD and Sony Ericsson IDOU. Currently, due to the large diversity found on the embedded consumer market, Texas Instruments has divided the OMAP family in two

different lines, covering different aspects. The high-end OMAP line supports the current sophisticated smart phones and powerful cell phone models, providing pre-integrated connectivity solutions for the latest technologies (3G, 4G, WLAN, Bluetooth and GPS), audio and video applications (WUXGA), including also high definition television. The low-end OMAP platforms cover down-market products providing older connectivity technologies (GSM/GPRS/EDGE) and low definition display (QVGA).

Recently, Texas Instrument released one of its latest high-end products. The OMAP4440 covers the connectivity besides high-quality video, image and audio support. This mobile platform came to supply the need of the increasingly multimedia applications convergence in a single embedded device. This platform incorporates the dual-core ARM Cortex A9 MPCore providing higher mobile general-purpose computing performance. The power management technique available in the ARM Cortex A9 MPCore balances the power consumption with the performance requirements, activating only the cores that are needed for a particular execution. In addition, due to the high performance requirement of today smart phones, up to eight threads can be concurrently fired in the MPCore, since each core is composed of four single-cores Cortex A9. The single-core ARM Cortex A9 implements superscalar execution, SIMD instruction set and DSP extensions, showing almost the same processing power as a personal computer into an embedded mobile platform. Excluding the ARM Cortex MPCore, the remainder processing elements are dedicated to multimedia execution.

In 2011, NVIDIA introduced the project named Kal-el (NVIDIA, 2011) mobile processor. This project is the first to encapsulate four processors in a single die for mobile computation. The main novelty introduced by this project is the Variable Symmetric Multiprocessing (vSMP) technology. vSMP introduces a fifth processor named “Companion Core” that executes tasks a low frequency for active standby mode, as mobile systems tend to keep in this mode for most time. All five processors are ARM Cortex-A9, but the companion core is built in a special low power silicon process. In addition, all cores can be enabled/disabled individually and when the active standby mode is on, only the “Companion Core” works, so battery life can significant improved. NVIDIA reports that the switching from the “Companion Core” to the regular cores are supported only by hardware and take less than 2 milliseconds being not perceptible to the end users. In comparison with Tegra 2 platform, vSMP achieves up to 61% of energy savings on running HD video playback.

As OMAP, Samsung designs are focused on multimedia-based development. Their projects are very similar due to the increasing market demand for powerful multimedia platforms, which stimulates the designer to take the same decision to achieve efficient multimedia execution. Commonly, the integration of specific accelerators is used, since this reduces the design time avoiding validation and testing time. In 2008, Samsung launched the most powerful of the Mobile MPSoC family. At first, S3C6410 was a multimedia MPSoC like OMAP4440. However, after its deployment in the Apple iPhone 3G employment, it has become one of the most popular MPSoCs, shipping 3 million units during the first life time month. After, Apple has developed iPhone 3GS, which assures better performance with lower power consumption. These benefits are supplied by the replacement of the S3C6410 architectures with the high-end S5PC100 version.

Following the multimedia-based multiprocessor trend, Samsung platforms are composed of several application specific accelerators building heterogeneous

multiprocessor architectures. S3C6410 and S5PC100 have a central general-purpose processing element, in both cases ARM-based, surrounded by several multimedia accelerators tightly targeted to DSP processing. Both platforms skeleton follow the same execution strategy, changing only the processing capability of their IP cores. Small platform changes are done from S3C6410 to S5PC100 aiming to increase the performance. More specifically, a 9-stage pipelined ARM 1176JZF-S core with SIMD extensions is replaced to a 13-stage superscalar-pipelined ARM Cortex A8 providing greater computation capability for general-purpose applications. Besides its double-sized L1 cache compared to ARM1176JZF-S, ARM Cortex A8 also includes a 256KB L2 cache avoiding external memory accesses due L1 cache misses. NEON ARM technology is included in ARM Cortex A8 to provide flexible and powerful acceleration for intensive multimedia applications. Its SIMD based-execution accelerates multimedia and signal-processing algorithms such as video encode/decode, 2D/3D graphics, speech-processing, image processing at least twice better than the previous SIMD technology. However, these hardware changes provide mandatory tool chain modifications to support the use of the new dedicated hardware, which consequently breaks the binary compatibility since the software developers must change and recompile the application code.

Regarding multimedia accelerators, both systems are able to provide suitable performance for any high-end mobile devices. However, S5PC100 includes the latest codec multimedia support using powerful accelerators. This strategy on changing some platform elements from S3C6410 to S5PC100 illustrates the growth and maturity phase of the functionality lifecycle discussed in the beginning of this work. In this phase, the electronic consumer market already has absorbed these functionalities, and their hard-wired execution is mandatory for energy and performance efficiency.

Other multiprocessing systems have already been released in the market, with different goal from the architectures discussed before. Sony, IBM and Toshiba have worked together to design the Cell Broadband Engine Architecture (CHEN, RAGHAVAN, *et al.*, 2007). The Cell architecture combines a powerful central processor with eight SIMD-based processing elements. Aiming to accelerate a large range of application behaviors, the IBM PowerPC architecture is used as general purpose processor. In addition, this processor has the responsibility to manage the processing elements surrounding it. These processing elements, called synergistic processing elements (SPE), are built to support streaming applications with SIMD execution. Each SPE has a local memory that only can be accessed by explicit and particular software directives. These facts make the software development for the Cell processor even more difficult, since the software team should be aware of this local memory, and manage it at the software level to better explore the SPE execution. Despite its high processing capability, the Cell processor does not yet have a large market acceptance because of the intrinsic difficulty to code software in order to use the SPEs. When it was launched, the Playstation console did not achieve a great part of the gaming entertainment marketplace, the game developers had not enough knowledge of the tool chain libraries to efficiently explore the complex Cell architecture, which implied in a restricted amount of games available in the market.

Homogeneous multiprocessing system organization is also explored in the market, mainly for personal computers with general purpose processors, because of the huge amount of different applications that these processors have to face, and hence due to the difficult task to define specialized hardware accelerators. In 2005, Sun Microsystems

announced its first homogeneous multiprocessor design, composed of up to 8 processing elements executing the SPARC V9 instruction set. UltraSparc T1, also called Niagara (JOHNSON e NAWATHE, 2007), is the first multithreaded homogeneous multiprocessor, and each processing element is able to execute four threads concurrently. In this way, Niagara can handle, at the same time, up to 32 threads. Recently, with the deployment of UltraSparc T2, this number has grown to 64 concurrent threads. Niagara family targets massive data computation with distributed tasks, like the market for web servers, database servers and network file systems.

Intel has announced its first multiprocessing system based on homogeneous organization prototyped with 80-cores, which is capable of executing 1 trillion floating-point operations per second, while consuming 62 Watts (VANGAL, HOWARD, *et al.*, 2007). The company expects to launch this chip within the next 5 years in the market. Hence, the x86 instruction set architecture era could be broken, since their processing elements is based on the very long instruction word (VLIW) approach, letting to the compiler the responsibility for the parallelism exploration. The interconnection mechanism used on the 80-core uses a mesh network to communicate among its processing elements. However, even employing the mesh communication turns out to be difficult, due to the great amount of processing elements. In this way, this ambitious project uses a 20 Mbytes stacked on-chip SRAM memory to improve the processing elements communication bandwidth.

Graphic processing unit (GPU) is another multiprocessing system approach aiming at graphic-based software acceleration. However, this approach has been arising as a promise architecture also to improve general-purpose software. Intel Larrabee (SEILER, CARMEAN, *et al.*, 2008) attacks both applications domain thanks to its CPU- and GPU- like architecture. In this project Intel has employed the assumption of energy efficiency by simple cores replication. Larrabee uses several P54C-based cores to explore general-purpose applications. In 1994, P54C was shipped in CMOS 0.6um technology reaching up to 100 MHz and does not include out-of-order superscalar execution. However, some modifications have been done in the P54C architecture, like supporting of SIMD execution aiming to provide more powerful graphic-based software execution. The SIMD Larrabee execution is similar to, but powerful than, the SSE technology available in the modern x86 processors. Each P54C is coupled to a 512-bit vector pipeline unit (VPU), capable of executing, in one processor cycle, 16 single precision floating-point operations. In addition, Larrabee employs a fixed-function graphics hardware that performs texture-sampling tasks like anisotropic filtering and texture decompression. However, in 2009, Intel discontinued Larrabee project.

NVIDIA Tesla (LINDHOLM, NICKOLLS, *et al.*, 2008) is another example of multiprocessing system based on the concept of a general-purpose graphic processor unit. Its massive-parallel computing architecture provides support to Compute Unified Device Architecture (CUDA) technology. CUDA, the NVIDIA´s computing engine, eases the parallel software development process by providing software extensions in its framework. In addition, CUDA provides permission to access the native instruction set and memory of the processing elements, turning the NVIDIA Tesla to a CPU-like architecture. Tesla architecture incorporates up to four multithreaded cores that communicate through a GDDR3 bus, which provides a huge data communication bandwidth.

Table 1. Summarized Commercial Multiprocessing Systems

	Architecture	Organization	Cores	Multithreaded Cores	Interconnection
OMAP4440	Heterogeneous	Heterogeneous	2 ARM Cortex A9 1 PowerVR graphics accelerator 1 Image Signal Processor	No	Integrated Bus
Samsung S3C6410/S5PC100	Heterogeneous	Heterogeneous	1 ARM1176JZF-S 5 Multimedia Accelerators	No	Integrated Bus
Cell	Heterogeneous	Heterogeneous	1 PowerPC 8 SPE	No	Integrated Bus
Niagara	Homogeneous	Homogeneous	8 SPARC V9 ISA	Yes (4 threads)	Crossbar
Intel 80-Cores	Homogeneous	Homogeneous	80 VLIW	No	Mesh
Intel Larrabee	Homogeneous	Homogeneous	n P54C x86 cores SIMD execution	No	Integrated Bus
NVIDIA Tesla (GeForce8800)	Homogeneous	Homogeneous	128 Stream Processors	Yes (up to 768 threads)	Network

As discussed in the beginning of this work, multiprocessing systems employment is a consensus to current/next generation for both general and embedded processors, since aggressive exploration of instruction level parallelism of single-threaded applications does not provide an advantageous tradeoff between extra transistor usage and performance improvement. All multiprocessing system designs mentioned in this section somehow explore thread level parallelism. Summarizing all commercial multiprocessing system discussed before, Table 1 compares their main characteristics showing their differences depending on the target market domain. Heterogeneous architectures, like the OMAP, Samsung and Cell, incorporate several specialized processing elements to attack specific applications for highly constrained mobile or portable devices. These architectures have multimedia-based processing elements, following the trend of embedded systems. However, as mentioned before, software productivity is affected when such strategy is used, each new platform launching implies on tool chain modifications, like library description, to explore the execution of the coupled specialized hardware. In addition, this approach can be optimized for performance and area, but they are costly to design and not programmable, making upgradability a difficult task and they bring no benefit excluding the targeted applications.

Unlike heterogeneous architectures, homogeneous ones aim at the general-purpose processing market, handling a wide range of applications behavior by replicating general-purpose processors. Commercial homogeneous architectures still use only homogeneous organizations, coupling several processing elements with the same ISA and the processing capability. Heterogeneous organizations have not been used on homogeneous architectures, since power management techniques, like DVFS, support the variable processing capability. However, most of these techniques are restricted to reduce only dynamic power, the circuit still consumes leakage power that is increasing with the technology scaling. Supposing a perfect power management that solves dynamic and leakage power, the homogeneous architecture and organization platform still relies on huge area overhead, what supports the need for homogeneous architecture and heterogeneous organization strategy.

2.3 Multi-Threaded Reconfigurable Systems

As the scope of this work is motivated by multiprocessing systems that use some kind of adaptability on exploiting instruction level parallelism, this sub-section only

contemplates the state of the art researches that employ multiprocessing systems together with reconfigurable architectures.

In (KOENIG, BAUER, *et al.*, 2010), the authors propose KAHRISMA, a heterogeneous organization and architecture platform. Figure 6 shows KAHRISMA's architecture overview, its multiple instruction set (RISC, 2- and 6-issue VLIW, and EPIC) coupling with fine- and coarse-grained reconfigurable encapsulated data path elements (EDPE) are the main novelty of this research. The resource allocation task is totally supported by a flexible software framework that, at compile time, analyzes the high-level C/C++ source code and builds an internal code representation. This code representation goes through an optimization process to eliminate dead code and constant propagation. After, the internal representation is used to identify/select parts of code that will implement custom instructions (CIs) to be executed in the reconfigurable arrays (FG- and CG-EDPE).

The entire process considers that the amount of free hardware resources can vary at run time, since some parts of code could present greater number of parallel executing threads than others, so multiple implementations of custom instructions are provided. The runtime system is responsible for the best CI's solution selection, which depends on the loading state of the architecture. Thus, the execution of a certain part of code can vary from RISC implementation (low performance) to the custom instruction implementation using FG- as well as CG-EDPEs (high performance). Speedups are shown in the execution of very intensive compute kernel from h.264 video encode-decode standard on exploring multiple ISAs, when the multithread scenario is considered.

However, this approach fails at several crucial constraints of the embedded systems. High memory usage is caused by multiple assembly generation of the same part of code, which could not always offer speedups due to the restricted amount of hardware resources at a certain time. KAHRISMA is able to optimize multi-threaded applications, however they also rely on compiler support, static profiling and a tool to associate the code or custom instructions to the different hardware components at design time. Despite inserting reconfigurable components in its platform, KAHRISMA maintains the main drawbacks of the current embedded multiprocessing systems (e.g. OMAP), since it maintains a mandatory time overhead on each platform change to produce custom instructions affecting the software productivity by breaking the binary compatibility.

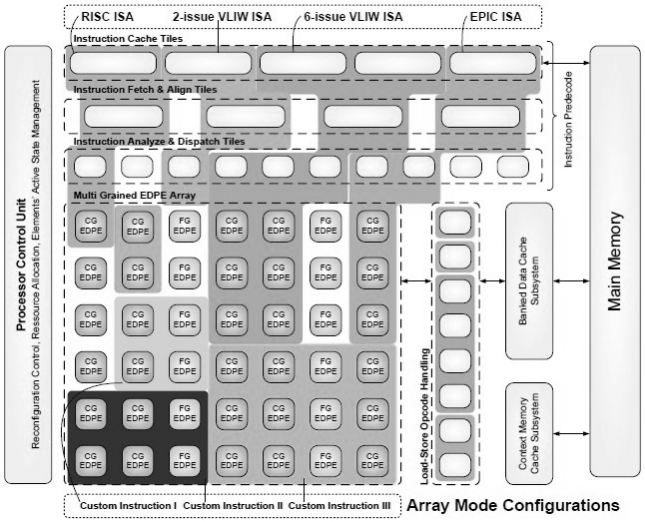


Figure 6. KAHRISMA architecture overview (KOENIG, BAUER, *et al.*, 2010)

Considering a system with homogeneous architecture and heterogeneous organization, one can find the Thread Warping (TW) (STITT e VAHID, 2007), which extends the aforementioned Warp Processing system shown in the Section 2.1. Prior work has developed a CAD algorithm that dynamically remaps critical code regions of single-threaded applications from processor instructions to FPGA circuits using a runtime synthesis. The contribution of TW consists of integrating existing CAD algorithm in a framework capable of dynamically synthesizing many thread accelerators. Figure 7 overviews the TW architecture and shows how the acceleration process occurs. As can be seen, the TW is composed of four ARM11 microprocessors, a Xilinx Virtex IV FPGA and an On-Chip CAD hardware used to the synthesizing process.

The thread creation process shown in the Step 1 of the Figure 7 is totally supported by an Application Programming Interface (API), so no source code modification is needed. However, changes in the operating system are mandatory to support the scheduling process. The operating system scheduler maintains a queue that stores the threads ready for execution. In addition, a structure, named schedulable resource list (SRL), holds the list of free resources. Thus, to trigger an execution of a thread, the operating system should check if the resource requirements of a certain ready thread match with the free resources in the SRL. An ARM11 is totally dedicated to run the operating system tasks needed to synchronize threads and to schedule their kernels in the FPGA (Step 2 of Figure 7).

The framework, implemented in hardware, analyzes waiting threads, and utilizes on-chip CAD tools to create custom accelerator circuits for executing in the FPGA (step 3). After some time, on average 22 minutes, the CAD tool finishes mapping the accelerators onto the FPGA and stores the custom accelerators circuits in a non-volatile library for future executions, named AccLib in the Figure 7. Assuming that the application has not finished during these 22 minutes, the operating system (OS) begins scheduling threads onto both FPGA accelerators and microprocessor cores (step 4). Since the area requirements of the existing accelerators could exceed the FPGA capacity, a greedy knapsack heuristic is used to generate a solution for the instantiation process of the accelerators in the FPGA.

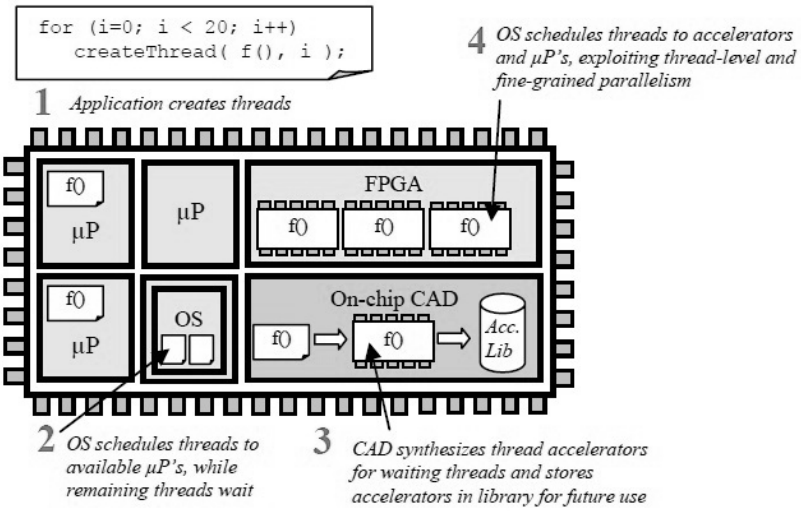


Figure 7. Overview of Thread Warping execution process (STITT e VAHID, 2007)

Despite its dynamic nature, that provides binary compatibility, there are several drawbacks in the Thread Warping proposal. First, the unacceptable latency on creating the CAD tool for applications those run less than 22 minutes. TW shows good speedups (502 times) when the initial execution of the applications is not considered. In other words, these results does not consider the period when the CAD tool is working to create the custom accelerator circuits. In the case of the custom instructions creation overhead is taken into account, all but one of the ten algorithms have shown performance loss. Summarizing, Thread Warping presents the same deficiency of the original work shown in the Section 2.1: only critical code regions are optimized, due to the high overhead in time and memory imposed by the dynamic detection hardware. Thus, TW only optimizes applications with few and very defined kernels, which narrows its field of application. The optimization of few kernels will very likely not satisfy the performance requirements of future embedded systems, where it is foreseen a high concentration of different software behaviors (SEMICONDUCTORS, 2009).

In (YAN, WU, *et al.*, 2010), Yan proposes the coupling of many reconfigurable processing units based on FPGA to SparcV9 general-purpose processors. ISA extensions are done to support the reconfigurable processing units' execution. However, the system can also work without using the accelerators in a backward-compatible manner. The reconfigurable architecture overview is show in the Figure 8. As it can be seen, a crossbar is employed to connect the reconfigurable processing units to the homogeneous SparcV9-based processors, which provides a low-latency parallel communication.

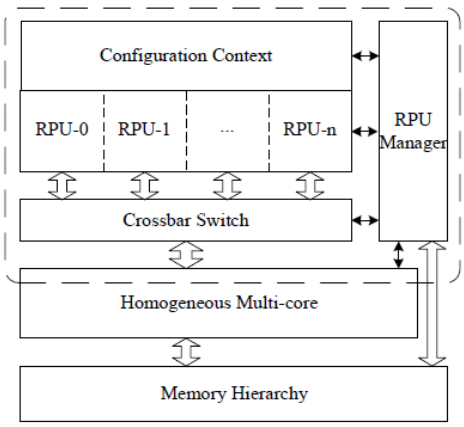


Figure 8. Blocks of the Reconfigurable Architecture (YAN, WU, *et al.*, 2010)

The Reconfigurable Processing Unit (RPU) is a data driven computing system, based on fine-grained reconfigurable logic structure similar to Xilinx Virtex-5. The RPU is composed of configurable logic block arrays to synthesize the logic; local buffer responsible for the communication between the RPU and the SparcV9 processors; configuration context that stores the already implemented custom instructions; and configuration selection multiplexer that selects the fetched custom instructions from the configuration context. As Thread Warping, this approach also employs an extra circuit to provide consistency and synchronization on data memory accesses.

A software-hardware co-operative implementation is used to support the triggering of the reconfigurable executions. The execution is divided in four phases: configuring, pre-load, processing and post-store phase. The configuring phase starts when a special instruction, that request an RPU execution, arrives in the execution stage of the SparcV9 processor. If the custom instruction is available at the configuration context, the pre-load phase starts and an interruption is generated to notify the operating system scheduling to configure the RPU with the configuration context. In this phase, the data required for the computation also are loaded to the local buffer of the respective RPU. In the processing phase the data driven computing is done. Finally, some special instructions are fired to fetch the results from the local buffer and to return the execution process to the SparcV9 processor.

This approach improves the performance over the software only execution by, on average, 2.4 times in an application environment composed of an encryption standard and an encode image algorithm. However, some implementation aspects make such an approach not viable to embedded domain, the binary compatibility is broken since a compilation phase is used to extend the original SparcV9 instruction set to support the RPU execution. The fine-grained reconfigurable structure relies on high reconfiguration overhead, which narrows the scope of such an approach to applications where very few kernels cover almost its whole execution time.

Different from other approaches, in (SMIT, 2008) is presented a multiprocessing reconfigurable architecture focused on accelerating streaming DSP applications. The authors argue that is easier to control the reconfigurable architecture when handling such kind of applications since most of them can be specified as a data flow graph with streams of data items (the edges) flowing between computation kernels (the nodes). Annabelle SoC is presented in the Figure 9, its heterogeneous architecture and organization aggregates a traditional ARM926 that is surrounded by ASIC blocks (e.g.

Viterbi Decoder and DDC) and four-domain specific coarse-grained reconfigurable data path, named Montium cores. A network-on-chip infrastructure supports inter-Montium communication with higher bandwidth and multiple concurrent transmissions. The communication among the rest of the system elements is done through a 5-layer AMBA bus. As each processor operates independently, they need to be controlled separately, so the ARM926 processor controls the other cores by sending configuration messages to their network interface. Since the cores might not be running at the same clock speed as the NoC, the network interface synchronizes the data transfers.

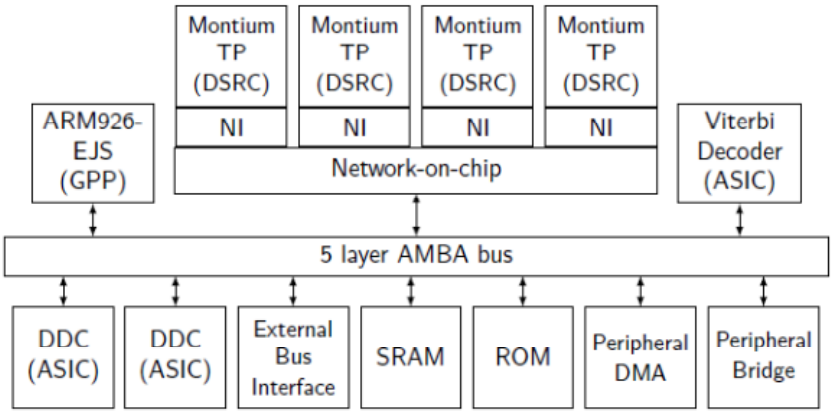


Figure 9. Block Diagram of Annabelle SoC (SMIT, 2008)

Figure 10 depicts the architecture of a single Montium Core that has five 16-bit width arithmetic and logic units interconnected by 10 local memories due to the high bandwidth required for DSP applications. An interesting point considered in this work is the locality of reference. In other words, the accesses on small and local memory is much more energy efficient than accessing a big and far distant memory because of increasing wire capacitance on recent nano-technologies. There is a communication and a configuration unit that provides the functionality to configure the Montium, to manage the memories by means of direct memory access (DMA) and to start/wait/reset the computation of the algorithm configured. Since the Montium core is based on a coarse-grained reconfigurable architecture, the configuration memory is relatively small, on average, it occupies only 2.6 Kbytes. Because the configuration memory can be accessed as a RAM memory, the system allows dynamic partial reconfiguration. Results show that energy savings can be achieved by only exploiting locality of reference. In addition, this work supports the use of coarse-grained reconfigurable architectures by demonstrating lower reconfiguration time overhead. Despite the fact that Annabelle explores a reconfigurable fabric to accelerate streaming applications, this system still relies on heterogeneous ISA implementation by coupling ASICs to provide efficient energy-performance execution. Like OMAP, such an approach affects software productivity since each new platform requires tool chain modifications.

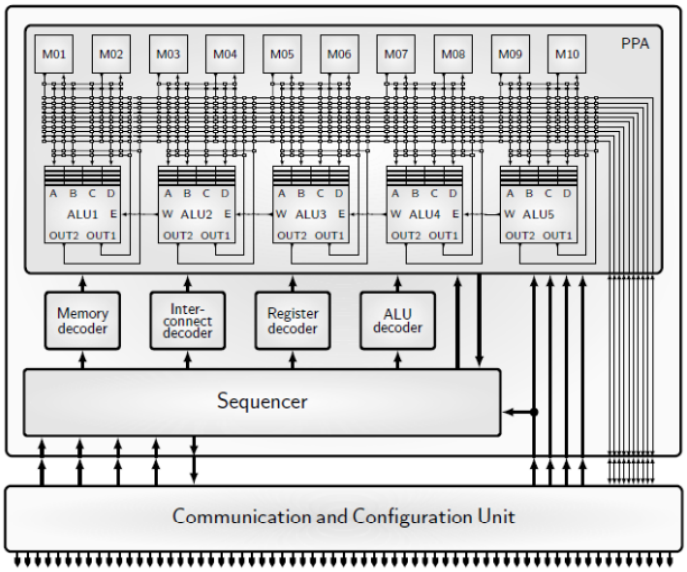


Figure 10. The Montium Core Architecture

Studies on sharing reconfigurable fabric among general-purpose processors are shown in (WATKINS, CIANCHETTI e ALBONESI, 2008) (GARCIA e COMPTON, 2008). These strategies are supported by the huge area overhead and non-concurrent utilization of the reconfigurable units by multiple processors. In (GARCIA e COMPTON, 2008), a reconfigurable fabric sharing approach focused on accelerating multithreaded applications is presented. This work exploits a type of parallelism named single program multiple data (SPMD), where each thread instantiation runs the same set of operations on different data. Multiple instantiations of the Xvid encoder are used to emulate such type of parallelism, acting as a digital video recorder to encode multiple video streams from different channels simultaneously. To avoid low utilization of reconfigurable hardware kernels, different threads share the already configured reconfigurable hardware kernels. For example, if two instances of Xvid are executing, a single physical copy of each reconfigurable hardware kernel could be shared, so both instances of Xvid can benefit from them. Although it does not specify any particular reconfigurable hardware design, the Xvid encoder instantiations are synthesized in a Xilinx Virtex-4 FPGA.

First experiments show that sharing single physical copy of each reconfigurable hardware kernel among all Xvid instances performs very poorly due to the frequent contention on accessing the kernels. Thus, the authors conclude that not all kernels can be effectively shared, so they created a modified strategy to provide better kernels allocation. Such an approach uses the concept of virtual kernels to control the physical kernel allocation. The algorithm uses the following strategy. When an application attempts to access a virtual kernel, the controller first checks if any instance of the corresponding virtual kernel is already mapped to a physical kernel and if any other physical kernel is free. If multiple physical kernels are available, one of them will be reserved to execute the virtual kernel even if other physical kernel already is executing the same virtual kernel. This strategy eliminates the waiting for busy shared physical kernel increasing the combined throughput of Xvid encoder in a multiprocessor system by 95-130% over the software execution alone.

Watkins (WATKINS, CIANCHETTI e ALBONESI, 2008) proposes, as a first work, a shared specialized programmable logic (SPL) to decrease the large power and area costs of FPGA when a multiprocessing environment is considered. The main motivation to apply such an approach in multiprocessing systems is supported by intermittent used of the reconfigurable fabric. There are inevitably periods where one fabric is highly utilized while another lies largely or even completely idle. The motivation is produced through interesting experiments that show the poor utilization of the SPL's rows on running applications of different domains in multiprocessing system composed of eight cores. These data are depicted in Figure 11. The leftmost bars for the individual benchmarks show the utilization of the fabric composed of 26-row configuration, which reflects twice the area of each core that the SPL is coupled. The utilization of seven SPL fabrics is less than 10%, and the average SPL utilization is only 7%.

As can be seen in Figure 11, reducing each SPL to 12 rows (roughly the same area of the coupled core) increases SPL utilization for some benchmarks and greatly reduces the area occupied. However, this comes at a high cost: an 18% overall performance loss, since all benchmarks use more than 12 rows. The two rightmost bars of Figure 11 show a spatially shared SPL organization with a naive control policy that equally divides the rows of the SPL among all cores at all times. Thus, a SPL fabric configuration composed of 24 rows shared among four cores (fourth bar of AvgUtilization in Figure 11) produces, on average, an utilization improvement of the fabrics, delivers the same performance of 26-row private configuration and still reduces the area and peak power cost by over four times.

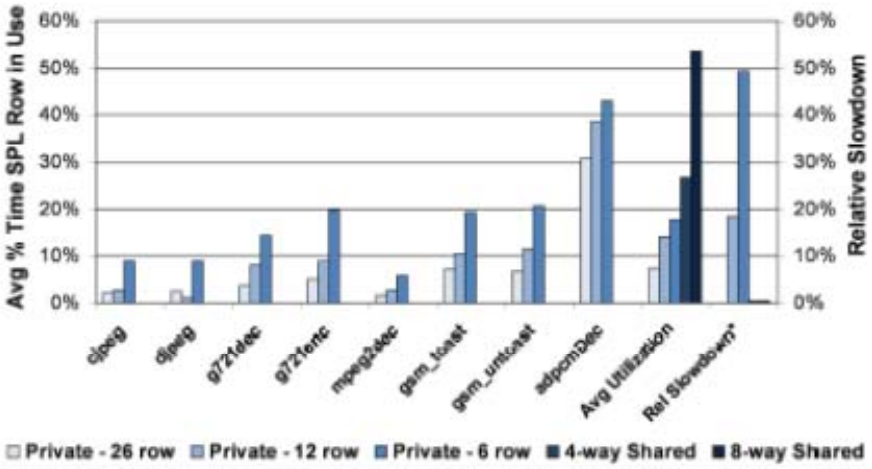


Figure 11. Fabric utilization considering many architecture organizations (WATKINS, CIANCHETTI e ALBONESI, 2008)

The fine-grained reconfigurable cell of such an approach is shown in Figure 12 (a). The SPL fabric is tightly integrated to the processor working as an additional functional unit. The main components of a SPL cell are: a 4-input look-up table (4-LUT), a set of two 2-LUTs plus a fast carry chain to compute carry bits or other logic functions if carry calculation is not needed, barrel shifters to align data as necessary, flip-flops to store results of computations, and an interconnect network between each row. These b-bit cells are arranged in a row to form a $c \times b$-bit row as shown in Figure 12 (b). Each cell in a row can perform a different operation and a number of these rows are grouped together to execute an application function. Each row completes the operation in a single SPL clock cycle.

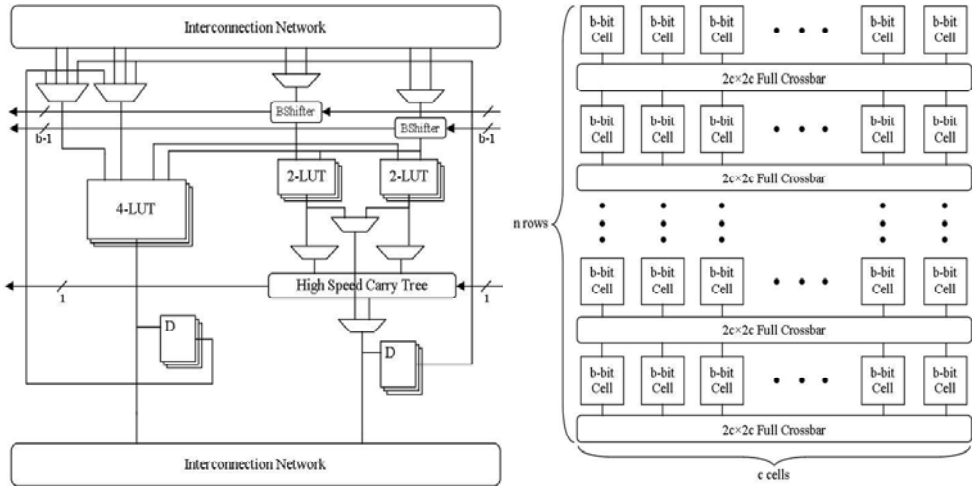


Figure 12. (a) SPL cell architecture (b) Interconnection strategy (WATKINS, CIANCHETTI e ALBONESI, 2008)

As explained before, each SPL row is dynamically shared among the cores. Two sharing strategies are used: spatial, where the shared fabric is physically partitioned among multiple cores (Figure 13(a)); or temporal, where the fabric is shared in a time multiplexed manner (Figure 13(b)). The spatial and temporal control policies bind the cores to particular SPL partitions, or pipeline time slots, based on runtime statistics. ISA extensions support the proposed sharing strategy.

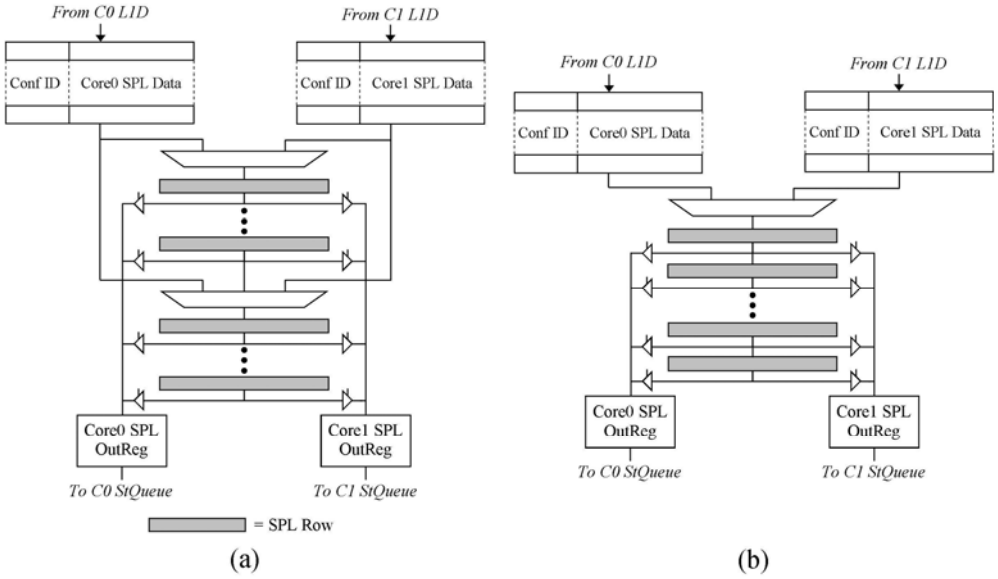


Figure 13. (a) Spatial sharing (b) Temporal sharing (WATKINS, CIANCHETTI e ALBONESI, 2008)

The grain of the sharing is a challenge that arises when the spatial approach is considered. The finest grain (a row) requires a large number of intermediate multiplexers, but provides the highest flexibility on sharing allocation mechanism since each one can vary the number of rows by the finest grain. The authors argue, after an investigation, that splitting the fabric in power of two, one can achieve good

flexibility/utilization tradeoff. If, for instance, there are 9-16 sharers, the SPL will be split into sixteen partitions. The authors propose a merging of SPL partitions policy that is based on idle cycle counter and an idle count threshold value, which in the current implementation is 1000. For temporal sharing, the SPL scheduling strategy uses a cycle-by-cycle round-robin algorithm to allocate the SPL fabric among the cores.

The coupling of a single private 26-row SPL on a single in-order core shows interesting speedups on running a mixed application workload, demonstrating that the adaptability provided by the reconfigurable architecture is suitable for single-threaded applications. In addition, a CMP environment composed of eight copies of one-way out-of-order cores and 26-row private SPL outperforms the 8-cores 4-way out-of-order chip multiprocessor. In addition, the latter consumes far more area and power than the former setup.

When the spatial sharing policy is applied, 26-row shared SPL outperforms 6-row private SPL in most of the benchmarks, reducing energy-delay product by up to 33% with little performance degradation. Particularly, on running the *crypt* application, which requires a large number of rows, precisely 298 rows, the spatial sharing approach presents 100% of performance slowdown and a larger energy-delay penalty. The authors report that the temporal sharing outperforms spatial sharing for two reasons: all benchmarks but *crypt* need a maximum of 26 rows for all functions, making temporal sharing more suitable than spatial sharing policy, since there are no significant periods where the benchmarks make concurrent accesses to the SPL.

In (ALBONESI e WATKINS, 2010) the previous explained work is extended, proposing hardware based fine-grained inter-core communication and barrier synchronization. Now, the entire system is named Reconfigurable Multicore Architecture for Parallel Processing (ReMAPP). The inter-core communication is established by the use of queues. The producing thread places data into the queue and the consuming thread reads data from the queue. Figure 14 summarizes the inter-thread communication. In the first step (Figure 14(a)), the producing thread place data into its input queue. Once all necessary data is loaded (Figure 14(b)), the consuming thread starts the execution in the SPL. As such an example, since the execution of the function does not occupy all SPL rows, the results are bypassed to the output queue of the consuming thread (Figure 14(c)). Finally, the consuming thread fetches data from the queue and stores it in the memory. A special table is used to maintain a mapping of the threads that stores, for each computation, its correspondent destination core.

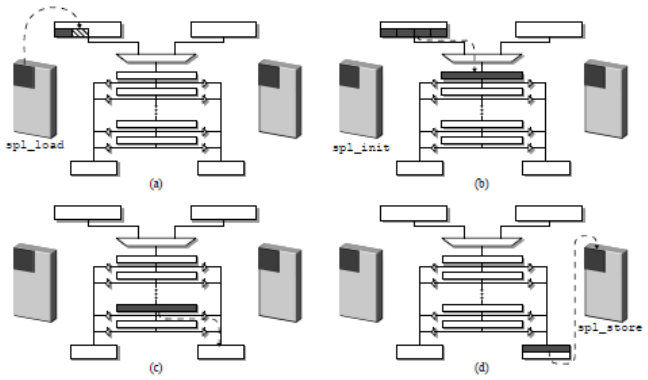


Figure 14. Thread Intercommunication steps (ALBONESI e WATKINS, 2010)

The barrier synchronization mechanism is also based on tables. To determine that all threads have arrived at the barrier, each SPL maintains a table that contains information related to each activated barrier. Each table contains as many entries as cores attached to ReMAPP, as each thread could be participating of different barriers. The table keeps track of the total number of threads, the number of arrived threads, and the number of cores that are participating of such part of code execution. Special instructions, named SPL barrier instructions, are implemented to provide the synchronization. Thus, SPL barrier instructions must not be issued to the fabric until all participating cores have arrived at the respective barrier. To achieve this, all participating threads compare the number of arrived thread information with the participating cores information, when these numbers become equals means that all participants arrived in the correspondent barrier and the execution can be kept on.

When compared to the single threaded SPL implementation, the SPL computation and communication mechanism using two threads improves the performance by 2 times and still provides better energy-delay product. In addition, performing barriers via ReMAPP significantly improves performance over software barriers by 9%, while achieving up to 62% better energy-delay product.

Summarizing, there are several works exploring the adaptability provided by the reconfigurable fabric on accelerating multithreaded applications. However, their implementations bring particular aspects that affect, in some way, the development process of the embedded system. These aspects are the following:

- Despite its heterogeneous architecture fashion that accelerates multithreaded applications, KAHRISMA relies on special tools to generate the binary code, which breaks the binary compatibility and affects the software productivity when platform changes are needed.
- Despite its dynamic nature on detect/accelerate parts of the application code, Thread Warping relies on an unacceptable latency to perform this task, which restricts its employment to optimize applications with few and very defined kernels.
- Despite good speedups shown on applying the strategy proposed in (YAN, WU, *et al.*, 2010), such implementation breaks the binary compatibility and affects the software productivity when platform changes are needed. Moreover, such an approach relies on high reconfiguration overhead, which makes it feasible only to accelerate applications with few kernels.
- Despite Annabelle demonstrates lower reconfiguration time, this work explores a reconfigurable fabric to accelerate only streaming applications and still relies on heterogeneous ISA implementation by coupling ASICs to provide efficient energy-performance execution. Like the commercial strategies such as OMAP, this approach affects software productivity since each new platform forces tool chain modifications.
- Despite its great area saving with the employment of a shared reconfigurable fabric strategy, ReMAPP relies on compiler support, static profiling and a tool to associate the code or custom instructions to the different hardware components at design time, not maintaining binary compatibility and affecting software productivity.

2.4 The Proposed Approach

In this work, we address the particular drawbacks of the aforementioned approaches by creating Custom Reconfigurable Arrays for Multiprocessor System (CReAMS) that:

- Unlike all strategies presented in the Section 2.1, explores the adaptability of reconfigurable system to achieve performance in a multithreaded environment. Kal-el project provides dynamic changing on performance when switches from "Companion Core" to Regular Core occur. However, this work produces neither ILP nor TLP adaptability since all cores have the same architecture and organization.
- Unlike (KOENIG, BAUER, *et al.*, 2010) (SMIT, 2008), builds a homogeneous platform on both architecture and organization nature, which eases the software development process since a unique tool chain is provided for any new version launched. Neither source code modifications nor new library learning process is necessary to explore the processing capabilities of the newest inserted processing elements.
- Unlike (KOENIG, BAUER, *et al.*, 2010) (YAN, WU, *et al.*, 2010) (SMIT, 2008) (ALBONESI e WATKINS, 2010), does not rely in special and particular tool chain to extract thread-level parallelism and to prepare the platform for execution. Our approach employs well-known application programming interfaces (e.g. OpenMP), which, despite automatically extracting TLP in a friendly interface, are coupled to the most commercial or academic compilers (e.g. gcc and icc), what makes the software development and the binary generation process easier than the aforementioned approaches.
- Unlike (ALBONESI e WATKINS, 2010) (STITT e VAHID, 2007) (YAN, WU, *et al.*, 2010), instead of exploring the flexibility of fine-grained architectures, employs a coarse-grained reconfigurable fabric that reduces the reconfiguration time and memory footprint due to the low context overhead. This grain choice increases the field of applications, since it opens room to accelerate the entire application code. Fine-grained architectures provide high acceleration levels but its scope is narrowed to applications that have few kernels responsible for a large part of the execution time.
- Unlike (WATKINS, CIANCHETTI e ALBONESI, 2008) (ALBONESI e WATKINS, 2010), instead of employing complex hardware design to share reconfigurable fabric among several processors to reduce area and power costs, proposes an heterogeneous organization platform, which can also achieve the same area savings and power consumption of the sharing policies. However, as will be shown, there are some applications that relies on a thread allocation strategy to achieve the same performance than the homogeneous organization, since there have been the best matching between the performance requirements of a certain thread and the different processing capabilities of the available processors.

Table 2 summarizes the characteristics of the single and multi threaded reconfigurable architectures considering the requirements of the current embedded system designs. CReAMS provides benefits in all characteristics, its energy consumption will be explored in the rest of this work.

Table 2. Characteristics of Multi and Single Threaded Reconfigurable Architectures

			Embedded Systems Requirements			
			SW Behavior Coverage	**SW Productivity**	**Design Time**	**Energy Consumption**
Single Threaded	Static	Piperench	X	X	X	V
		Chimaera	X	X	X	V
		GARP	X	X	X	V
		TRIPS	X	X	X	V
		Wavescalar	X	X	X	V
		Molen	X	X	X	V
	Dynamic	CCA	X	V	V	V
		Warp	X	V	V	V
		DIM	X	V	V	V
Multi Threaded	Static	KAHRISMA	V	X	X	V
		Annabelle	V	X	X	V
		ReMAPP	V	X	X	V
	Dynamic	Thread Warping	V	X	X	V
		CReAMS	V	V	V	?

3 ANALYTICAL MODEL

In this sub-section, we figure out the potential of single parallelism exploitation by modeling a multiprocessing architecture (*MP-Multi-Processor*). The considered architecture is composed of many simple and homogeneous cores without any capability to explore instruction level parallelism (ILP). This way we can elucidate the advantages of thread level parallelism (TLP) exploitation. We also compare its execution time (ET) to a high-end single processor (*SHE – Single High-End*) model, which is able to exploit only the ILP available in applications. First, we consider different amounts of fine- (instruction) and coarse- (thread) level parallelism available in the application code without any latency of the interconnection infrastructure modeled. This approach aims at investigating the performance potentials of both the aforementioned architectures. After, we create a latency modeling of a Network-on-Chip to verify the impact of inter-thread communication over the multiprocessing systems.

Considering a portion of a certain application code, we classify it in four different ways:

- α – the instructions that can be executed in parallel in a single processor;
- β – the instructions that cannot be executed in parallel in a single processor;
- δ – the amount of instructions that can be distributed among the processors of the multiprocessor environment.
- γ - the amount of instructions that cannot be split, and, therefore, must be executed in one of the processors among those in the multiprocessor environment.

Figure 15 exemplifies how the previously stated classification, considering a certain application “A”, would be applied. In the example shown, when the application is executed in the multiprocessor system (Figure 15(a)), 70% of the application code can run in parallel at some degree (i.e., divided in threads) and executed on different cores at the same time, so $\delta = 0.7$ and $\gamma = 0.3$. On the other hand, when the very same application A is executed on the high-end single-processor (Figure 15(b)), 64% of the instructions can be executed in parallel at some degree, so $\alpha = 0.64$ and $\beta = 0.36$.

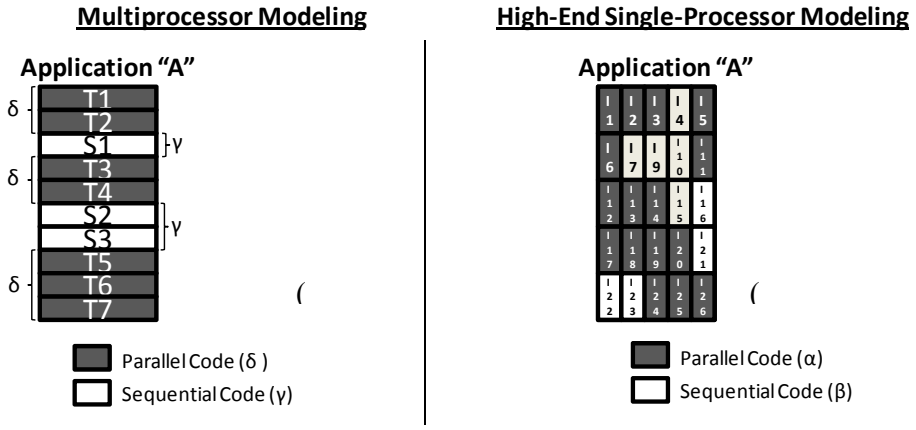


Figure 15. Modeling of the (a) Multiprocessor System and the (b) High-End Single-Processor

3.1 Performance Comparison

Let us start with the basic equation relating execution time (ET) with the number of instructions,

$$ET = \#Instructions * CPI * CycleTime \quad (1)$$

where *CPI* is the mean number of cycles necessary to execute an instruction, and *CycleTime* is the clock period of the processor.

This model does not consider information about cache accesses and performance of the disk. However, although simple, it can provide interesting performance clues on the potential of multiprocessing architectures and aggressive instruction level parallelism exploitation for a wide range of different applications classes.

3.1.1 Low End Single Processor

Based on equation (1), for a Low-End Single processor (*SLE- Single Low End*), the execution time can be termed as:

$$ET_{SLE} = \#Instructions * (\propto CPI_{SLE} + \ \beta CPI_{SLE}) * CycleTime_{SLE} \ (2)$$

Since the low-end processor is a single-issue processor, it cannot exploit ILP. Therefore, classifying instructions as either α or β as previously stated does not make sense. In this case, α is zero and β equal to one, but we will keep the notation and their meaning for comparison purposes.

3.1.2 High End Single Processor

In the case of a high-end ILP exploitation architecture, based on equation (1) and (2), one can state that the *Execution Time of the High End Single Processor* (ET_{SHE}) is given by the following equation:

$$ET_{SHE} = \#Instructions * (\propto CPI_{SHE} + \ \beta \ CPI_{SLE}) * CycleTime_{SHE} \ (3)$$

The coefficients α and β refer to the percentage of instructions that can be executed in parallel or not (this way, α + β = 1), respectively. $CycleTime_{SHE}$ represents the clock cycle time of the high-end single processor.

The CPI_{SHE} is usually smaller than 1, because a single high-end processor can exploit high levels of ILP, thanks to the replication of functional units, branch prediction, speculative execution, mechanisms to handle false data dependencies and so on. A typical value of CPI_{SHE} for a current high-end single processor is 0.62 (GUTHAUS, RINGENBERG, *et al.*, 2002), which shows that more than one instruction can be issued and executed per cycle. The CPI_{SHE}, could also be written as $\propto CPI_{SLE}/issue$, where *issue* is the number of instructions that can be issued in parallel to the functional units, when considering the average situation (i.e., a High-End Single processor would have the same CPI as the CPI of a Low-End Processor divided by the mean number of instructions issued per cycle). Thus, based on equation (3), one gets:

$$ET_{SHE} = \#Instructions * \left(\frac{\propto CPI_{SLE}}{issue} + \beta\ CPI_{SLE}\right) * CycleTime_{SHE} \quad (4)$$

Having stated the equations to calculate the performance of both high-end and low-end single processor models, now the potential of using a homogeneous multiprocessing architecture to exploit TLP is studied. We consider that such architecture is built by the replication of low-end processors (that do not exploit ILP), so that a large number of them can be integrated within the same die.

If one considers that each application has a certain number of sequences of instructions that can be split (transformed to threads) to be executed on several processors, one could write the following equation, based on equations (1) and (2):

$$ET_{MP} = \#Instructions * \left(\frac{\delta}{P} + \gamma\right) * (\alpha CPI_{SLE} + \beta CPI_{SLE}) * CycleTime_{MP} \quad (5)$$

where δ is the amount of sequential code that can run in parallel (i.e. transformed into multithreaded code), while γ is the part of the code that must be executed sequentially (so no TLP is exploited). *P* is the number of low-end processors that is available in the chip. As can be observed in the second term of equation (5), because the single low-end processor is considered, the multiprocessor architecture does not exploit ILP ($\alpha = 0$ and $\beta = 1$). Therefore, when one increases the number of processors *P*, only the portion of code that presents TLP (δ) will benefit from the extra processors.

3.1.3 High-End Single Processor versus Homogeneous Multiprocessor Chip

First, we compare the performance of the high-end single processor to the multiprocessor architecture disregarding the communication overhead among the threads. This scenario demonstrates the potential results of the multiprocessing systems against a superscalar processor. Since power is crucial in an embedded system design, we have chosen a certain total power budget as a fair performance factor to compare both designs. Thus, based on equations (3) and (5), one can consider the following equation:

$$\frac{ET_{SHE}}{ET_{MP}} = \frac{\left[Instructions\left(\propto \frac{CPI_{SLE}}{issue} + \beta CPI_{SLE}\right) CycleTime_{SHE}\right]}{\left[Instructions\left(\frac{\delta}{P} + \gamma\right)(\propto CPI_{SLE} + \beta CPI_{SLE}) CycleTime_{MP}\right]} \quad (6)$$

If one considers that, in the model of the multiprocessor environment, a single low end processor is not capable of exploiting instruction level parallelism, and then $\propto = 0$, one can reduce the equation 6 to:

$$\frac{ET_{SHE}}{ET_{MP}} = \frac{\left[Instructions\left(\propto \frac{CPI_{SLE}}{issue} + \beta CPI_{SLE}\right) CycleTime_{SHE}\right]}{\left[Instructions\ \left(\frac{\delta}{P} + \gamma\right)(0 * CPI_{SLE} + 1 * CPI_{SLE}) CycleTime_{MP}\right]} \quad (7)$$

and, by simplifying (7), one obtains

$$\frac{ET_{SHE}}{ET_{MP}} = \frac{\left[Instructions\left(\propto \frac{CPI_{SLE}}{issue} + \beta CPI_{SLE}\right) CycleTime_{SHE}\right]}{\left[Instructions\ \left(\frac{\delta}{P} + \gamma\right)(CPI_{SLE}) CycleTime_{MP}\right]} \quad (8)$$

We are also considering that, as a homogeneous multiprocessor design is composed of several low-end processors with a very simple organization, those processors could run at much higher frequencies than a single and complex high-end processor. Therefore, we will assume that

$$\left(\frac{1}{CycleTime_{MP}}\right) = K * \left(\frac{1}{CycleTime_{SHE}}\right), \quad (9)$$

where *K* is the frequency adjustment factor to equal the power consumption of the homogeneous multiprocessor with the high-end single processor.

By merging and simplifying equations (8) and (9), one gets:

$$\frac{ET_{SHE}}{ET_{MP}} = \left[\frac{1}{\frac{\delta}{P} + \gamma}\right]\left[\frac{\propto \frac{CPI_{SLE}}{issue} + \beta CPI_{SLE}}{CPI_{SLE}}\right] K \quad (10)$$

According to equation (10), a machine based on a high-end single core will be faster than a multiprocessor-based machine if $\left(\frac{ET_{SHE}}{ET_{MP}}\right) < 1$. This equation also shows that, although the multiprocessor architecture with low-end simple processors could have a faster cycle time (by a factor of *K*), that factor alone is not enough to attain performance, as demonstrated in the second term in brackets of equation (10). Since the high-end processor can execute many instructions in parallel, better performance improvements can be obtained, as long as ILP is the dominant factor, instead of TLP.

To better illustrate this point, let us imagine the extreme case: $P=\infty$, meaning that infinite processors are available. In addition, if one considers that the multiprocessor design is composed of low end processors that do not exploit ILP and, therefore, $\propto CPI_{SLE}$ is always zero, it can be removed from the equation. Therefore, equation (10) reduces to:

$$\frac{ET_{SHE}}{ET_{MP}} = \left[\frac{\propto \frac{CPI_{SLE}}{issue} + \beta CPI_{SLE}}{\gamma CPI_{SLE}}\right] K \quad (11)$$

Let us consider that the execution of the very same application on both multiprocessor and single high-end architectures presents exactly the same amount of sequential code, so $\beta = \gamma$. In this case, the operating frequency (given by the K factor) will determine which architecture runs faster if the issue width of the high-end superscalar processor also tends to infinite.

In another example, if one applies equation 11 in a scenario where an application

presents 10% of sequential code ($\beta = \gamma = 0.1$) and it is executing on a four issue high-end single processor, the operating frequency of the four issue high-end single processor should be 3.2 times (K=0.31) greater than the multiprocessor to achieve the same execution time. On the other hand, if that application now presents 90% of sequential code ($\beta = \gamma = 0.9$), the high-end single processor should run only 20% (K=0.8) faster than the multiprocessor design. With these corner cases, one can conclude that when the applications are massively sequential, both architectures, operating at the same frequency, will present almost the same performance, regardless the number of processors in a multiprocessor system. For applications with huge amount of parallel code, complex single processors must run at higher frequencies than multiprocessors systems.

3.1.4 Applying the Performance Modeling in Real Processors

Given the analytical model, one can briefly experiment it with numbers based on real data. Let us consider a high-end single core: a 4-issue SPARC64 superscalar processor with CPI equal to 0.6 (GUTHAUS, RINGENBERG, *et al.*, 2002); and a multiprocessor design composed of low-end single-issue TurboSPARC processors with CPI equal to 1.3 (GUTHAUS, RINGENBERG, *et al.*, 2002). A comparison between both architectures is done using the equations of the aforementioned analytical model. In addition, we consider that the TurboSPARC has 5,200,000 transistors (FUJITSU MICROELECTRONICS), and that the SPARC64 V design (DIEFENDORFF, 1999) requires 180,000,000 transistors to be implemented. For the multiprocessing design we add 37% of area overhead due to the intercommunication mechanism (INTEL, 2007). Therefore, aiming to make a fair performance comparison among the high-end single core and the multiprocessor system, we have devised an 18-Core design composed of low-end processors that has the same area of the 4-issue superscalar processor and consumes the same amount of power.

Figure 16 shows, in a logarithmic scale, the performance of the superscalar processor, when parameters α and β change, and the performance of the many in-order TurboSPARC cores, when the δ and γ and the number of processors (from 8 to 128) varies. The x-axis of Figure 16 represents the amount of the instruction- and thread-level parallelism in the application, considering that the α factor is only valid for the superscalar processor, while δ is valid for all the multiprocessing systems' setups.

The goal of this comparison is to demonstrate which technique better explores its particular parallelism type at different levels, considering six values for both ILP and TLP. For instance, δ=0.01 means that a hypothetic application only shows 1% of TLP available within its code (in the case of the multiprocessing systems). In the same way, when α=0.01, it is assumed that only 1% of the total number of instructions can be executed in parallel on the superscalar processor. In these experiments, we considered the same power budget for the high-end single core and the multiprocessor approaches. In order to normalize the power budget of both approaches we have tuned the adjustment factor K of equation 9. For that, we fixed the power consumption of the 4-issue superscalar to use it as the reference, changing the operating frequency (K factor) of the remaining approaches to achieve the same power consumption.

Thus, the operating frequency of the 8-Core multiprocessing system must be 3 times higher than the one of the 4-issue superscalar processor. For the 18-Core setup, the operating frequency must be a 25% higher than the reference value. Since a considerable number of cores is employed in the 48-Core setup, it must execute 2 times

slower than the superscalar processor to operate under the same power budget. Finally, the operating frequency of the 128-Core design must be 5.3 times lower than the superscalar.

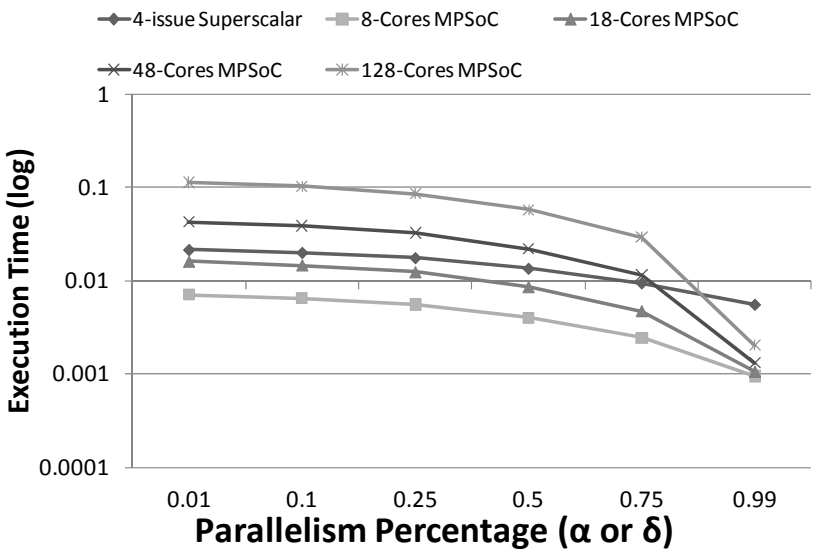


Figure 16. Multiprocessor system and Superscalar performance regarding a power budget using different ILP and TLP; $\alpha = \delta$ is assumed.

In the leftmost side of Figure 16, one considers any application that has a minimum amount of instruction (α=0.01) and thread (δ =0.01) level parallelism. In this case, the superscalar processor is slower than the 8- and 18- Core designs since the parallelism is insignificant, the higher operating frequency of both multiprocessing system is responsible for faster execution. Moreover, when the application shows higher parallelism levels (α>0.25 and δ>0.25), the 18- and 8-Core better handles the extra TLP available than the superscalar does with the ILP, presenting more performance improvements. So, considering only the 18- Core design, the multiprocessing system achieve better performance with the same area and power budget in the whole spectrum of parallelism available.

However, as more cores are added in a multiprocessor design, the overall clock frequency tends to decrease, since the adjustment factor K must be decreased to respect the power budget. Therefore, the performance of applications that present low TLP (small δ) worsens when the number of cores increases. Applications with δ =0.01 in Figure 16 are good examples of this case: performance is significantly affected as the number of cores increases. As another representative example, even when almost the whole application presents high TLP (δ > 0.99), the 128-Core design takes longer than the other multiprocessor designs. Figure 16 concludes that the increasing on the number of cores not always produces a satisfactory tradeoff among energy, performance and area.

3.1.5 Communication Modeling in Multiprocessing Systems

As stated the heavy task to outperform the multiprocessing systems' performance when considering a scenario where there is no communication among threads. Now, we

introduce this issue in the modeling to bring the analytical model closer from a real multiprocessor design. For this purpose, we use the communication strategy shown in (CHEN, LU, *et al.*, 2009). The employment of a 2D-mesh NoC is supported by its higher scalability, energy efficiency and area overhead over buses and crossbar interconnections. According to (CHEN, LU, *et al.*, 2009), the communication latency in a 2D-mesh NoC is divided in two parts: minimal latency and contention latency. The minimal latency is calculated by hop count, which means the distance of the two communication tasks within the NoC. The contention latency depends on the arbitration mechanism and the way that the routing is implemented.

As the amount of data stored in each processor varies for each application, we modeled the corner cases to state the best and worst case of the communication overhead. The best communication case is considered when the traffic of data is uniformly distributed among the NoC-nodes. On the other hand, when the traffic of data is concentrated in from/to a specific node, the worst case occurs. The latter strategy is named as centralized traffic and the former as distributed traffic. We use $AgvHops$ to model the minimal latency of both communication strategies, meaning the average number of hops performed by a single inter-thread communication.

Distributed
$$\begin{cases} AvgHops = \frac{2h}{3}, \ h\ even & (12) \\ AvgHops = 2\left(\frac{h}{3} - \frac{1}{3h}\right), \ \ h\ odd & (13) \end{cases}$$

Centralized
$$AvgHops = \frac{k^2}{k+1} \quad (14)$$

where $h = \sqrt{N}$, being N the number of processor nodes.

As we already stated the average number of hops for a single communication, now we can use this measurement to state the communication latency (CL),

$$CL = AgvHops * \#Communications * Cycletime_{NoC} \quad (14)$$

where $Cycletime_{NoC}$ is the clock period of the Network-on-Chip, $\#Communications$ is the total number of communications performed by the processors considering the whole application execution.

Communications (#Communications) occurred in the execution of a certain application can be divided in:

- ε – the portion of instructions that are data transfer (e.g. load and store instructions) and performs accesses in a local storage. It depends on δ that is the amount of instructions that can be distributed among the processors of the multiprocessor environment. Thus, $\varepsilon = 0.25$ means that 25% of the instructions executed in parallel are data transfer accessing local storage.
- θ – the portion of instructions that are data transfer (e.g. load and store instructions) and performs accesses in a remote storage. It depends on δ that is the amount of instructions that can be distributed among the processors of the multiprocessor environment. Thus, $\theta = 0.25$ means that 25% of the instructions executed in parallel are data transfer in a remote storage and 75% of the data transfer are in a local storage.

To model the number of communications we need only θ, since ε does not produce traffic in the NoC. In this way, one can get

$$\#Communications = \#Instructions * \delta * \theta \quad (15)$$

To make simple the communication model, we will overlook the contention latency and assume that the routers take one clock cycle to route a data from the input to the output.

Now, considering the communication latency (CL) on the multiprocessing system, one can get, based on the equation (5) and (15),

$$ET_{MP} = Instructions\left(\frac{\delta}{P} + \gamma\right)(\alpha CPI_{SLE} + \beta CPI_{SLE})\, CycleTime_{MP} + AgvHops * \#Communications * Cycletime_{NoC} \quad (16)$$

3.1.6 Applying the Performance Modeling in Real Processors considering the Communication Overhead

We apply the same data presented in Section 3.1.4 on the analytical model considering the communication overhead. We create four different scenarios to show the communication overhead: $\theta = 0.16$, $\theta = 0.33$, $\theta = 0.66$, $\theta = 0.99$, meaning that 16%, 33%, 66% and 99% of the instructions executed in parallel produces data traffic in the NoC, respectively. We also considered the distributed and centralized modeling to calculated $AgvHops$.

In these experiments, we considered the same power budget for the high-end single core and the multiprocessor systems. In order to normalize the power budget of both approaches we have to tune again the operating frequency (K of equation 9 in the Section 3.1.4) of the multiprocessing systems to achieve the same power consumption since the power of the NoC should be considered.

Thus, the operating frequency of the 8-Core multiprocessing system changes from 3 times higher than the one of the 4-issue superscalar processor to 2.7 times. For the 18-Core setup, the operating frequency changes from 25% higher to 13% higher than the superscalar processor. The operating frequency of 48-Core setup changes from 2 times slower than the superscalar processor to 2.2 times. Finally, the operating frequency of the 128-Core design changes from 5.3 times lower to 5.9 times slower than the superscalar processor.

Figure 17 draws the execution time of all designs presented in Section 3.1.4 considering $\theta = 0.16$. When applications provide low levels of thread level parallelism ($0.01 < \delta < 0.25$), the communication overhead does not affect the execution time. It can be notice by comparing Figure 16, where no communication is considered, with Figure 17. However, when the TLP increases ($\delta > 0.25$), the impact of the communication becomes more evident, the curves of the multiprocessing designs do not fall in the same pace as the curves shown in Figure 16, since more TLP means more communication. As expected, when the data is centralized in one processor, the impact in the execution time is greater than when it is distributed, since the average number of hops is greater in the former case. Even applying such small communication overhead some conclusions presented in the Section 3.1.4 changed, the 4-issue superscalar processor outperforms the 18-, 48- and 128- Core Designs when $\delta > 0.95$ and application presents centralized data.

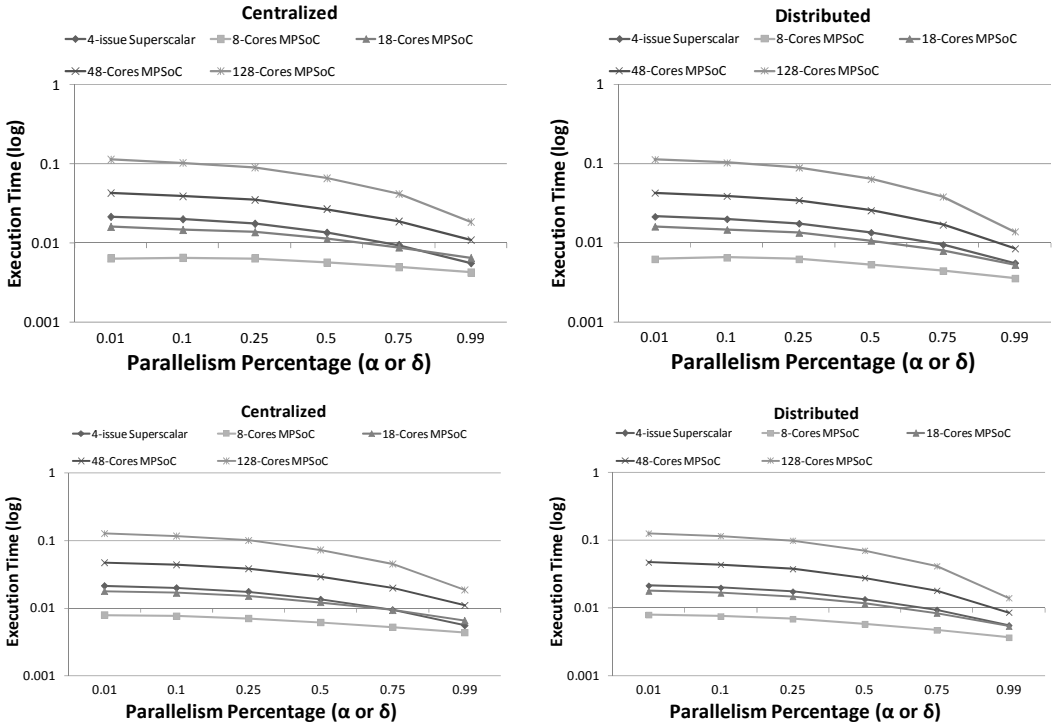


Figure 17.Execution time of different designs considering $\theta = 0.16$

The multiprocessing designs start losing steam when the communication overhead increases to 33%. The gains obtained by exploring higher thread level parallelism are hiding by the communication overhead. As can be seen in Figure 18, the 8-Core Design provides the same execution time regardless the available thread level parallelism (δ). Considering this degree of communication, the 4-issue superscalar processor achieves better execution time than all multiprocessing systems when $\delta > 0.95$ and data are centralized. The superscalar processor outperforms the 18-Core Design (both designs present the same area and power budget) when 50% of the application could be parallelized either in instructions or in threads. When data is uniformly distributed among the cores, with 33% of communication overhead, the 8-Core Design still compete with 4-issue superscalar processor.

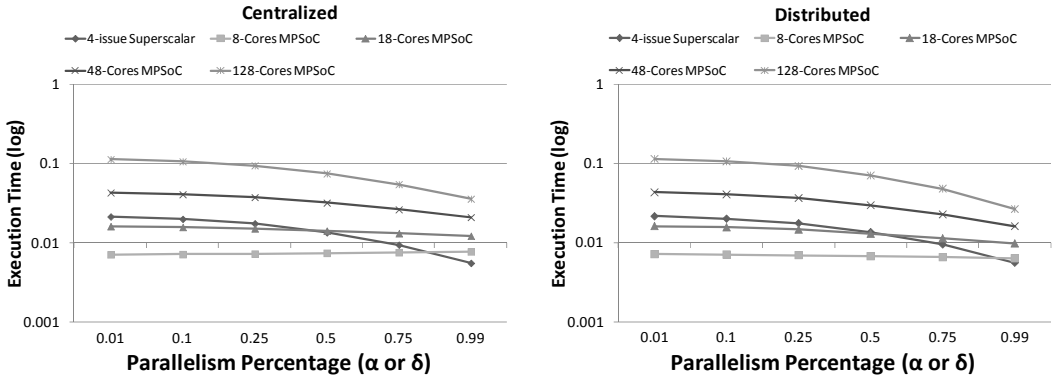


Figure 18. Execution time of different designs considering $\theta = 0.33$

When the communication overcome 66% of parallel instructions, multiprocessing systems with many cores (48- and 128) does not show more gains with the increasing

on the thread level parallelism (Figure 19), the curves becomes almost flat, meaning that applications with higher communication degree are not suitable even for many core designs. The scenario is more critical for 8- and 18-Core Designs, considering this communication degree, the increasing on thread level parallelism produces losses in performance. When 25% of the application would be parallelized either in instructions or in threads, the 4-issue superscalar processor outperforms the 18-Core Design. However, when 65% of parallelism is available in both levels, the superscalar processor outperforms the 8-Core Design.

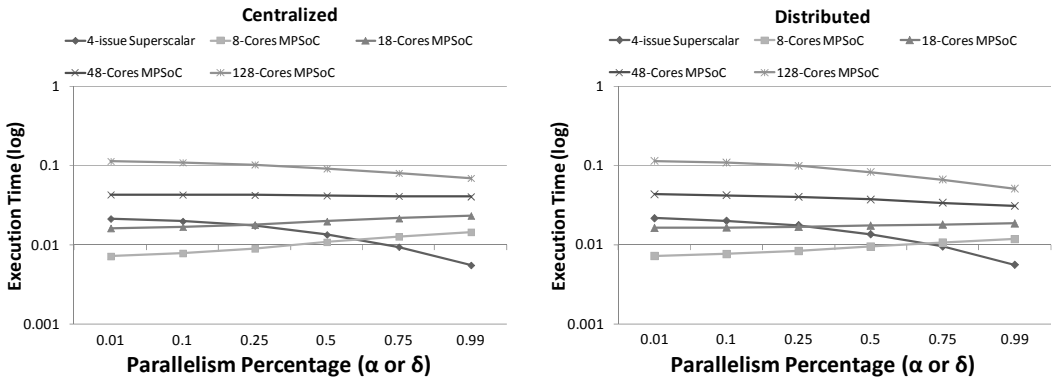


Figure 19. Execution time of different designs considering $\theta = 0.66$

Figure 20 shows a scenario where 99% of parallel instructions produce communication in the NoC. Although this scenario is unlikely, it was built only to show the potential of superscalar over the multiprocessing designs. As we had concluded above, the multiprocessing system composed of huge number of cores, due to the power budget assumption, are not feasible for applications based on high communication overhead. Considering the 8-Core Design, if applications have their code parallelized, either in instructions or in threads, up to 50%, regardless the communication overhead, the designer should decide for 8-Core Design instead of 4-issue superscalar processor. Parallelism higher than 50%, the designer should select the latter. However, if one considers designs with the same area and power budget, it means 4-issue superscalar versus 18-Core Design, the former outperforms the latter from 15% of parallelism available.

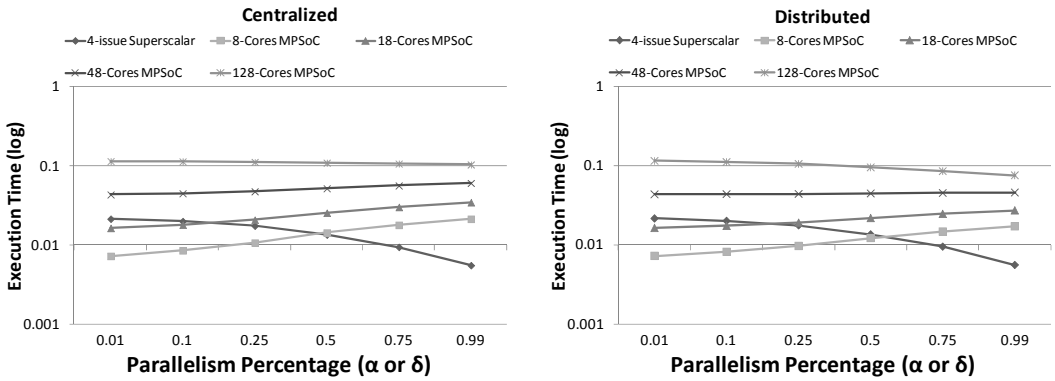


Figure 20. Execution time of different designs considering $\theta = 0.99$

3.2 Energy Comparison

If superscalar processors present better performance than multiprocessing systems when communication is considered, when one measures energy consumption there is a completely new scenario: power in CMOS circuits is proportional to the switching capacitance, to the operating frequency and to the square of the power supply. In the simple energy model that we will present herein, we will assume that the power is dissipated only in the data path. This is overly optimistic in regards of dissipated power by a superscalar processor, but this can also give an idea of the lower bound of energy dissipation in high-end single processors.

The power dissipated by a high-end single processor can be written as

$$P_{SHE} \approx issue * C * \left(\frac{1}{CPI_{SLE}}\right) * V_{SHE}^2 \quad (17)$$

were C is the capacitance switching of the single issue processor, and V_{SHE} is the voltage the processor is operating on. The term $\left(\frac{1}{CPI_{SHE}}\right)$ is included to consider the extra power required during the speculation process to sustain performance with a CPI smaller than 1. The energy of the high-end single processor is given by:

$$E_{SHE} = P_{SHE} * T_{SHE}. \quad (18)$$

and the power consumed by a homogeneous multiprocessing system is given by

$$P_{MP} \approx P * C * \left(\frac{1}{CPI_{SLE}}\right) * V_{MP}^2 \quad (19)$$

As in the case of superscalar processor, the term considering the CPI of the single low-end processor $\left(\frac{1}{CPI_{SLE}}\right)$ has been also included. The energy of the single low-end processor is given by

$$E_{MP} = P_{MP} * T_{MP}, \quad (20)$$

It is possible to term E_{SHE} and E_{MP} as:

$$\frac{E_{SHE}}{E_{MP}} = \left[\frac{1}{\frac{\delta}{P} + \gamma}\right]\left[\frac{\propto \frac{CPI_{SLE}}{issue} + \beta CPI_{SLE}}{CPI_{SLE}}\right] * K * \left[\frac{issue * C * \left(\frac{1}{CPI_{SLE}}\right) * V_{SHE}^2}{C * \left(\frac{1}{CPI_{SLE}}\right) * V_{MP}^2}\right] \quad (16)$$

simplifying (16), one gets

$$\frac{E_{SHE}}{E_{MP}} = \left[\frac{V_{SHE}^2 (\propto + issue * \beta)}{V_{MP}^2 (\delta + P\gamma)}\right] * K \quad (21)$$

Equation (17) demonstrates that both approaches unnecessarily spend power when there is no ILP or TLP available since there is no power management technique modeled to reduce power supply (V_{SHE}^2 and V_{MP}^2).

3.2.1 Applying the Energy Modeling in Real Processors

Figure 21 shows the energy results considering the same power budget, as it was already done in the performance model. For this first experiment, we do not consider the communication overhead for the multiprocessing environment that will be modeled later. In addition, we only show the energy of 8- and 18-Core Designs, since the

conclusions of these setups are also valid for the rest of the setups.

The high-end single processor organization spends higher energy than the 18-Core multiprocessor the same amount of energy when considering all levels of available parallelism since the latter is faster than the former in all cases (Figure 16).

To obey the given power budget, the 8-Core multiprocessor runs 3 times faster than 4-issue superscalar and the 18-Core multiprocessor. Thus, as the 8-Core Design present 3 times lower execution time than the 4-issue superscalar, the former spends 3 times less energy. When the parallelism is more exposed the superscalar approaches to the 8-Core Design, since its execution time decreases. Multiprocessors composed of a significant number of cores present worst performance in applications with low/medium TLP (Figure 16). Consequently, in those cases and if no power management techniques are considered (e.g., cores are turned off when not used), energy consumption of such multiprocessor designs tend to be higher than those with fewer cores. As can be seen in Figure 21, the 8-Core multiprocessor consumes less energy than the 18-Core for low/medium TLP values ($\delta < 0.75$). However, when applications present greater thread level parallelism ($\delta > 0.9$), the energy consumed by the 18-Core multiprocessor reaches the same values as the 8-Core design, thanks to the better usage of the available processors.

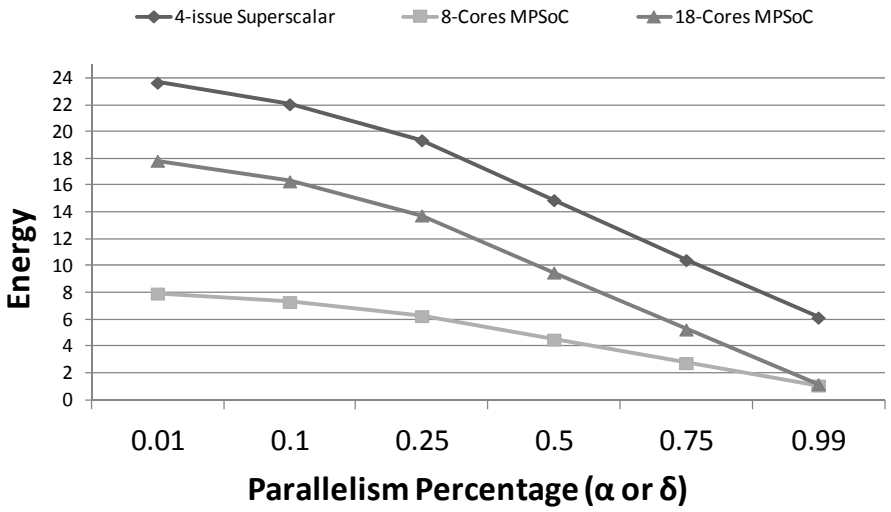


Figure 21. Multiprocessing Systems and High-end single processor energy consumption; $\alpha = \delta$ is assumed.

3.2.2 Communication Modeling in Energy of Multiprocessing Systems

The energy results shown in Figure 21 do not consider the communication produced by data transfer. The power dissipated by the Network on Chip is proportional to its capacitance, operating frequency, square of the power supply, power of one router and the number of routers. With these variables one can get,

$$P_{NoC} \approx \#Routers * C * P_{Router} * V_{NoC}^2 \quad (22)$$

The energy of the NoC can be modeled as,

$$E_{NoC} = P_{NoC} * CL \quad (23)$$

being CL the communication latency modeled in Equation 14.

The energy of the multiprocessing system, based on Equation 23 and 30, is given by,

$$E_{MP} = P_{MP} * T_{MP} + P_{NoC} * CL \quad (24)$$

3.2.3 Applying the Energy Modeling in Real Processors considering the Communication Overhead for Multiprocessing Systems

The energy consumption shown in Section 3.2.1 does not consider communication costs. Thus, we apply the equations shown in Section 3.2.2 using the same scenarios presented in Section 3.1.5, where we consider that the data traffic in the NoC is produced by 16%, 33%, 66% and 99% of the instructions executed in parallel.

Figure 22 draws the energy consumption of the multiprocessing systems and the 4-issue superscalar processor. As can be seen, the curve of the superscalar processor places above of the 18-Core and 8-Core designs up to 85% of parallelism, since the multiprocessing systems provide lower execution time than the superscalar processor (refer to Figure 17). As the parallelism grows, the superscalar decreases the energy consumption in a higher factor than the 18-Core due to its more efficient exploitation. Considering neither the centralized nor the distributed data schemes, the superscalar processor fails both in performance and in energy consumption in comparison to the 8-Core design when 16% of communication is considered.

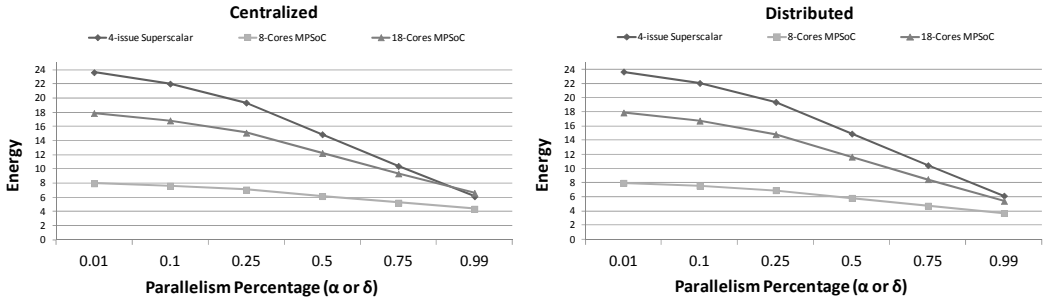


Figure 22. Energy consumption of different designs considering $\theta = 0.16$

The scenario changes when the communication increases to 33% (Figure 23), the superscalar processor outperforms the 18-Core design and achieves better energy consumption when the parallelism reaches 50%. When the data traffic increases to 66% (Figure 24), the superscalar processor shows better performance and energy consumption than the 8-Core design after 25% of parallelism. Thus, we can conclude that, for a same area design, the employment of a 4-issue superscalar processor is more energy and performance efficient than a 18-Core design to execute the following scenario: an application that makes available 25% of TLP or ILP and more than 66% of its instructions executed in parallel produce inter-thread communication. However, in comparison with the 8-Core design, the employment of the superscalar processor is worthwhile only when the application provides higher than 65% and 75% of parallelism, for centralized and distributed communication approaches, respectively.

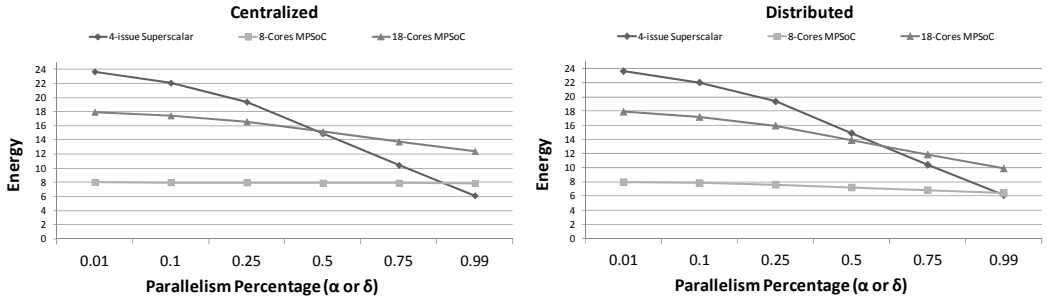


Figure 23. Energy consumption of different designs considering θ =0.33

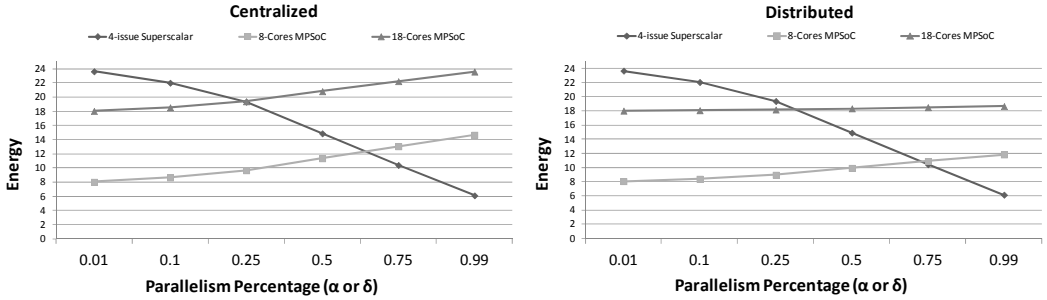


Figure 24. Energy consumption of different designs considering θ =0.66

Figure 25 shows the worst scenario for the multiprocessing system, when 99% of the parallel instructions produce data traffic in the NoC. Here, one can conclude that, for a same chip area design, if an application can be parallelized up to 20%, the employment of the 18-Core Design is worthwhile than the superscalar in both performance and energy consumption when communication is centralized. This percentage increases to almost 25% when the data are uniformly distributed among the processors. The 8-Core design is more competitive, when an application is parallelized up to 50%. Besides 8-Core design occupies less area, it produces better performance and energy consumption than the superscalar processor when the communication is centralized. This percentage increases to 55% when the data is distributed uniformly among processors.

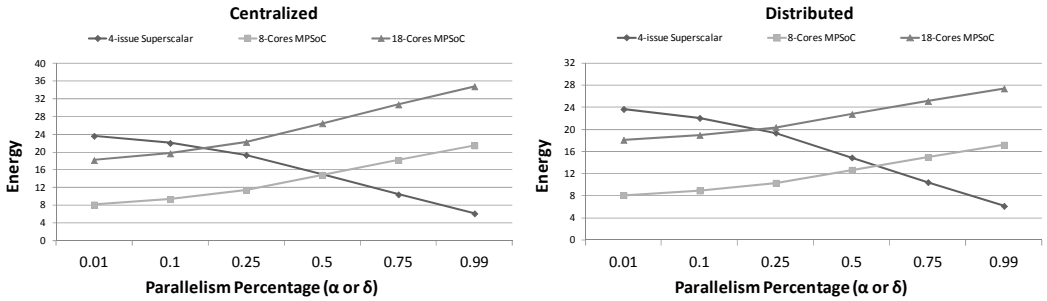


Figure 25. Energy consumption of different designs considering θ =0.99

Summarizing, the best scenario for TLP exploitation (0% of communication (Figure 16)) shows that the 8-Core and 18-Core design outperforms the superscalar processor in the whole spectrum of parallelism. On the other hand, when the worst scenario for TLP exploitation is applied (99% of communication), the superscalar processor provides better performance and energy when the parallelism is higher than 25% and 50%, for 18-Core and 8-Core, respectively. Thus, the analytical model shows that the

performance of the multiprocessing systems, besides relying on TLP available in the application, it also heavily depends on the communication rate.

One can conclude that the ideal approach would be the usage of a heterogeneous multiprocessor system to exploit both TLP and ILP, so it would be possible to balance the performance and energy of a wide range of application domains and would be possible to avoid the huge communication costs provided by designs based on many cores. To support such assumption, there is an additional tradeoff that must be considered, when more cores are included in the chip, the multiprocessor performance tends to worsen since the operating frequency must be decreased to respect the power budget limits. For instance, the 128-Core design takes longer execution time than the other multiprocessor designs in all levels of parallelism available (Figure 16) since its operating frequency is very low.

Considering real applications, thread level parallelism exploitation is widespread employed to accelerate most multimedia applications used in the embedded devices thanks to their data independent iteration loops. However, even applications with high TLP could still obtain some performance improvement by also exploiting ILP. Hence, in a multiprocessor design, ILP techniques also should be investigated to conclude what is the best fit considering the particular design requirements. Hence, the analytical modeling indicates that heterogeneous multiprocessor system is necessary to balance the performance and energy of a wide range of application classes. Section 3.3 reinforces this trend by running a real embedded application over a multiprocessor environment only exploiting TLP, a superscalar processor and a multiprocessor environment exploiting both TLP and ILP.

3.3 Example of a Application Parallelization Process in a Multiprocessing System

We evaluate the performance of both superscalar and multiprocessor environments regarding an actual application execution. An 18-tap FIR filter is used as a benchmark to make this evaluation. The C-like description of the FIR filter employed in this experiment is illustrated in Figure 27. Superscalar machines explore the instruction level parallelism of such an application in a transparent way, working on its original binary code. Unlike the superscalar approach, to explore the potential of the multiprocessor architecture there is a need to make manual source code annotations in order to split the application code among many processors. In this way, some code highlights are shown in Figure 27 to simulate these annotations, indicating the necessary number of cores to explore the ideal thread level parallelism of each part of the FIR filter code. For instance, the first annotation considers a loop controlled for IMP_SIZE value, which depends on the number of FIR taps. In this case, 54 loop iterations are done since the experiment regards an 18-tap FIR filter.

The OpenMP (MENON, 1998) programming language provides specific code directives to easily split loop iterations among processors. Using OpenMP directives, the ideal exploration of this loop is done through 54-Core multiprocessor design, each one being responsible for executing a loop iteration. However, when the amount of processors is lower than the number of loop iterations, OpenMP combines them in groups and distributes the tasks among the available resources. Hence, regarding the execution of 54 loop iterations in a 18-Cores multiprocessor design, OpenMP creates 18 groups, each one composed of 3 iterations. Since the FIR code is made up of several

loops, almost the entire application can be parallelized, as shown in Figure 27. In general, DSP applications (ex: FFT and DCT) are loop-based which turns it suitable for OpenMP usage. However, despite OpenMP use, some loops still executing sequentially due to the data dependency among iterations. This fact is illustrated by the last loop of the FIR filter description, since the iterations of the loop that perform shifting in the array presents dependencies among each other.

We evaluated the 18-tap FIR execution over three different architectures aiming to illustrate their performance impact on applying TLP and ILP exploration: a 4-issue superscalar SPARC V (SS); 6- 18- and 54- Core multiprocessor designs based on in-order single-issue TurboSPARC cores, with no ILP exploration capabilities (MP_{IOC}). Finally, in order to have a glimpse on the future, we imagined a 6- 18- and 54- Cores MPs based on a 4-issue Superscalar processor, able to explore both ILP and TLP (MP_{SS}). We have gathered data about performance with a cycle accurate simulator (RUTZIG, BECK e CARRO, 2009). The execution time is measured in order to obtain their speedup over the baseline processor. It is important to point out that instruction and thread communication overhead has not been considered in this experiment.

The results shown in Figure 26 reflect the speedup provided over a single in-order core performance running the sequential code version of the C-like description of the 18-tap FIR filter presented in Figure 27. The leftmost bar shows the speedup provided for the ILP exploration of a 4-issue superscalar processor. In this case, the speedup of the superscalar processor over an in-order core is only 2.2 times showing that the FIR filter has neither high nor low ILP since a 4-issue superscalar processor could theoretically achieve up to 4 times the performance of an single-issue in-order core.

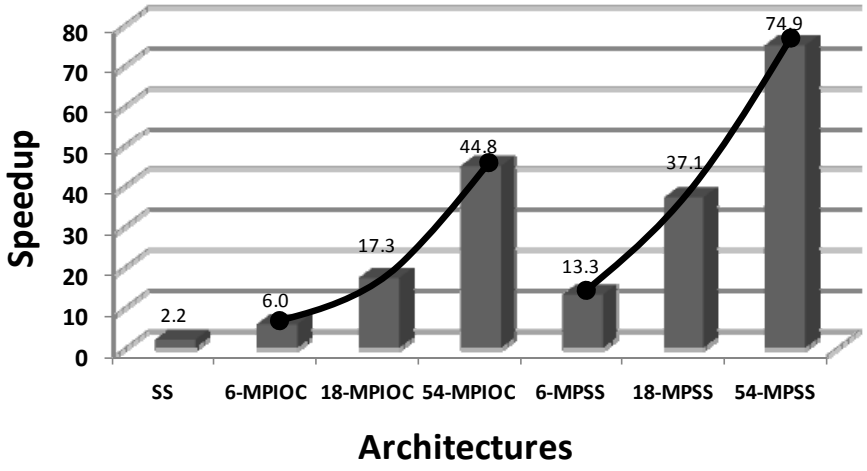


Figure 26. Speedup provided in 18-tap FIR filter execution for Superscalar, MPSoC and a mix of both approaches

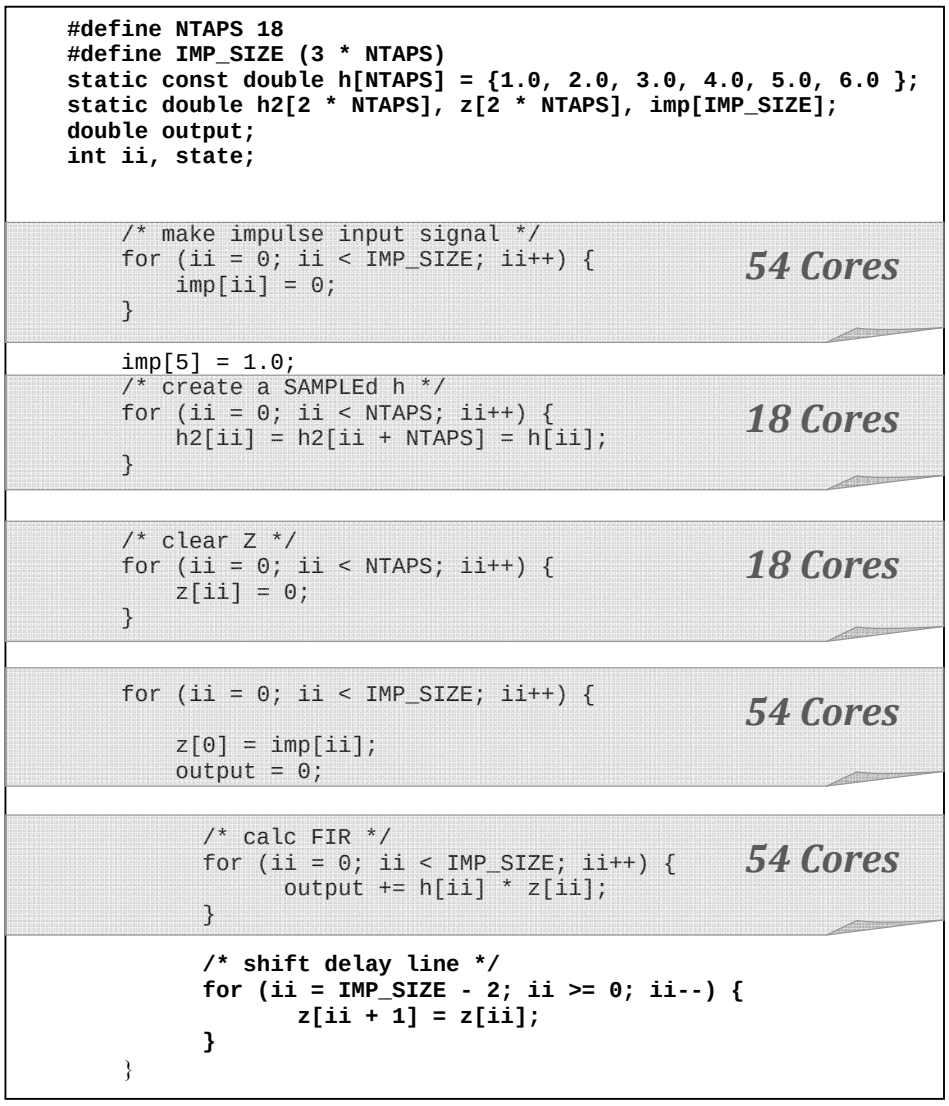


Figure 27. C-like FIR Filter

Regarding the multiprocessor designs composed of in-order cores, the 6-Core machine provides almost a linear speedup, decreasing the single in-order core execution time by 5.46 times. This behavior is maintained when more in-order cores are inserted. However, when the TLP of the 18-tap FIR filter is aggressively explored (54-MP_{IOC}), a speedup of only 44.8 times is achieved, showing that even applications that are potentially suitable for TLP exploration suffer with the presence of sequential code parts.

Amdahl´s Law shows that it is not sufficient to build architectures with a large number of processors, since most applications contain a certain amount of sequential code (WOO e LEE, 2008). Hence, there is a need to balance the number of processors

with a suitable ILP exploration approach to achieve greater performance. The MP_{SS} approach combines TLP with ILP exploration of 4-issue superscalar aiming to show that simple TLP extraction is not enough to achieve linear speedups even for applications with high TLP. Figure 26 illustrates the speedup of the MP_{SS} approach. As can be seen, the performance of 6-MP_{SS} on running 18-tap FIR filter is twice better than 6-MP_{IOC}. The 6-MP_{SS} takes advantage of the large room for ILP exploitation provided when the applications is split in only 6 threads. In this case, when the first loop of Figure 27 is split among the multiprocessor machine, each core receives 9 loop iterations that executed sequentially. However, when the number of cores increases, the sequential code decreases, making lower the room for the ILP optimization. Nevertheless, the 18-tap FIR filter execution in 54-MP_{SS} is 66% faster than the 54-MP_{IOC} execution.

Summarizing, some real applications could benefit for thread level parallelism exploration thanks to their loop-based behavior. However, even applications with high TLP could still obtain some performance improvement by also exploiting ILP. Hence, in a multiprocessor design ILP techniques also should be investigated to conclude what is the best fit considering the design requirements/constraints. Finally, one could conclude that replications of simple processing elements leaves a significant optimization possibility unexplored, indicating that mixed parallelism exploitation could be a possible solution to balance the performance when applications with different behaviors are considered.

4 CREAMS

A general overview of the CReAMS platform is given in Figure 28 (a). The thread level parallelism is explored by replicating the number of Dynamic Adaptive Processors (DAPs) (in the example of the Figure 28 (a), by four DAPs). The communication among DAPs is done through an on-chip unified 512 KB 8-way set associative L2 shared cache. As mixed parallelism exploitation is mandatory when a heterogeneous software environment is considered, we extend the single-thread based reconfigurable architecture presented in (BECK, RUTZIG, *et al.*, 2008) to handle multithreaded applications in CReAMS platform.

4.1 Dynamic Adaptive Processor (DAP)

We divided DAP in four blocks to better explain it, as illustrated in Figure 28(b). These blocks are discussed in the following sections.

4.1.1 Processor Pipeline (Block 2)

A SparcV8-Based architecture is used as the baseline processor to work together with the reconfigurable system. Its five stage pipeline reflects a traditional RISC execution flow (instruction fetch, decode, execution, data fetch and write back) that support its employment on embedded system. In addition, similarities to the processors used in well-known embedded platforms (e.g. OMAP) support the employment of SparcV8 processor in this work, since all are based on RISC architectures (e.g. MIPS, ARM).

4.1.2 Reconfigurable Data Path Structure (Block 1)

Following the classifications shown in (HAUCK e COMPTON, 2002), the reconfigurable data path is coarse-grained and tightly coupled to the SparcV8 pipeline, avoiding external accesses to the memory, saving power and reducing the reconfiguration time. Because it is coarse-grained, the size of the memory necessary to keep each configuration is lower when compared to fine-grained data paths (e.g. FPGAs), since the basic processing elements are functional units that work at the word level (arithmetic and logic, memory access and multiplier). As illustrated in the Figure 28(b), the data path is organized as a matrix of rows and columns. The number of rows dictates the maximum instruction level parallelism that can be exploited, since instructions located at the same column are executed in parallel. For example, the illustrated data path (Block 1 of Figure 28(b)) is able to execute up to four arithmetic and logic operations, two memory accesses (two memory ports are available in the L1 data cache) and one multiplication without true (read after write) dependences. The number of columns determines the maximum number of data dependent instructions

that can be stored in one configuration. Three columns of arithmetic and logic units (ALU) compose a level. A level does not affect the SparcV8 critical path (which, in this case, is given by the multiplier circuit). Therefore, up to three ALU instructions can be executed in the reconfigurable data path within one SparcV8 cycle, without affecting its original frequency (600 MHz). Memory accesses and multiplications take one equivalent SparcV8 cycle to perform their operations.

We have coupled sleep transistors (SHI e HOWARD, 2006) to switch power on/off of each functional unit in the reconfigurable data path. The dynamic reconfiguration process is responsible for the sleep transistors management. Their states are stored in the reconfiguration memory, together with the reconfiguration data. Thus, for a given configuration, idle functional units are set to the off state, avoiding leakage or dynamic power dissipation, since the incoming bits do not produce switching activity in the disconnected circuit. Although the sleep transistors are bigger and in series to the regular transistors used to implement the data path circuit, they have been designed so that their delays do not significantly impact the critical path or the reconfiguration time.

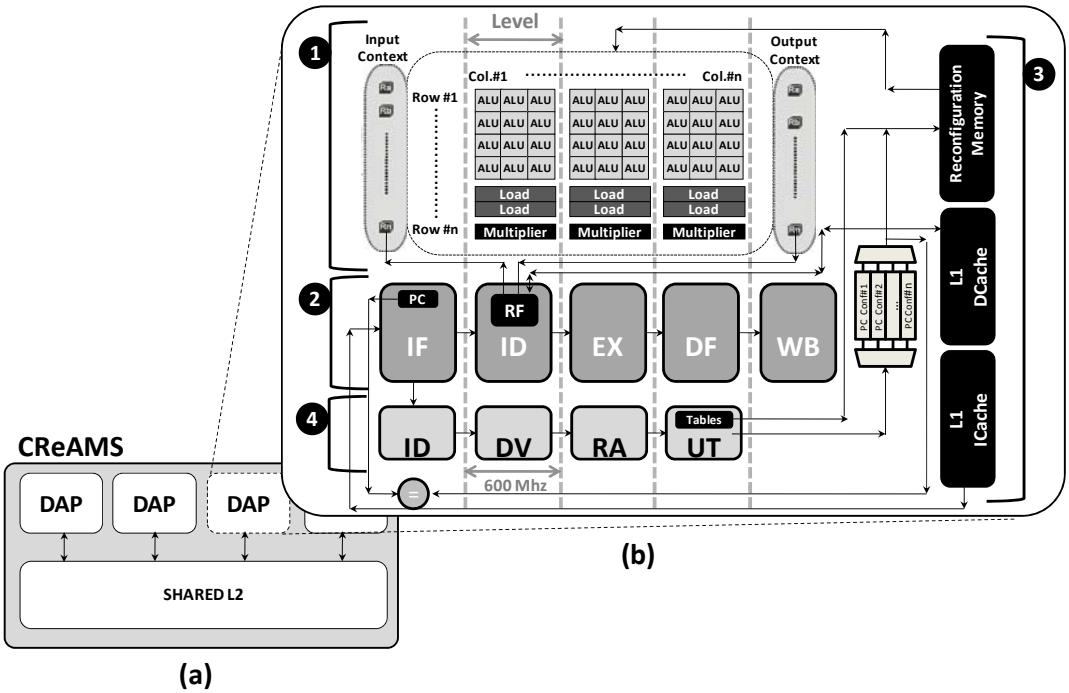


Figure 28. (a) CReAMS architecture (b) DAP blocks

The entire structure of the reconfigurable data path is totally combinational: there is no temporal barrier among the functional units. The only exception is for the entry and exit points. The entry point is used to keep the input context and the exit point is used to store the results, both structures are connected to the processor register file.

The feeding of the input context with the necessary data is the first step to configure the data path before firing the data path execution. After that, results are stored in the output context registers through the exit point of the data path. The values stored in the output context are sent to the SparcV8 register file on demand. It means that if any value is produced at any data path level (a cycle of SparcV8 processor) and if it will not be changed in the subsequent levels, this value is written back in the cycle after that it was produced. In the current implementation, the SparcV8 register file has two write/read ports.

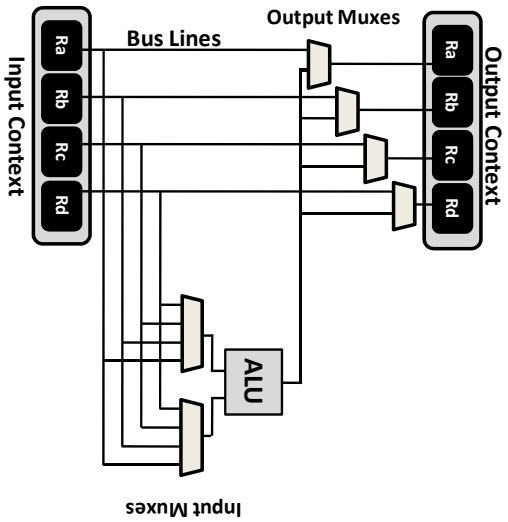


Figure 29. Interconnection mechanism

The interconnection structure of the reconfigurable data path is shown in the Figure 29. The data that comes from the SparcV8 register file is stored in the register of the input context. For each register, a bus line propagates the value to the functional units. These bus lines are connected to the functional units through multiplexers that are responsible for choosing the correct value. Each functional unit has two multiplexers in their inputs that make the selection of the issuing operands. We call them input multiplexers. After the operation is completed, there is a multiplexer for each bus line that will choose which result will be bypassed through that bus line. These are the output multiplexers.

The input and the output context size limits the number of instructions allocated in a single data path configuration, when all registers of the input context are already allocated, a new configuration should be created to hold the following instructions. On the other hand, a small input context size restricts the performance since a configuration is broken even if having available functional units. On the other hand, an increase in the size of the input context provides a huge overhead in the data path area, since each input register entails in one output multiplexer per data path column. For instance, in the data path presented in the Figure 28 (b), each additional input register aggregates nine output multiplexers in the data path structure. Moreover, each additional input register increases one input port in all input multiplexers of the reconfigurable data path. The interconnection structure also provides deleterious effects on energy consumption. Sleep transistors do not work in these components meaning that the data will be propagate over all interconnection structure due to the combinational fashion of the data path, which will spend power even when the functional units have not being used.

4.1.3 Dynamic Detection Hardware (Block 4)

The hardware responsible for code detection, named Dynamic Detection Hardware (DDH), is implemented as a 4-stage pipelined circuit to avoid the increasing on the original critical path of the SparcV8 processor. These four stages are the following:

- *Instruction Decode* (ID) –The instruction is broken into operation, source operands and target operand.
- *Dependence Verification* (DV) – on each data path´s column there is a bitmap responsible for storing the target operands of the already allocated instructions in the respective column, named as Write Bitmap (Figure 30). Thus, the source operands of each incoming instruction are compared to the target operands stored in the

bitmap of previously detected instructions to verify which column the current instruction should be allocated, according to their data dependencies. In this way, the dependence detection hardware spent only 8-bits width *xor* gate (supposing that the input context contain 8 registers) per each data path column.

To better explain the process, it is presented in the left side of Figure 30 an example of a code region detected by the DDH, in the right side of the same Figure is demonstrated its allocation inside of the reconfigurable data path. The first incoming instruction is always allocated at the highest functional unit of the leftmost data path column. In this process, the seventh bit of the write bitmap of such column is set since the R7 is the target operand of this instruction.

The dependence detection starts from the second instruction. In our example, the instruction number two reads R7 register that is written by previous instruction creating a read after write (RAW) dependence. The DDH detects it with a simple *xor* operation and allocates the instruction number two at the later column of instruction number one. In this process, the eighth bit of the second write bitmap is set since this instruction has the register R8 as the target operand.

There is a true dependence between the third and the second instruction, so the third instruction should be allocated at the third column setting the sixth bit of the write bitmap of such a column.

Otherwise, the instruction number four does not present any data dependence with instruction number two and three but it has a RAW dependence with instruction number one. In this way, it can be allocated at the later column of instruction number one executing in parallel with the instruction number two and temporally before of instruction number three. The write bitmap of the second column is updated, since the target operand of the instruction number four is the R1 register.

The instruction number five, a memory access operation, does not produce any data dependence to the previous instructions, so it is allocated at the leftmost load functional unit. As this kind of operation takes an entire processor cycle and covers three data path columns, the third bit of the write bitmap of the columns 1, 2 and 3 should be set to maintain the allocation consistency.

The instruction number six depends on the result of the previous memory access (instruction number five), so that instruction must be allocated at the fourth column. As its target operand is the R4 register, the correspondent bit of the fourth write bitmap is set.

The instruction number seven stores the value of the R4 register in the memory, as the previous instruction has this register as a target register, the instruction number seven should be allocated at a later column that provides a memory access functional unit available.

Finally, the instruction number eight does not have any data dependence with the previous instruction, so it is allocated at the first data path column. As this operation takes an entire processor cycle and covers three data path columns, the second bit (correspondent bit of its target operand) of the write bitmap of the columns 1, 2 and 3 are set.

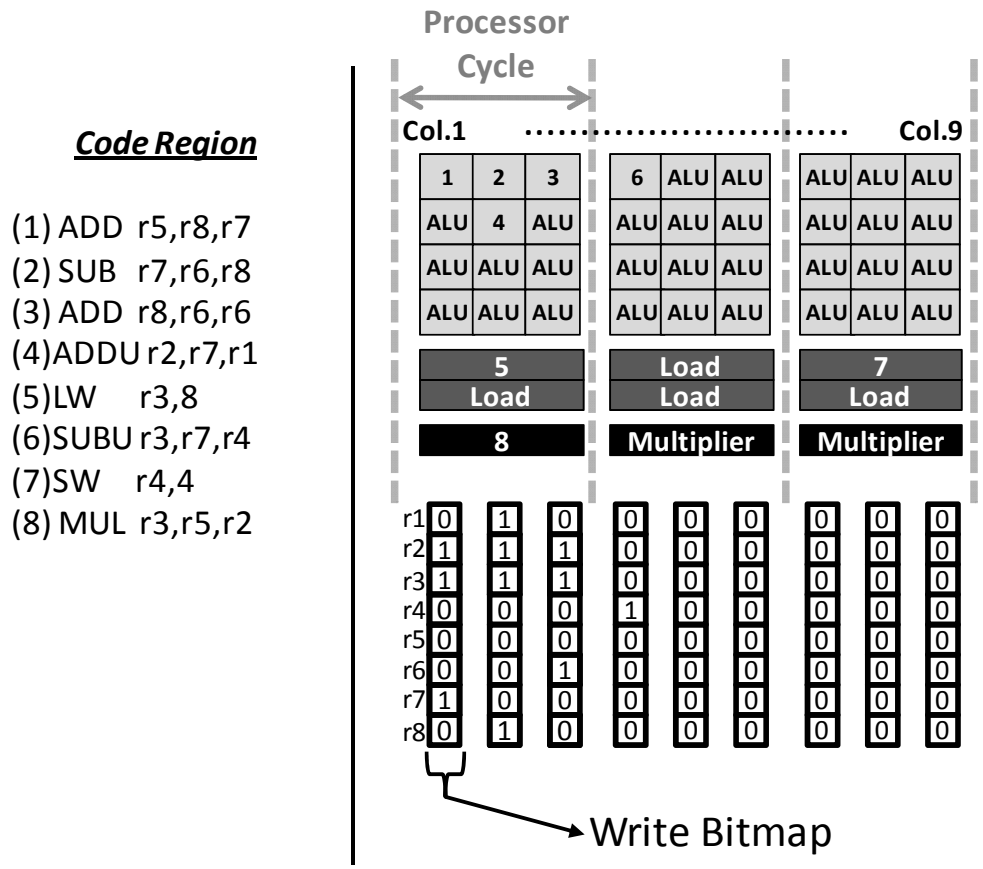


Figure 30. Example of an allocation of a code region inside of the data path

- *Resource Allocation* (RA) – In this stage, the data dependence is already solved and the correct data path column is known. Hence, the RA stage is responsible for verifying the resources availability in that column, linking the instruction operation to the correct type of functional unit. If there is no functional unit available at this column, the next column at the right side will be checked. This process is repeated until finding a free functional unit.
- *Update Tables* (UT) – This stage configures the interconnection components of the reconfigurable data path to feed that functional unit with the correct source operands from the input context and to write the result in the correct register of the output context. After that, the bitmaps and tables are updated and the configuration is finished: their configuration bits are sent to the reconfiguration memory and the address cache is updated with the memory address of the first instruction detected in the configuration.

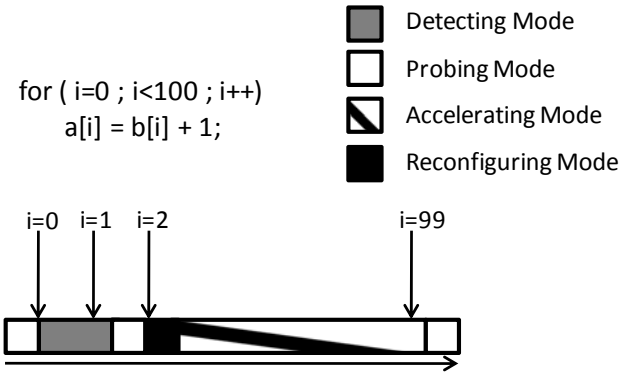


Figure 31. DAP acceleration process

Figure 31 shows a simple example of how a DAP could dynamically accelerate a code segment of a single thread. The DAP works in four modes: *probing*, *detecting*, *reconfiguring* and *accelerating*. The flow is as follows:

At the beginning of the time bar shown in Figure 31, the DAP searches for an already translated code segment to accelerate, comparing the address cache entries to the content of the program counter register. However, when the first loop iteration appears (i=0), the DDH detects that there is a new code segment to translate, and it changes to detecting mode.

In the *detecting* mode, concomitantly with the instruction execution in the SparcV8 pipeline, these instructions are also translated into a configuration by the DDH pipeline. The process does not stop when a branch instruction is found, since speculative execution is used. Thus, up to three basic blocks can compose a single configuration. When the second loop iteration is found (i=1), the DDH is still finishing the detection process that started when i=0. It takes few cycles to store the configuration bits into the reconfiguration memory, and to update the address cache with the memory address of the first detected instruction.

Then, when the first instruction of the third loop iteration comes to the fetch stage of the SparcV8 pipeline (i=2), the *probing* mode detects a valid configuration in the reconfiguration memory: the program counter content was found in the address cache entry**.**

After that, the DAP enters in the *reconfiguring* mode, where it feeds the reconfigurable data path with the necessary operands. For example, if 8 operands are needed, 4 cycles are necessary, since 2 read ports are available in the register file. In parallel with the operands fetch, the reconfiguration bits are also loaded from the reconfiguration memory. The reconfiguration memory is accessed on demand: at each clock cycle, only the necessary bits to configure a data path level are fetched, instead of fetching all the reconfiguration bits at once. This approach decreases the port width of the reconfiguration memory, which is one of the main sources of power consumption in memories (BERTICELLI LO, BECK, *et al.*, 2010).

Finally, the *accelerating* mode is activated and the next loop iterations (until the 99th) are efficiently executed, taking advantage of the reconfigurable logic.

Figure 32 summarizes, by an activity diagram, the whole DDH´s process to create a configuration. The first step is the execution support verification. If there is no compatible functional unit to execute such an operation (e.g. division), the configuration is finished and the next instruction is a candidate to start a new configuration. On the other hand, if there is support, the data dependency among previously allocated

instructions is verified (DV stage) and the correct functional unit within that column is defined. Then, the current configuration is sent to the reconfiguration memory.

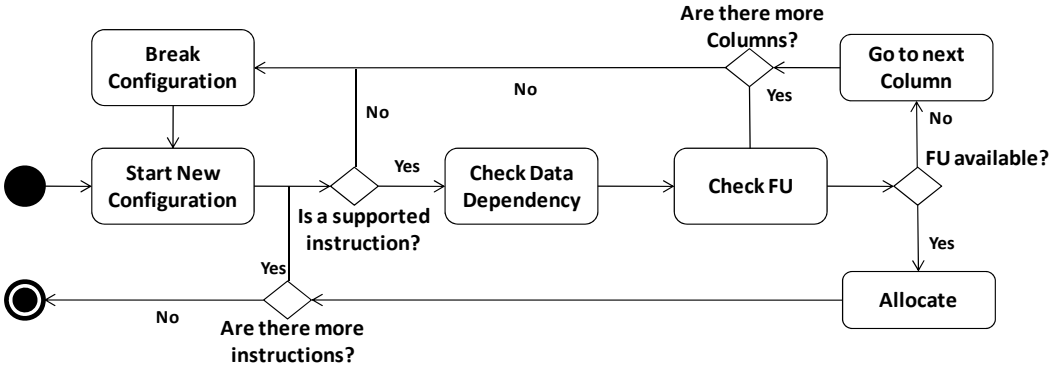


Figure 32. Activity Diagram of DIM process

4.1.4 Storage Components (Block 3)

Two storage components are part of the DAP acceleration process: address cache and reconfiguration memory. The address cache holds the memory address of the first instruction of every configuration built by the dynamic detection hardware. It is used to verify the existence of a configuration in the reconfiguration memory: an address cache hit indicates that a configuration was found. The address cache is implemented as a 4-way set associative table containing 64 entries. The reconfiguration memory stores the routing bits and the necessary information to fire a configuration, such as the input and output contexts and the immediate values.

Besides the two storage components explained before, the current DAP implementation has a private 32 KB 4-way set associative L1 data cache and a private 8 KB 4-way set associative L1 instruction cache. According to our experiments, the same hit rate is achieved by the SparcV8 in the DAP compared to the standalone SparcV8 using a quarter size of its 32KB L1 instruction cache. This happens because, as translated instructions are stored in the reconfiguration memory, the SparcV8 within the DAP has fewer memory accesses in the L1 instruction cache than the standalone SparcV8 processor. Thus, the impact of the additional area of the address cache and the reconfiguration memory in the DAP design is amortized.

5 RESULTS

This section presents the methodology and the results regarding the proposed approach. Considering the results, first we show the comparison of CReAMS with a multiprocessing system composed of in-order scalar SparcV8 processors, which demonstrates the potential of the proposed approach. After, the impact of inter-thread communication over both CReAMS and multiprocessing systems composed with different number of processors is verified. In addition, a subsection is dedicated to show the results where CReAMS is conceived as heterogeneous organization. Finally, we compared the adaptability of CReAMS on exploiting ILP and TLP with a multiprocessing system composed of 4-issue Out-Of-Order Superscalar SparcV8 processors.

5.1 Methodology

5.1.1 Benchmarks

In order to measure the performance and energy efficiency of CReAMS considering a heterogeneous environment, benchmarks from different suites were selected to cover a wide range of behaviors in terms of type (i.e. TLP and ILP) and degree of existing parallelism. The scope is to mimic future complex embedded applications that will run in portable devices. From the parallel suites (WOO, OHARA, *et al.*, 1995) (BIENIA, KUMAR, *et al.*, 2008) (DORTA, RODRIGUEZ, *et al.*, 2005), we have selected *md*, *jacobi* and *lu* that are, due to their nature, applications where TLP is dominant. Three SPEC OMPM2001 (DIXIT, 1993) applications (*apsi*, *equake* and *ammp*) were chosen to evaluate the CReAMS efficiency over originally single–threaded applications that were parallelized to take advantage of multiprocessing environments. Finally, we have selected four applications (*susan edges*, *susan smoothing*, *susan corners* and *patricia*) from the MiBench suite (GUTHAUS, RINGENBERG, *et al.*, 2002), which reflects a traditional embedded scenario.

The benchmarks were parallelized using OpenMP and POSIX threads. These libraries provide methods that discover, at run time, the number of processors of the underlying multiprocessor architecture, so they can take full advantage of the available resources even when the platform changes (e.g. processors are added), with no need for source code modifications and recompilation.

We have done a study over the selected applications to characterize their potential on obtaining performance improvement when TLP or ILP exploration is applied. The mean basic block size characteristic gives us some clues about the limits of instruction level parallelism that the selected applications provide. In addition, the percentage of the

entire application code that are executed in parallel, when multithreaded application environment is considered, is an important metric to obtain the actual thread level parallelism available in the applications. This metric is also called load balancing. Considering the mean size of the basic blocks, Table 3 shows that some applications such as *ammp*, *susan edges*, *susan corners* and *susan smoothing* provide a wide room for ILP exploitation. However, *equake*, *jacobi*, *patricia* and *apsi* do not show great potential for performance improvement when ILP exploitation is applied due to their small mean basic block sizes. In addition, the data provided in Table 3 demonstrate that the selected workload is very heterogeneous in terms of load balancing, it contains applications that have perfect load balancing, such as *susan smoothing*, *jacobi* and *md*, being suitable for TLP exploitation. However, TLP exploitation is not appropriate for some applications such as *equake*, *ammp* and *apsi* since their instructions are poorly load balanced even when few threads are considered (4 threads).

Table 3. Load balancing and mean basic block size of the selected applications

Benchmark	Mean BB size (#instr)	Load Balancing (%)			
		4 threads	8 threads	16 threads	64 threads
equake	4.80	18.49	10.32	5.10	0.92
apsi	6.86	17.45	9.20	4.80	1.10
ammp	14.56	27.35	12.40	6.20	1.09
susan_e	16.60	39.80	24.90	4.80	0.90
patricia	5.04	22.75	13.45	6.41	1.12
susan_c	17.36	67.58	49.18	34.94	12.50
susan_s	12.10	88.20	77.13	83.16	74.52
swaptions	5.92	99.00	99.00	99.00	98.00
blackscholes	4.83	99.00	99.00	99.00	98.00
md	6.51	95.04	83.24	88.92	89.87
jacobi	6.94	97.02	97.02	92.07	93.12
lu	8.32	81.20	56.77	29.35	7.03

5.1.2 Simulation Environment

For performance evaluation, we have used the scheme presented in Figure 33(a). In the following subsections we show the details about the tools and the whole simulation process.

5.1.2.1 Simics Simulator

The base platform is Simics (MAGNUSSON, CHRISTENSSON, *et al.*, 2002), an instruction level simulator. It was created a new Simics environment that comprises a Linux Ubuntu operating system running over a single SparcV8 processor. The applications shown above were compiled inside of this environment to allow the simulation process. As OpenMP and Pthreads provide procedures that allow the choice of the number of spawning at run-time, even with a single SparcV8 in the platform we get simulations with different number of threads. Simics produces a sequential trace that comprises the instructions and data accesses of all threads. As these instructions are mixed in a single and sequential trace, there are marks at the beginning and the end of each portion of code indicating what instructions belongs to which thread. The whole Simics process reflects the box “Thread Trace/Tracker” of the Figure 33(a).

5.1.2.2 Splitter.py

The simulator explained above sends the sequential trace for a python script, named as Splitter. This script is responsible for recognize the marks that informs what instructions belongs to which thread splitting these instructions in several buffers that contain instructions of each thread. In addition, this script is responsible for the dynamic thread scheduling shown in the Section 5.4.

5.1.2.3 Mkfifo

As both Simics and Splitter communicate in a producer-consumer way, a first-in first-out (FIFO) structure was inserted to achieve such communication. Mkfifo is a UNIX process that manages automatically a FIFO behavior. Thus, when a FIFO is full the producer (Simics) will stall and when a FIFO is empty the consumer (splitter.py) will stall allowing the proper simulation. In addition, the buffers referred in the splitter subsection are mkfifo processes, since their behaviors also reflects producer-consumer process.

5.1.2.4 Dynamic Adapted Processor

There is one timing simulator for each DAP (in the case of the CReAMS simulation) or for each standalone SparcV8 processor (when simulating the MPSparcV8). The DAP consumes the instructions sent by the splitter for its correspondent buffer. Thus, each DAP simulates the instructions of a unique thread (in case of static scheduling). The DAP simulator implements synchronization mechanisms, such as locks and barriers. This way, the time spent with blocking synchronization and memory transfers is precisely calculated. DAP holds in a plain text file partial results about the performance, communications among the threads, energy and power consumption of the instructions executed between each barrier. Each DAP has its own plain text file. In addition, the DAP simulator is available in C++ and comprises more than 5000 code lines.

5.1.2.5 Backward

This process is activated in the end of the simulation. As the plain text file of each DAP contains, for each thread, partial results about performance, energy and power consumption of each barrier, the backward is responsible for joining these results providing overall performance, energy and power consumption results of the whole application simulation.

5.1.3 VHDL descriptions

We have described the entire CReAMS architecture in VHDL, including the power management technique using Sleep Transistors. The MPSparcV8 VHDL description was obtained from (GAISLER, 2006). The Synopsys Design and Power Compiler tools, using a CMOS 90nm technology, were employed to synthesize the VHDL descriptions to standard cell and gather data about power, operating frequency, critical path and area. We use the data gathered from VHDL descriptions to calibrate the DAP cycle accurate simulators to obtain the overall energy/power consumption.

We assume for all experiments a perfect switching off/on for the processing elements of both CReAMS and MPSparcV8. It means that the processing elements do not consume energy in idle times. The power consumption of the reconfiguration memory,

the address cache, L1 and L2 memory cache were obtained with the CACTI 6.5 tool (WILTON e JOUPPI, 1996).

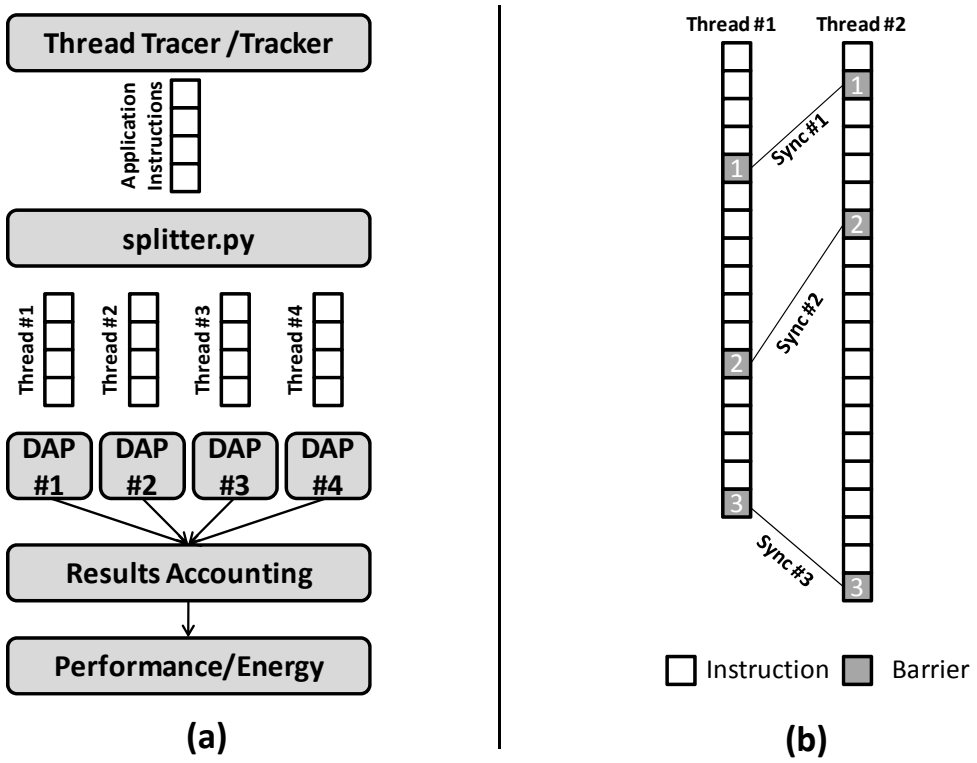


Figure 33. (a) Simulation Flow (b) How the synchronization process is done

5.1.4 How does the thread synchronization work?

Figure 33 (b) shows how the DAP cycle accurate simulators make the synchronization process among the running threads. Let us suppose a dual DAP, CReAMS platform executing two threads, where barriers are responsible for DAPs synchronization. In our cycle accurate simulator, barriers are represented by "magic instructions" that are inserted in the trace when software synchronization points are executed in the binary code. Figure 33 (b) depicts the execution of two threads, the white box represents a regular instruction and the gray box means a barrier in the trace. As can be seen, there are three synchronization points in the example of the Figure 33. Let us suppose that each regular instruction takes one DAP cycle to execute. Thread #1 takes four DAP cycles to reach the first barrier, while the thread #2 takes only one DAP cycle. After that, the thread #1 takes 6 cycles to reach the synchronization point two, while the thread #2 takes 4 DAP cycles. To get the third synchronization point, four and twelve cycles are taken for the thread #1 and thread #2, respectively. These partial performance results are stored by each DAP cycle-accurate simulator and joined when the execution of both threads ends. Thus, the simulation environment only takes into account, for the overall results, the longest thread execution time between barriers. In the example of the Figure 33(b), the longest execution time until the synchronization point one belongs to thread #1, which is also true for the second synchronization point. However, the thread #2 has the longest execution time in the third synchronization segment. The overall performance of the dual DAP CReAMS on executing the two threads depicted in Figure 33(b) is 22 cycles.

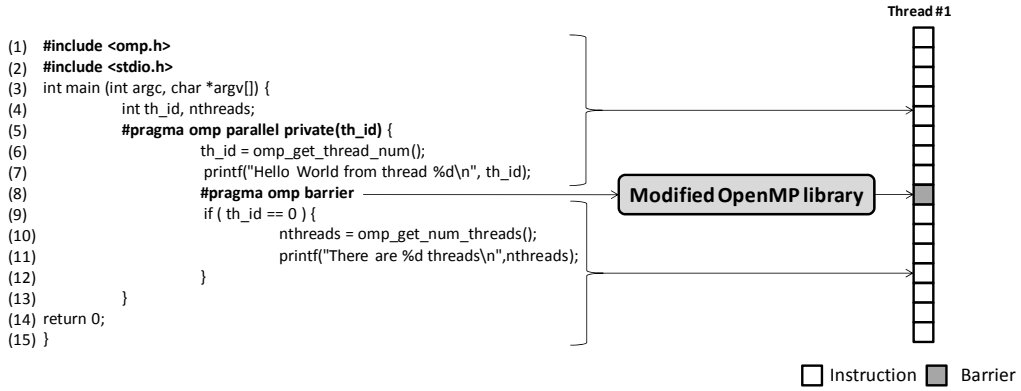


Figure 34. How the simulation handles synchronization from the software point of view

Figure 34 depicts how the simulation handles synchronization from the software point of view. A simple example of how to parallelize an application with OpenMP (MENON, 1998) is shown in the left side of the same Figure. The example works as following. First, when the parallel region starts (line 5), each thread gets its thread identification number. After, each thread prints a message on the screen showing its thread identification number (Line 7). Finally, the master thread, th_id=0, prints the amount of threads that have participated of the application execution. Before doing this last printing, a barrier was inserted (line 8) to avoid the printing of this last message by the master thread before the printing of the other thread identification messages. In this case, threads that already reached the barrier will stay halted until that all threads arrive in this synchronization point.

In the right side of the Figure 34 one can find how our simulation environment handles the software barrier explained above. When a certain thread reaches a barrier, the OpenMP library is accessed, so the procedure that processes the respective barrier is called. In this way, we have modified this procedure by inserting an assembly instruction in the C language, which we named as a “magic instruction”. When the DAP cycle accurate simulator reaches this “magic instruction” a barrier is recognized allowing the synchronization among threads to be accomplished.

It is important to point out that we consider speculative execution, it means that a single configuration can contain up to 3 basic blocks. The speculative policy is based on bimodal branch predictor [6]. For each level of the tree of basic blocks, the counter must achieve the maximum or minimum value (indicating the way of the branch). In addition, interruptions and traps are not considered, since our experiments do not run operating system code.

5.1.5 Organization of this Chapter

The rest of this section is divided on four subsections that explore the main topics discussed in this work. In the first subsection, we show the potential of CReAMS employment comparing it against a multiprocessing system composed of simple processor, named as MPSparcV8. In this part, we do not consider the communication overhead provided by the applications. In the second subsection, we built the modeling of the Network-on-Chip shown in Section 3.1.5 in both CReAMS and MPSparcV8. Thus, we are able to explore different latency of the communication infrastructure to highlight the need for mixed and adaptable parallelism exploitation when this aspect is considered. In the third section, we present the heterogeneous CReAMS, where is

encapsulated, in a single die, DAPs with different capabilities on exploiting instruction level parallelism. Finally, we present some studies comparing CReAMS against a multiprocessing system composed of 4-issue out-of-order superscalar SparcV8 processors.

5.2 The Potential of CReAMS

In all experiments of this subsection, we have compared CReAMS to a multiprocessing platform built by replication of standalone SparcV8 processors, named MPSparcV8. The configurations of DAP and standalone SparcV8 processor used in this evaluation are shown in Table 4. Both CReAMS and MPSparcV8 have an on-chip unified 512 KB 8-way set associative L2 shared cache.

Table 4. The configuration of both basic processors

	L1 I-Cache	L1 D-Cache	Pipeline	Frequency
DAP	8KB 4-Way	32KB 4-Way	5-Stages	600MHz
Standalone SparcV8	32KB 4-Way	32KB 4-Way	5-Stages	600Mhz

For the experiments, we have considered CReAMS setups with different number of DAPs (4, 8, 16 and 64 DAPs). The same was done with the MPSparcV8 system (4, 8, 16 and 64 SparcV8s). Each DAP has a reconfigurable data path (Block 1 of the Section 4.1.2) composed of 6 arithmetic and logic, 4 load/store and 2 multipliers units per column. The entire reconfigurable data path has 24 columns. A 48 KB reconfiguration memory, implemented as a DRAM memory, is able to store up to 64 configurations. The configurations are indexed by a 64-entries 4-way set associative Address Cache. As already discussed in Section 4.1.4, the DAP has one quarter less L1 I-Cache accesses than the SparcV8 processor. Thus, to avoid that the memory hierarchy bias the results in our favor, we reduced to 8KB the size of the DAP L1 I-Cache to achieve the same miss ratio in the main memory of the 32KB L1 I-Cache of the standalone SparcV8. In addition, in all experiments the number of spawning threads is equal to the number of DAPs available in the CReAMS platform.

In this subsection, as we wanted to present the potential of the proposed approach, the inter-thread communication latency is overlooked. Since the baseline processors of both multiprocessing systems could be connected through any communication infrastructure, in this first part of results we disregarded it to avoid that a particular approach bias our comparison. Next subsection presents results considering a mesh Network on Chip as a communication infrastructure with different latencies.

Table 5. (a) Area (um^2) of DAP and SparcV8 components

Processors Area (um2)	DAP	SparcV8
SparcV8 processor	247,615	247,615
Reconfigurable Data Path	3,298,602	-
32KB 4-Way L1 I-Cache	-	483,303
8KB 4-Way L1 I-Cache	130,980	-
32KB 4-Way L1 D-Cache	483,303	483,303
48 KB Reconfiguration Memory	688,255	-
128 Entries 4-Way Address Cache	19,473	-
Total	**4,868,228**	**1,214,221**

5.2.1 Considering the Same Chip Area

Table 5 shows the area, in um^2, of a standalone SparcV8 and a DAP, including memories. As it can be seen, the area of a DAP design is almost four times bigger than the standalone SparcV8 processor. For the first experiment, we wanted to compare the performance and energy of MPSparcV8 and CReAMS considering the same chip area. Hence, we have built two comparison schemes: "Same Area #1" (Table 6 (a)): composed of 4 DAPs and 16 SparcV8 processors; and "Same Area #2" (Table 6 (b)): composed of 16 DAPs and 64 SparcV8 processors.

Table 6. Area, in um^2, of : (a) Same Area Chip scheme #1 (b) Same Area Chip scheme #2

System Area (um2)	CReAMS 4 DAPs	MPSparcV8 16 SparcV8
Processors	19,472,912	19,427,537
512KB L2 Cache	15,118,865	15,118,865
Total	**34,591,777**	**34,546,402**

(a)

System Area (um2)	CReAMS 16 DAPs	MPSparcV8 64 SparcV8
Processors	77,891,647	77,710,146
512KB L2 Cache	15,118,865	15,118,865
Total	**93,010,512**	**92,829,011**

(b)

Table 7 shows the speedup obtained by MPSparcV8 and CReAMS over a standalone SparcV8 processor for all applications. The data of this table elucidate the heterogeneity of the selected applications in terms of instruction and thread level parallelism. Focusing only on MPSparcV8 speedup, where only TLP is exploited, one can conclude that *md*, *jacobi, susan_e* and *susan_s* contain massive thread level parallelism, since the speedup increases linearly as the number of cores increases. However, when the exploitation of instruction level parallelism is inserted by applying CReAMS, performance improvements are demonstrated in such an applications. These results reinforce the conclusion gathered from the analytical model in Section 3: even for high TLP-based applications, there is a need for finer grain parallelism exploitation to complement the TLP gains. When a huge amount of processors is available, MPSparcV8 presents small performance improvement on executing the rest of the applications, which supports CReAMS employment in a heterogeneous application environment where there is a diversity of thread and instruction level parallelism.

Table 8 shows the execution time, in milliseconds, when running all benchmarks on both platforms. Under the same table, one can find the indicative arrows comparing the setups with the same area. As it can be seen, considering the "Same Area #1" scheme, CReAMS outperforms MPSparcV8 in six benchmarks (*equake*, *apsi*, *ammp*, *susan edges*, *patricia* and *lu*), even considering applications that contain massive TLP, such as *lu* and *susan edges*. *ammp* is the one that benefits most from being executed on CReAMS: it presents an execution time 41% smaller than MPSparcV8. This application contains several synchronization points required to maintain data consistency among the execution threads. In addition, as can be seen in Table 3, these synchronization points are very load unbalanced, which affects the speedup when thread level parallelism exploitation becomes more aggressive. In contrast, this application provides a wider room for performance improvements when instruction level parallelism exploitation is applied. As shown in Table 3, *ammp* has mean basic block size of 14.56 instructions, which elucidates its data flow nature. In addition, this application also has few and well-defined kernels that contribute for the acceleration offered by DAP.

Table 7. Speedup provided by MPSparcV8 and CReAMS over a standalone single SparcV8 processor

	MPSparcV8				CReAMS			
	4SparcV8	8SparcV8	16SparcV8	64SparcV8	4DAPs	8DAPs	16DAPs	64DAPs
	Speedup							
equake	1.57	1.74	1.86	1.93	2.11	2.34	2.48	2.58
apsi	1.59	1.77	1.87	1.92	2.02	2.25	2.37	2.43
ammp	1.42	1.51	1.58	2.02	2.69	2.95	3.15	4.38
susan_e	2.15	2.69	3.06	3.39	3.38	4.67	5.76	6.84
patricia	1.24	1.47	1.28	1.23	1.94	2.87	2.49	2.39
susan_c	2.98	4.60	6.29	8.77	4.59	7.77	11.90	19.88
susan_s	3.93	7.81	14.98	48.93	7.69	15.04	28.65	92.84
md	3.91	7.54	14.06	38.22	10.91	20.22	34.83	74.91
jacobi	3.92	7.78	15.24	50.97	5.57	11.00	21.41	69.74
lu	3.08	4.86	4.18	3.51	6.28	10.55	8.30	6.85
Average	***2.58***	***4.18***	***6.44***	***16.09***	***4.72***	***7.97***	***12.13***	***28.28***

While *ammp* is the best application example for ILP exploitation and the worst for TLP, *jacobi* is the opposite. Its execution time decreases linearly as the number of processors increases when only TLP is applied, which means that their threads are completely synchronized which provides a perfect load balancing as shown in Table 3. Considering the instruction level parallelism exploitation, the small mean size of its basic blocks restricts the room for optimization when this strategy is applied. In this way, since the number of SparcV8 in the "Same Area #1" is four times bigger than DAP, the MPSparV8 design outperforms in 2.7 times CReAMS execution. However, considering all applications, CReAMS reduces the execution time by, on average, 36% in the "Same Area #1" scheme.

Considering the "Same Area #2" scheme, the performance and energy gains provided by CReAMS over MPSparcV8 are more evident than the "Same Area #1" setup. As the number of cores increases, the level of TLP tends to stagnate. In this case, CReAMS outperforms MPSparcV8 in seven benchmarks (*equake*, *apsi*, *ammp*, *susan_e*, *susan_c*, *patricia* and *lu*). In this case, *lu* execution on CReAMS is 1.3 times faster in comparison to MPSparcV8. However, as already explained, since *susan smoothing*, *md* and *jacobi* have perfect load balancing among their threads (Table 3), the MPSparcV8 is faster than CReAMS. However, even in cases of applications that have massive TLP with perfect balance, the employment of CReAMS can be considered satisfactory since, as will be show later, energy savings are obtained.

Table 8. Execution time of MPSparcV8 and CReAMS

	MPSparcV8				CReAMS			
	4SparcV8	8SparcV8	16SparcV8	64SparcV8	4DAPs	8DAPs	16DAPs	64DAPs
	Execution Time (in ms)							
equake	1501	1349	1267	1216	1113	1006	947	910
apsi	9070	8149	7693	7502	7111	6412	6066	5914
ammp	13632	12794	12206	9589	7197	6545	6139	4418
susan_e	218.2	174.4	153.0	138.3	138.6	100.3	81.4	68.5
patricia	138.0	116.2	134.0	139.8	88.3	59.7	68.8	71.7
susan_c	68.95	44.70	32.68	23.44	44.77	26.43	17.27	10.34
susan_s	815.8	410.3	214.0	65.5	416.8	213.2	111.9	34.5
md	1.001	0.519	0.278	0.102	0.359	0.194	0.112	0.052
jacobi	354.1	178.6	91.2	27.3	249.5	126.3	64.9	19.9
lu	0.741	0.470	0.545	0.650	0.363	0.216	0.275	0.333

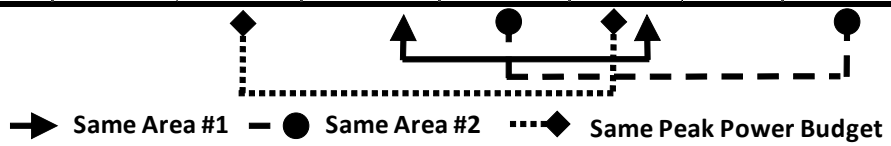


Table 9 and Table 10 show, respectively, the average power dissipation and energy consumption of MPSparcV8 and CReAMS architectures. As can be seen in Table 10, besides providing 41% lower execution time than MPSparcV8 in the "Same Area #1" on running *ammp*, CReAMS also spends 32% less energy. As already explained, MPSparcV8 is faster than CReAMS in 2.7 times when executing *jacobi* in the "Same Area #1" scheme but it spends more energy. In this case, as can be seen in Table 9, CReAMS provides a higher factor on reducing average power than MPSparcV8 provides in the execution time. In all applications, except *patricia*, CReAMS dissipates less average power than MPSparcV8 in the "Same Area #1" scheme. In this particular case, in *patricia* execution, the ILP exploitation process of CReAMS is not so energy efficient, there are a huge amount of configurations fetched from the reconfiguration memory. However, these configurations do not have significant performance gains on instruction level parallelism exploitation to dilute the energy spent to access the reconfiguration memory.

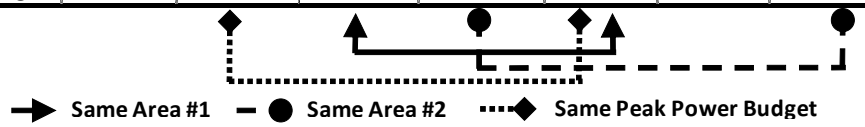
The main sources of CReAMS energy savings are:

- although more power is spent because of the DDH hardware and reconfigurable data path, total average power is reduced (refer to Table 7) since there are fewer memory accesses for instructions in the L1 instruction cache. Once they were translated to a data path configuration, they will reside in the reconfiguration memory. In addition, energy savings are also obtained in a single instruction memory access since, as already discussed, CReAMS has an instruction memory four times smaller than the MPSparcV8;
- the reconfiguration strategy: considering the loop example of the Figure 9, the data path is only reconfigured once to execute 98 loop iterations, thus avoiding several accesses to reconfiguration memory;

- shorter execution time: as CReAMS accelerates the code, it provides less energy consumption;
- the use of sleep transistors avoids that idle functional units spend unnecessary power.

Table 9. Average Power consumption of MPSparcV8 and CReAMS

	MPSparcV8				*CReAMS*			
	4SparcV8	***8SparcV8***	***16SparcV8***	***64SparcV8***	***4DAPs***	***8DAPs***	***16DAPs***	***64DAPs***
				Average Power (mW)				
equake	130.0	167.5	178.5	231.9	136.5	172.8	180.7	179.2
apsi	130.9	156.4	171.8	182.7	125.8	149.4	163.7	153.1
ammp	106.0	135.3	139.4	139.4	105.3	146.6	152.9	152.9
susan_e	167.5	234.9	280.0	319.9	179.9	283.3	366.7	451.6
patricia	152.6	145.0	141.4	135.7	188.7	217.0	215.4	206.9
susan_c	216.6	407.2	587.6	841.7	257.4	538.3	871.5	1498.2
susan_s	266.2	608.5	1305.6	4350.1	239.9	548.7	1170.7	3917.5
md	281.3	633.3	1266.1	3671.8	516.0	1116.4	2063.7	4749.0
jacobi	279.9	647.9	1363.3	4855.5	279.1	643.8	1345.9	4671.7
lu	225.7	413.2	426.3	507.5	312.6	608.8	572.5	683.7
Average	***195.7***	***354.9***	***586.0***	***1523.6***	***234.1***	***442.5***	***710.4***	***1666.4***



5.2.2 Considering the Power Budget

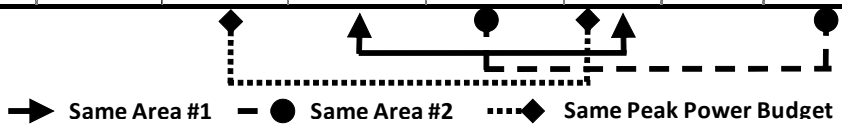
Current embedded systems have severe power constraints, since most of them are battery dependent. This way, we have also evaluated the performance and energy of both platforms considering a *peak power budget* of 3 Watts for both platforms, which is limit value foreseen for the coming year batteries (SEMICONDUCTORS, 2009). The peak power of the standalone SparcV8 is 385.14 mWatts, while the DAP consumes 699.33 mWatts. Therefore, we have compared the 8-SparcV8 MPSparcV8 against the 4-DAP CReAMS setups, since both reach nearly 3 Watts of peak power. The performance, power and energy results of both platforms that consider the *peak power budget* scheme are linked by arrows under Table 7, Table 8, Table 9 and Table 10.

As can be seen in Table 8, CReAMS outperforms MPSparcV8 in seven benchmarks (*equake, apsi, ammp, susan_e, patricia, jacobi* and *lu)*. As in the Same Area schemes, *ammp* is the application that benefits the most from CReAMS when the same peak power budget is considered, achieving 43% shorter execution time and consuming 30% less energy (Table 10) than the MPSparcV8. Although *susan smoothing* presents an increasing on the execution time by only 1%, CReAMS spends 59% less energy than the MPSparcV8 for that application. Energy savings are possible because, although the peak power is the same in both architectures, CReAMS consumes lower average power (refer to Table 9), mainly thanks to the use of sleep transistors to turn off idle functional units of the reconfigurable data path. In addition, its energy efficient method of join several ordinary instructions into a single data path configuration produces an

advantageous tradeoff in replacing several instruction memory accesses by a single reconfiguration memory access.

Table 10. Energy consumption of MPSparcV8 and CReAMS

	MPSparcV8				CReAMS			
	4SparcV8	8SparcV8	16SparcV8	64SparcV8	4DAPs	8DAPs	16DAPs	64DAPs
	Energy (mJ)							
equake	195.1	226.0	226.2	281.9	151.9	173.9	171.2	163.1
apsi	1187	1275	1322	1371	894	958	993	905
ammp	1445	1732	1701	1336	758	960	939	675
susan_e	36.55	40.98	42.83	44.25	24.94	28.41	29.85	30.95
patricia	21.05	16.85	18.95	18.97	16.67	12.94	14.81	14.83
susan_c	14.93	18.20	19.20	19.73	11.53	14.23	15.05	15.48
susan_s	217.1	249.7	279.5	285.0	100.0	117.0	131.0	135.3
md	0.282	0.329	0.353	0.376	0.185	0.216	0.232	0.248
jacobi	99.1	115.7	124.3	132.3	69.6	81.3	87.3	93.1
lu	0.167	0.194	0.232	0.330	0.113	0.132	0.157	0.227
Average	***321.7***	***367.4***	***373.4***	***349.0***	***202.7***	***234.6***	***238.1***	***203.4***



5.2.3 Energy-Delay Product

We correlate the energy and the performance results of both platforms to make more evident CReAMS efficiency. The energy-delay product is shown in Table 11. The schemes with the same area are linked under this table. As explained in Section 5.2.1, CReAMS on running *ammp* saves 32% of the energy consumption and improves in 41% the performance compared to the MPSparcV8 when the "Same Area #1" is considered. Thus, CReAMS provides a reduction in the energy-delay product of a factor of almost four on *ammp* execution.

The gains on energy and performance provided by the "Same Area #2" scheme are greater than the first comparison shown above since the performance improvements, when only TLP is explored, loses steam with the increasing on the number of processors. In this case, CReAMS outperforms MPSparcV8 in seven benchmarks (*equake*, *apsi*, *ammp*, *susan_e*, *susan_c*, *patricia* and *lu*). In this case, the execution of *lu* in the CReAMS is 57% faster than in the MPSparcV8 and consumes 50% less energy when the "Same Area #2" scheme is considered, which reflects in a energy delay product reduction in the factor of five. As can be seen, the average reduction achieved by CReAMS in energy-delay product is about 76% considering the same area chip schemes.

Table 11. Energy-Delay product of MPSparcV8 and CReAMS

	MPSparcV8				CReAMS			
	4SparcV8	8SparcV8	16SparcV8	64SparcV8	4DAPs	8DAPs	16DAPs	64DAPs
	Energy-Delay Product (J*1e-3s)							
equake	293	305	287	343	169	175	162	149
apsi	10768	10386	10169	10282	6361	6143	6023	5355
ammp	19698	22155	20764	12814	5452	6282	5761	2984
susan_e	7.97	7.15	6.55	6.12	3.46	2.85	2.43	2.12
patricia	2.91	1.96	2.54	2.65	1.47	0.77	1.02	1.06
susan_c	1.0296	0.8134	0.6274	0.4624	0.5160	0.3761	0.2600	0.1600
susan_s	177.2	102.5	59.8	18.7	41.7	24.9	14.7	4.7
md	0.000282	0.000171	0.000098	0.000039	0.000066	0.000042	0.000026	0.000013
jacobi	35.1	20.7	11.3	3.6	17.4	10.3	5.7	1.9
lu	0.000124	0.000091	0.000127	0.000215	0.000041	0.000028	0.000043	0.000076

As already explained, all but three (*susan smoothing*, *md* and *jacobi*) of the ten benchmarks present better performance in the CReAMS than MPSparcV8 considering the same chip area schemes. As can be noticed in Table 3, the threads of these three exceptions have a perfect load balancing which produces almost linear speedup with the increasing on the number of processors. However, even showing worst performance than MPSparcV8, CReAMS provides reductions in the energy-delay product on executing *susan smoothing* and *md* by 26% and 33%, respectively. Due to its massive TLP and perfect load balancing, *jacobi* is the only application where CReAMS does not provide gains neither in performance nor in energy considering the same chip area schemes. Using the energy-delay product evaluation, we demonstrated the CReAMS efficiency to dynamically adapt to the applications with different levels of parallelism, providing gains in performance or/and in energy consumption.

5.3 The impact of Inter-thread Communication

Up to now, the results shown in the previous subsection overlook the overhead on performance and energy consumption of the inter-thread communication. This subsection aims at including a Network-on-Chip infrastructure in both multiprocessing systems to demonstrate the impact of the communication latency over the results shown in the previous subsection. For that, we use the modeling of the mesh-NoC presented in the Section 3.1.5. We apply $AgvHops$ concept shown in the Section 3.1.5 to model the latency of a single inter-thread communication. Since threads can be arbitrarily allocated in the processors within the NoC, we applied the distributed and centralized approach to explore the impact of different data distribution in the infrastructure. In addition, to mimic a perfect communication approach, where there are only communications among the neighbors processors, we create the Ideal scenario, where the $AgvHops$ is equal to one. In addition, we assume that one hop takes one clock cycle and the NoC is running at the same frequency of the processors (600 MHz). The

average number of hops of the distributed and centralized schemes was calculated as follows:

Distributed $\begin{cases} AvgHops = \frac{2h}{3}, \ h \ even \\ AvgHops = 2\left(\frac{h}{3} - \frac{1}{3h}\right), \ \ h \ odd \end{cases}$

Centralized $AvgHops = \frac{k^2}{k+1}$

Ideal $AvgHops = 1$

where $h = \sqrt{N}$, being N the number of processor nodes.

Table 12 depicts the average number of hops for a single communication considering the methodology presented above.

Table 12. Average number of hops for different multiprocessing systems

Number of Hops					
	4 Proc	***8 Proc***	***16 Proc***	***32 Proc***	***64 Proc***
Distributed	1.33	1.88	2.66	3.77	5.33
Centralized	1.33	2.09	3.20	4.81	7.11
Ideal	1	1	1	1	1

5.3.1 Considering the Same Chip Area

In the previous subsection, we compare CReAMS with the MPSparcV8 by using two different scenarios: same area and same power budget. In this section, we follow the same strategy, but now we should take into account the area occupied by the NoC and its power consumption. Hence, we have used the same comparison schemes: "Same Area #1" (Table 13 (b)): composed of 4 DAPs and 16 SparcV8 processors; and "Same Area #2" (Table 13 (b)): composed of 16 DAPs and 64 SparcV8 processors. Small changes have been done in the DAP configuration aiming at respecting the same chip area. Now, each DAP is composed of 24 columns, each column has 5 arithmetic and logic units, 3 load/store units and 2 multipliers. A 92 KB reconfiguration memory, implemented as a DRAM memory, is able to store up to 128 configurations. The area of both CReAMS and MPSparcV8 components are shown in Table 13 (a).

Table 14 shows the execution time of both CReAMS and MPSparcV8 platforms when the designs, belong to the Same Area #1 scheme, are exposed to the communication latency presented above. As can be seen, even when considering all communication latency schemes (Ideal, Distributed and Centralized), CReAMS still outperforming MPSparcV8 in six applications (*equake*, *apsi*, *ammp*, *susan edges*, *patricia* and *lu*). However, the gains in performance showed by CReAMS over MPSparcV8 increase from the previous subsection. When no communication latency is considered, *ammp* presents 41% smaller execution time than MPSparcV8. Considering the Same Area #1 and the Centralized schemes, performance improvements of CReAMS over MPSparcV8 grows to 45%. For the rest of benchmarks referred above, the gains on performance provided by CReAMS increases from 13%, 8%, 10%, 50% to 36%, 27%, 13%, 60%, in *equake*, *apsi*, *susan edges* and *lu*, respectively.

Table 13. (a) Area of DAP and SParcV8 components (b) Same Area #1 Scheme (c) Same Area #2 scheme

Processors Area (um2)	DAP	SparcV8
SparcV8 processor	247,615	247,615
Reconfigurable Data Path	2,800,914	-
32KB 4-Way L1 I-Cache	-	483,303
8KB 4-Way L1 I-Cache	130,980	-
32KB 4-Way L1 D-Cache	483,303	483,303
92 KB Reconfiguration Memory	1,376,510	-
128 Entries 4-Way Address Cache	19,473	-
Total	**5,058,795**	**1,214,221**

(a)

System Area (um2)	CReAMS 4 DAPs	SparcV8 MP 16 SparcV8
Processors	20,235,180	19,427,537
512KB 8-Way L2 Cache	15,118,865	15,118,865
NoC Routers	113,884	455,536
Total	**35,467,929**	**35,001,938**

(b)

System Area (um2)	CReAMS 16 DAPs	SparcV8 MP 64 SparcV8
Processors	80,940,718	77,710,146
512KB 8-Way L2 Cache	15,118,865	15,118,865
NoC Routers	455,536	1,822,144
Total	**96,515,119**	**94,651,155**

(c)

Table 14. Execution time (in ms) considering the Same Area #1 scheme for CReAMS and MPSparcV8

	MPSparcV8			***CReAMS***		
	16SparcV8 Ideal	***16SparcV8 Distrib.***	***16SparcV8 Central.***	***4DAPs Ideal***	***4DAPs Distrib.***	***4DAPs Central.***
			Execution Time (in ms)			
equake	1366.09	1530.97	1583.73	1041.21	1079.33	1079.33
apsi	8350.71	9446.73	9797.45	7398.40	7667.47	7667.47
ammp	12897.11	14384.21	14860.09	7942.10	8246.90	8246.90
susan_e	159.88	171.54	175.27	149.56	154.65	154.65
patricia	196.83	212.15	217.05	109.08	112.76	112.76
susan_c	34.84	38.43	39.59	51.19	53.54	53.54
susan_s	237.29	276.03	288.43	492.79	519.68	519.68
md	0.30	0.34	0.36	0.44	0.47	0.47
jacobi	102.96	122.67	128.97	290.52	304.57	304.57
lu	0.61	0.71	0.74	0.43	0.46	0.46
Total Ex. Time	***23346.61***	***26183.78***	***27091.68***	***17475.73***	***18139.85***	***18139.85***

In cases where MPSparcV8 outperforms CReAMS (*susan corners*, *susan smoothing*, *md* and *jacobi*), the communication latency affect more the former than the latter. The increasing on the number of processors makes communication more significant in the execution time. When the Centralized communication latency is considered, the execution time of CReAMS approximates to MPSparcV8. This fact is more evident in *jacobi* where performance gains of MPSparcV8 over CReAMS fall from 2.76 to 2.32 only due to the communication overhead. *jacobi* spends 41.5% of its execution time in data communication when 16-Core MPSparcV8 is considered, while in the 4-DAP CReAMS this percentage falls to 22.6%. Thus, for some applications, as shown in the previous section, TLP exploitation can provide performance improvements when the number of processor increases, but the gains can be sorely affected by the communication overhead. On average, the performance improvement of CReAMS over

MPSparcV8 increases from 30% to 34% when the communication overhead is included in the Same Area #1 scheme.

Table 15. Execution time (in ms) considering the Same Area #2 scheme for CReAMS and MPSparcV8

	MPSparcV8			***CReAMS***		
	64SparcV8 Ideal	***64SparcV8 Distrib.***	***64SparcV8 Central.***	***16DAPs Ideal***	***16DAPs Distrib.***	***16DAPs Central.***
			Execution Time (in ms)			
equake	1309.44	1715.24	1881.72	860.26	1025.12	1077.87
apsi	8146.53	10940.09	12086.19	6277.66	7373.69	7724.42
ammp	12770.80	16750.42	18383.12	6718.75	8205.81	8681.68
susan_e	143.28	164.80	173.62	84.54	96.20	99.93
patricia	205.32	246.82	263.85	85.51	100.83	105.74
susan_c	24.35	28.47	30.29	19.30	22.90	24.05
susan_s	72.64	103.47	116.12	131.56	170.30	182.69
md	0.11	0.16	0.17	0.13	0.17	0.19
jacobi	32.07	53.05	61.66	75.70	95.41	101.72
lu	0.71	0.99	1.11	0.32	0.43	0.46
Total Ex. Time	***22705.25***	***30003.51***	***32997.85***	***14253.74***	***17090.85***	***17998.75***

Table 15 shows the execution time of the Same Area #2 scheme when the communication latency is considered for both CReAMS and MPSparcV8. Since the number of cores is larger in this comparison scheme, the communication overhead is more evident than in the Same Area #1. Therefore, CReAMS presents higher performance improvements over the MPSparcV8 in those applications where the TLP is restricted and the portion of the communication in the execution time is significant. Those are the cases of *equake*, *ammp*, *apsi*, *susan edges*, *susan corners* and *lu* where CReAMS provides lower execution time than MPparcV8 by 75%, 56%, 1.1 times, 73%, 25% and 1.4 times, respectively. *susan corners* is the most appealing case, in the Same Area #1 scheme MPSparcV8 outperforms CReaMS by 25%, but due to the communication latency of the 64 cores of MPSparcV8, CReAMS turn around this scenario and outperforms MPSparcV8 by 25%.

As can be noticed on comparing Table 14 and Table 15, the execution time of *equake*, *apsi*, *ammp*, *patricia* and *lu* grows when the number of processors increases from 16 to 64 when considering MPSparcV8. This fact supports the conclusions presented by the Analytical Model in the Section 3.1.6, for some application the gains in performance by increasing the number of processors (exploiting TLP) are smaller than the overhead caused by the inter-thread communication for both Distributed and Centralized schemes. Meaning that, there are some cases where is not worthwhile the increasing on the number of processors, it is worth investing on more aggressive ILP exploitation.

Table 16. Energy (in mJoules) considering the Same Area #1 for CReAMS and MPSparcV8

	MPSparcV8			***CReAMS***		
	16SparcV8 Ideal	***16SparcV8 Distrib.***	***16SparcV8 Central.***	***4DAPs Ideal***	***4DAPs Distrib.***	***4DAPs Central.***
			Energy (in mJoules)			
equake	226.72	227.54	227.80	123.73	123.85	123.85
apsi	1324.78	1329.83	1331.44	775.34	776.16	776.16
ammp	1690.02	1695.68	1697.49	658.87	659.68	659.68
susan_e	42.93	43.09	43.14	28.95	28.98	28.98
patricia	19.22	19.26	19.27	10.74	10.75	10.75
susan_c	19.25	19.34	19.37	12.91	12.93	12.93
susan_s	280.14	281.30	281.67	115.20	115.39	115.39
md	0.35	0.35	0.35	0.18	0.18	0.18
jacobi	124.63	125.20	125.39	80.06	80.15	80.15
lu	0.24	0.24	0.24	0.11	0.11	0.11
Total Energy	***3728.29***	***3741.83***	***3746.17***	***1806.08***	***1808.17***	***1808.17***

Table 16 and Table 17 show the energy consumption of CReAMS and MPSparcV8 when the Same Area #1 and #2 are considered, respectively. The gains on energy consumption provided by CReAMS over the MPSparcV8 are due to the same reasons mentioned in the previous subsection. The power overhead introduced by the Network-on-Chip is almost negligible in comparison with the power dissipated by the computation. While a router of the NoC spends 11.7 mWatts (MATOS, CONCATTO, *et al.*, 2011) to transfer one package from a certain input to the target output, a SparcV8 consumes 385.14 mWatts and a DAP consumes 696.75 mWatts. The power consumption of a DAP has a small variation from the previous subsection due to the changes on its configuration. As mentioned earlier, we decrease the number of ALUs and increase the number of slots in the reconfiguration memory aiming at respecting the Same Area schemes.

However, when the Same Area #1 scheme is considered, CReAMS increases the energy gains over the MPSparcV8 in comparison with the results where communication is overlooked. Those are the cases of *equake*, *apsi*, *ammp*, *patricia*, *md* and *lu* where the energy gains increases from 48.92%, 47.77%, 124.54%, 13.72%, 90.44%, 104.81% to 83.93%, 71.54%, 157.32%, 30.87%, 93.57%, 116.61%, respectively. CReAMS reduces the overall energy consumption by 52% on running all benchmarks in the Same Area #1 scheme.

Table 17. Energy (in mJoules) considering the Same Area #2 for CReAMS and MPSparcV8

	MPSparcV8			***CReAMS***		
	64SparcV8 Ideal	***64SparcV8 Distrib.***	***64SparcV8 Central.***	***16DAPs Ideal***	***16DAPs Distrib.***	***16DAPs Central.***
			Energy (in mJoules)			
equake	282.63	285.64	286.87	140.69	141.51	141.77
apsi	1373.18	1387.57	1393.47	863.41	868.45	870.07
ammp	1750.25	1766.24	1772.80	810.10	815.76	817.57
susan_e	44.36	44.83	45.03	27.92	28.08	28.13
patricia	19.25	19.35	19.39	9.64	9.67	9.68
susan_c	19.79	20.03	20.12	12.89	12.98	13.01
susan_s	285.78	288.99	290.30	112.90	114.06	114.43
md	0.38	0.39	0.39	0.17	0.17	0.17
jacobi	132.83	135.01	135.90	76.51	77.08	77.27
lu	0.33	0.34	0.34	0.12	0.12	0.12
Total Energy	***3908.79***	***3948.37***	***3964.61***	***2054.35***	***2067.89***	***2072.22***

The communication/computation ratio keeps constant with the increasing on the number of processor for the following benchmarks: *susan smoothing*, *patricia*, *md*, *equake* and *apsi*. This characteristic has a small increase, at most by 8%, in *ammp*, *jacobi* and falls, at most by 7%, in *susan corners*, *susan edges* and *lu*. However, as a single communication becomes more significant in our experiments by using different data traffic schemes (Ideal, Distributed and Centralized), the overall communication latency tends to produce a greater impact in the execution time. Moreover, as shown in Table 12, the impact of a single communication grows as well as the number of processors increases. Thus, the Same Area #2 scheme highlights the energy savings provided by CReAMS, since the energy spent by the routers becomes much more significant in 64-Processors MPSparcV8 than in 16-DAPs CReAMS. *jacobi* is the most affected by inter-thread communication, the computation/communication ratio in this benchmark is 17%, regardless of the number of processors. However, when the single communication latency increases (emulated by the Ideal, Distributed and Centralized scheme), this factor becomes significant in the overall execution time. For instance, considering the *jacobi* benchmark emulating the Distributed traffic scheme applied to the 64-Processor MPSparcV8, the inter-thread communication produces an overhead on its original execution time (disregarding communication) of 94% and considering the Centralized scheme by a factor of 1.2. On the other hand, when this application is running at 16-DAP CReAMS platform, these percentages fall to 48% and 58%, respectively. This means that, CReAMS spends 75% less energy than MPSparcV8, 51% of energy savings comes from faster computation and 24% comes from communication avoidance. On average, CReAMS achieves 96% of energy savings over MPSparcV8 when the Centralized Scheme and the Same Area #2 scheme are considered, 59% comes from faster computation and 37% from communication avoidance.

5.3.2 Considering the Power Budget

Similar to the previous subsection, a power budget is used to compare the performance and energy of both CReAMS and MPSparcV8, but now we consider the power consumption of the Network-on-Chip. We built setups for both platforms that spend 3 Watts. As the power consumption spent by a router (11.7 mW) is almost negligible in comparison with the processors, 696 mW by a DAP and 385 mW by a SparcV8, the number of processors of the previous comparison was maintained.

Table 18 shows the execution time of both CReAMS and MPSparcV8 when the power budget of 3 Watts is considered. As can be seen, comparing Table 10 and Table 18, the gains provided by CReAMS over MPSparcV8 increase when the inter-thread communication is introduced. The execution time of *equake* on both CReAMS and MPSparcV8 grows 25% due to the inter-thread communication. However, such increasing provides greater impact on the execution time of MPSparcV8 than CReAMS, since when the inter-thread communication latency is disregarded, the latter achieves 21% smaller execution time than the former. Thus, considering inter-thread communication, gains on performance shown by CReAMS over MPSparcV8 increase to 44%. On average, CReAMS outperforms MPSparcV8 by 29% when inter-thread communication is overlooked and by 33% when this characteristic is introduced in the Same Power Budget scheme.

Table 18. Execution time (in ms) of both CReAMS and MPSparcV8 considering a power budget

	MPSparcV8			***CReAMS***		
	8SparcV8 Ideal	***8SparcV8 Distrib.***	***8SparcV8 Central.***	***4DAPs Ideal***	***4DAPs Distrib.***	***4DAPs Central.***
			Execution Time (in ms)			
equake	1451.45	1542.00	1562.85	1041.21	1079.33	1079.33
apsi	8854.87	9480.46	9624.57	7398.40	7667.47	7667.47
ammp	13726.69	14552.44	14742.66	7942.10	8246.90	8246.90
susan_e	184.20	192.88	194.88	149.56	154.65	154.65
patricia	170.49	177.43	179.03	109.08	112.76	112.76
susan_c	48.49	51.86	52.63	51.19	53.54	53.54
susan_s	453.58	491.88	500.70	492.79	519.68	519.68
md	0.57	0.61	0.62	0.44	0.47	0.47
jacobi	200.50	219.88	224.35	290.52	304.57	304.57
lu	0.61	0.67	0.68	0.43	0.46	0.46
Total Ex. Time	***25091.46***	***26710.10***	***27082.97***	***17475.73***	***18139.85***	***18139.85***

Table 19 depicts the energy consumption of CReAMS and MPSparcV8 by applying the Same Power Budget scheme. For some benchmarks, the inter-thread communication enlarges the gains of CReAMS over MPSparcV8. Those cases are: *equake*, *apsi*, *ammp*, *patricia*, *md* and *lu* where CReAMS improves the energy saving over the MPSparcV8 from 48.77%, 42.50%, 128.57%, 1.07%, 77.64%, 31.10% to 83.28%, 64.98%, 163.55%, 16.16%, 80.16%, 102.71%, respectively. In other cases, reductions on the energy savings by CReAMS over the MPSparcV8 occurred when the communication is taken into account. However, it does not occur due to the inter-thread communication,

the reason for such reduction is the size of the reconfiguration memory that was modified. For the inter-thread communication comparison, a DAP has a reconfiguration memory twice larger than previous subsection where the results do not consider such characteristic. As those benchmarks produce a large amount of accesses in the reconfiguration memory, and such accesses does not produce the required performance improvement for fully amortizing the increasing on the power spent by the new reconfiguration cache size, CReAMS becomes less energy efficient. However, on average, when the inter-thread communication is considered, CReAMS spends 31% less energy than disregarding such characteristics, it means 51.5% less energy consumption than MPSparcV8.

Table 19. Energy (in mJoules) of both CReAMS and MPSparcV8 considering a power budget

	MPSparcV8			***CReAMS***		
	8SparcV8 Ideal	***8SparcV8 Distrib.***	***8SparcV8 Central.***	***4DAPs Ideal***	***4DAPs Distrib.***	***4DAPs Central.***
			Energy (in mJoules)			
equake	226.48	226.90	227.00	123.73	123.85	123.85
apsi	1277.42	1279.92	1280.49	775.34	776.16	776.16
ammp	1734.87	1737.86	1738.55	658.87	659.68	659.68
susan_e	41.06	41.14	41.16	28.95	28.98	28.98
patricia	17.08	17.10	17.10	10.74	10.75	10.75
susan_c	19.77	19.81	19.82	12.91	12.93	12.93
susan_s	250.26	250.77	250.88	115.20	115.39	115.39
md	0.33	0.33	0.33	0.18	0.18	0.18
jacobi	116.02	116.28	116.34	80.06	80.15	80.15
lu	0.22	0.22	0.22	0.11	0.11	0.11
Total Energy	***3683.52***	***3690.32***	***3691.89***	***1806.08***	***1808.17***	***1808.17***

5.3.3 Energy-Delay Product

Table 20 shows the energy-delay product of both CReaMS and MPSparcV8 platforms for the Same Area #2 scheme. As can be seen, the conclusions remain the same of the Table 11, when CReAMS outperforms MPSparcV8 in all but one application, due to the perfect load balancing shown by *jacobi*. However, when the communication is considered, CReAMS diminished the losses on the energy-delay product on running such an application from 36% to only 6%. The rest of applications also provide better energy-delay product that increases, on average, from 81% to 88% better than the MPSparcV8 for the Same Area #2 scheme.

Table 20. Energy-Delay product of MPSparcV8 and CReAMS considering the Same Area #2 scheme

	MPSparcV8			***CReAMS***		
	64SparcV8 Ideal	***64SparcV8 Distrib.***	***64SparcV8 Central.***	***16DAPs Ideal***	***16DAPs Distrib.***	***16DAPs Central.***
			EnergyDelay			
equake	370	490	540	121	145	153
apsi	11187	15180	16842	5420	6404	6721
ammp	22352	29585	32590	5443	6694	7098
susan_e	6.36	7.39	7.82	2.36	2.70	2.81
patricia	3.95	4.78	5.12	0.82	0.98	1.02
susan_c	0.48	0.57	0.61	0.25	0.30	0.31
susan_s	20.76	29.90	33.71	14.85	19.42	20.90
md	0.00004	0.00006	0.00007	0.00002	0.00003	0.00003
jacobi	4.26	7.16	8.38	5.79	7.35	7.86
lu	0.00024	0.00033	0.00037	0.00004	0.00005	0.00006
Total EDP	***33944.64***	***45305.14***	***50026.81***	***11008.12***	***13273.46***	***14004.35***

Table 21. Energy-Delay product of MPSparcV8 and CReAMS considering the power budget

	MPSparcV8			***CReAMS***		
	8SparcV8 Ideal	***8SparcV8 Distrib.***	***8SparcV8 Central.***	***4DAPs Ideal***	***4DAPs Distrib.***	***4DAPs Central.***
			EnergyDelay			
equake	329	350	355	129	134	134
apsi	11311	12134	12324	5736	5951	5951
ammp	23814	25290	25631	5233	5440	5440
susan_e	7.56	7.93	8.02	4.33	4.48	4.48
patricia	2.91	3.03	3.06	1.17	1.21	1.21
susan_c	0.96	1.03	1.04	0.66	0.69	0.69
susan_s	113.51	123.35	125.62	56.77	59.97	59.97
md	0.00019	0.00020	0.00020	0.00008	0.00009	0.00009
jacobi	23.26	25.57	26.10	23.26	24.41	24.41
lu	0.00014	0.00015	0.00015	0.00005	0.00005	0.00005
Total EDP	***35602.35***	***37935.06***	***38473.57***	***11184.04***	***11615.88***	***11615.88***

Table 21 shows the energy delay product when the power budget scheme is applied for both CReaMS and MPSparcV8 considering the inter thread communication latency. As can be seen on comparing Table 21 and Table 11 that inter-thread communication changes the conclusions over the energy delay product measurements. When the inter-thread communication is considered, CReAMS achieves better results in all benchmarks. As mentioned before, inter-thread communication sorely affects the execution time of *jacobi*, so CReAMS achieves better energy-delay product by 2% and 6% considering the distributed and centralized scheme, respectively. Hence, one can

conclude that for the whole spectrum of application behaviors, even for high parallel applications, with almost perfect load balancing, CReAMS achieves either better performance or less energy consumption, and produces lower energy-delay product than MPSparcV8 when a power budget of 3 Watts is applied.

5.4 Heterogeneous Organization CReAMS

In this subsection we show the results considering CReAMS as heterogeneous organization. First, the methodology used to gather data about heterogeneity is shown. After, the same schemes applied in the previous sections are employed here to compare the performance, area, power and energy consumption of homogeneous versus heterogeneous CReAMS.

5.4.1 Methodology

We build three DAP configurations', named as Small, Medium and Large, aiming at comparing the performance, area, energy and power consumption of both homogeneous and heterogeneous CReAMS, Table 22 (a) shows the number of components that composes each DAP. For the sake of the comparison, we refer to a homogeneous CReAMS design composed of small DAPs as HomoSmall CReAMS. The same terminology is used for Medium and Large CReAMS, so they are referred as HomoMedium and HomoLarge, respectively. For instance, a CReAMS composed of four small DAPs is referred as 4-HomoSmall CReAMS. Considering the exploitation of the design space of the heterogeneous organization, we encapsulate in the same chip the three DAP configurations to provide different levels of instruction level parallelism exploitation. Table 22 (b) shows the percentage of Small, Medium and Large DAPs that compose each heterogeneous CReAMS. For instance, 50% of the DAPs that compose the HeteroSmall CReAMS are SmallDAPS, 25% are MediumDAPs and 25% are LargeDAPs. Thus, If one would build a HeteroSmall CReAMS composed of eight DAPs, four of them would be SmallDAPs, two MediumDAPs and two LargeDAPs, and this configuration is named as 8-HeteroSmall CReAMS.

Table 22. (a) Different DAPs sizes (b) Percentage of DAPs that composes each Heterogeneous CReAMS

	SmallDAP	MediumDAP	LargeDAP
Number of Columns	9	15	24
Number of ALU per Column	3	4	5
Number of LD/ST per Column	2	2	3
Number of Multipliers per Column	1	2	2
Reconfiguration Memory Slots	32	64	128
Input Context Size	8	12	24

(a)

	%SmallDAP	%MediumDAP	%LargeDAP
HeteroSmall CReAMS	50	25	25
HeteroMedium CReAMS	25	50	25
HeteroLarge CReAMS	25	25	50

(b)

Table 23 (a) depicts the area occupied by each component that composes the three DAP configurations. As can be seen, the data path and the reconfiguration memory are larger components. Particularly, the size of the reconfiguration memory grows due to the increasing on the number of functional units (refer to Table 22(a)) and to the enlargement on the number of slots to store configurations. Table 23 (b) shows the area occupied by the homogeneous and heterogeneous CReAMS designs. As can be seen, the area of N-DAP HeteroLarge CReAMS is almost the same of 2N-DAP HomoSmall CReAMS, where N is the baseline amount of DAPs. Thus, these configurations produce the Same Area #1 scheme. With this scheme, we wanted to provide some clues about which multiprocessing system is worth, those that are composed of larger number of DAPs that slightly explore ILP in a homogeneous fashion, or those that are composed of

a smaller number of DAPs and explore the ILP in a heterogeneous fashion. On the other hand, we can notice in Table 23 (b) that the 2N-DAP HeteroSmall CReAMS has similar area of the N-DAP HomoLarge CReAMS, these configurations compose the Same Area #2 scheme. This scheme reflects quite the opposite of the Same Area #1, since now we wanted to verify if it is more efficient the coupling of a large number of DAPs with heterogeneous ILP exploitation or a small number of DAPs with an aggressive ILP exploitation. Figure 35 depicts an example of Same Area #1 and Same Area #2 comparison schemes with N equal to four.

Table 23. (a) Area of the components of the different DAP sizes (b) Area of the Homogeneous and Heterogeneous CReAMS setups

Processors Area (um2)	SmallDAP	MediumDAP	LargeDAP
SparcV8 processor	247,615	247,615	247,615
Reconfigurable Data Path	618,488	1,439,516	2,800,914
8KB 4-Way L1 I-D-Cache	614,283	614,283	614,283
Reconfiguration Memory	185,997	430,159	1,389,695
4-Way Address Cache	4,868	9,737	19,473
Total	**1,671,251**	**2,741,310**	**5,071,979**

(a)

Area (um2)	4 DAPs	8 DAPs	16 DAPs	32 DAPs	64 DAPs
HomoSmall	6,671,311	13,342,621	26,685,242	53,370,485	106,740,969
HomoMedium	11,118,070	22,236,140	44,472,281	88,944,561	177,889,122
HomoLarge	20,401,802	40,803,603	81,607,206	163,214,413	326,428,826
HeteroLarge	14,648,246	29,296,492	58,592,984	117,185,968	234,371,936
HeteroMedium	12,327,313	24,654,626	49,309,253	98,618,505	197,237,010
HeteroSmall	11,215,623	22,431,246	44,862,493	89,724,986	179,449,972

(b)

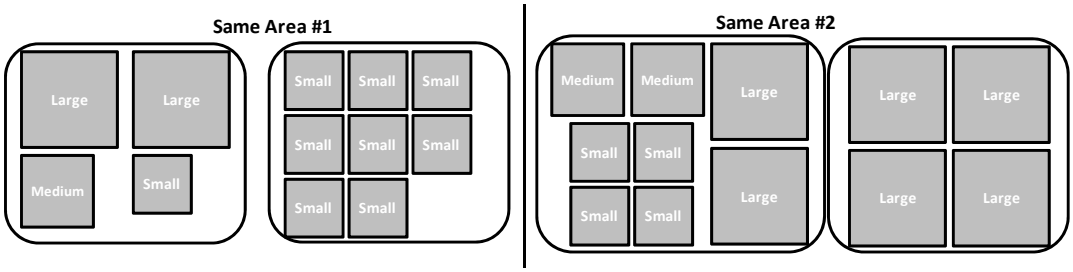


Figure 35. Example of Same Area #1 (left) and Same Area #2 (right) comparison schemes

Figure 36 shows the performance of N-DAP HeteroLarge CReAMS by applying the Same Area #1 scheme. The data provided in this figure is normalized to the performance of the 2N-DAP HomoSmall CReAMS. Thus, speedups and slowdown of 2N-DAP HomoSmall CReAMS over its respective homogeneous CReAMS is given by numbers greater and smaller than one, respectively. As can be seen in this Figure, the heterogeneous CReAMS outperforms the homogeneous platform in those applications where there is a lack of thread level parallelism and room for instruction level parallelism exploitation. *equake* and *apsi*, which present such behaviors, are the most benefited from heterogeneity by achieving 22% and 10% of performance improvement considering the same area designs and N equal to four. Moreover, as N increases such improvement increases as well due to the growth on the number of Large DAPs that explore ILP aggressively. When N is equal to 32, the heterogeneous outperforms the homogeneous CReAMS by 48% and 27% on running *equake* and *apsi*, respectively.

On the other hand, applications that provide a massive thread level parallelism do not perform better in heterogeneous CReAMS, since to respect the same area schemes, such designs contain half the number of DAPs. Heterogenous CReAMS shows a slowdown of 53% over the homogeneous platform on running *jacobi*, *susan smoothing* and *md*. As already explained before, these applications have perfect load balancing and linearly improve their performance with the increasing on the number of DAPs. It is difficult to reach linear performance improvement by applying any instruction level parallelism exploitation and providing an area overhead by a factor of two.

susan corners shows a irregular behavior on running at heterogeneous CReAMS organization. Up to N equal to sixteen, homogeneous outperforms heterogeneous CReAMS when the Same Area #1 scheme is applied. However, when N is equal to 32 the heterogeneity achieves 20% of performance improvement over the homogeneous CReAMS. The sudden drop of the load balancing that occurs from 32 to 64 threads gives this gain. Thus, as some of the 64-Core HomoSmall CReAMS become idle and the working DAPs do not push up the performance by exploiting ILP, the 32-Core HeteroLarge achieves better execution time. The performance of *patricia* and *lu* follows the same behavior of *susan corners*, but the sudden drop of the load balancing starts with smaller number of threads. When N is equal to 8, heterogeneous organization shows a performance improvement of 27% and 23% over the homogeneous CReAMS on running *lu* and *patricia*. The gains on performance provided by heterogeneity increase to 58% in *lu* when N increases to 32, since its load balancing keeps dropping in the same pace as the number of DAPs grows. However, quite the opposite occurs in *patricia* execution, the gains provided by heterogeneous CReAMS falls up to 5% when N is equal to 32. The performance improvement provided by only exploiting TLP is irregular in this benchmark, when the number of threads increases from 4 to 8 performance gains are shown. Thus, when 4-DAPs HeteroLarge is compared to 8-Core HomoSmall, the TLP exploitation of 8 DAPs provides larger gains than ILP. However, when the number of threads becomes in between 8 and 64, a significant load unbalancing occurs and the execution time increases in comparison with 8 threads. The losses on performance occur at the same pace as the number of threads grows. Thus, TLP loses steam and heterogeneous ILP provided by CReAMS achieves larger performance improvement.

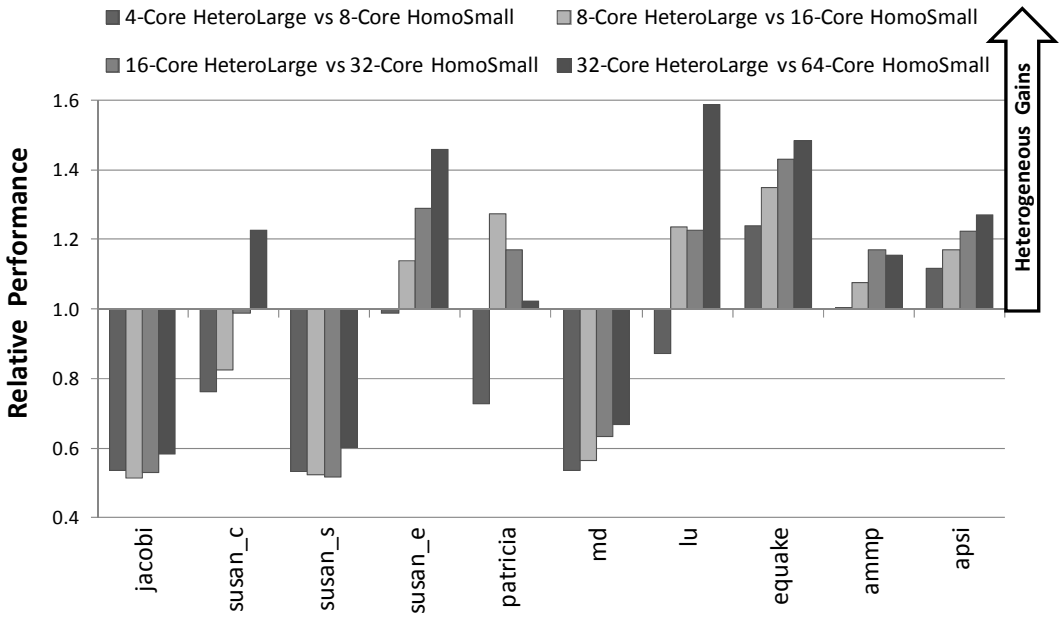


Figure 36. Relative Performance of HeteroLarge over HomoSmall CReAMS considering the SameArea #1 scheme

Figure 37 shows the relative energy consumption of N-DAP HeteroLarge over 2N-DAP HomoSmall. As can be seen, heterogeneous setups spend less energy than homogeneous CReAMS. Table 24 (a) shows the Thermal Design Power (TDP) of the

different DAPs configurations. Table 24 (b) presents the TDP of CReAMS conceived as heterogeneous and homogeneous organizations. Considering the Same Area #1 scheme, Table 24 (b) shows that the TDP of the N-DAP HeteroLarge CReAMS is 78% lower than 2N-DAP HomoSmall design. Thus, even showing 45% larger execution time than the homogenous CReAMS when running *jacobi*, the heterogeneous design achieves 5% of energy savings due to the lower power consumption.

Table 24. (a) Thermal Design Power (TDP) of DAP configurations (b) TDP of heterogeneous and homogeneous CReAMS

DAP	TDP (mWatts)
Small	532.86
Medium	604.78
Large	696.75

(a)

TDP (mWatts)	4 DAPs	8 DAPs	16 DAPs	32 DAPs	64 DAPs
HomoSmall	2,131	4,263	8,526	17,051	34,103
HomoMedium	2,419	4,838	9,676	19,353	38,706
HomoLarge	2,787	5,574	11,148	22,296	44,592
HeteroLarge	2,531	5,062	10,125	20,249	40,498
HeteroMedium	2,439	4,878	9,757	19,513	39,027
HeteroSmall	2,367	4,734	9,469	18,938	37,876

(b)

As shown in Section 5.2.1, CReAMS shows energy savings by avoiding fetches of instructions in the main memory. Thus, as DAP becomes larger, more instructions are packaged in a single configuration and more energy savings is produced. *lu* is the most benefited from this characteristic, when N is equal to 4, the heterogeneous design spend 13% less energy than homogeneous CReAMS. Besides showing 59% lower execution time on running *lu* when N grows to 32, the heterogeneous organization also reduces by 37% the energy consumption in comparison with homogeneous CReAMS. On the other hand, *md* is the most harmed application by using heterogeneous CReAMS, its execution time increases 44%. However, 22% of energy savings are shown by replacing homogeneous CReAMS for heterogeneous considering designs with same areas, which reinforces the feasibility of the employment of heterogeneous CReAMS for embedded domain. For the Same Area #1 scheme and considering all applications, N-DAP HeteroLarge CReAMS outperforms 2N-DAP Homogeneous CReAMS by 4% and spends 5% less energy consumption.

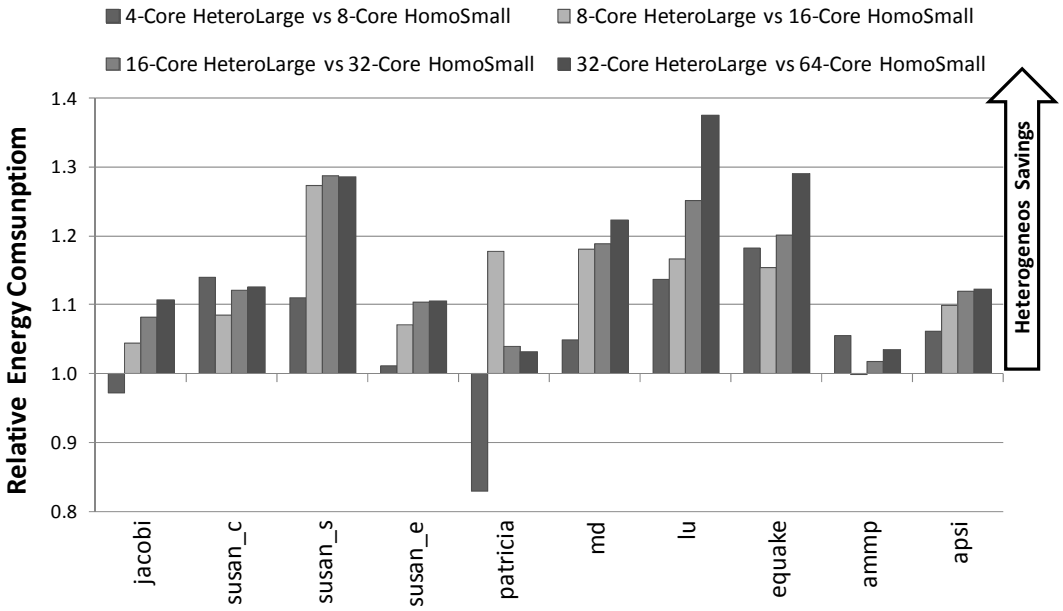


Figure 37. Relative Energy Consumption of HeteroLarge over HomoSmall CReAMS considering the Same Area #1 scheme

Figure 38 shows the relative performance of the 2N-DAP HeteroSmall over N-DAP HomoLarge reflecting the Same Area #2 scheme. This scheme proposes a reverse comparison to the one that was done in the Same Area #1. Now, we wanted to verify the feasibility of heterogeneity where most of DAPs are not be able to explore aggressively instruction level parallelism (refer to Figure 35). Thus, in comparison with the Same Area #1 scheme, here we diminished the capability of exploiting ILP and increased the TLP in the heterogeneous CReAMS. On the other hand, we decrease the TLP and increase the ILP in the homogeneous CReAMS target to the comparison. As can be seen in Figure 38, the TLP oriented applications, such as *jacobi* and *susan smoothing*, where in the Same Area #1 scheme are not benefit from heterogeneity, here show the opposite. The 2N-DAP HeteroSmall outperforms N-DAP HomoLarge in these applications, since the larger number of DAPs in the heterogeneous design provides better performance than the aggressive ILP exploitation available in the HomoLarge.

Applications from SPEC OMP2001 (*apsi*, *equake* and *ammp*) present neither TLP nor ILP available in a massive degree. The behavior of SPEC OMP2011 applications represents the huge amount of sequential application already available in the market. Therefore, these applications reflect the difficult work to split already written code in threads, since they should be parallelized in threads over an existing sequential code. The heterogeneity provides better performance on those applications in both Same Area #1 and Same Area #2 schemes. It means that neither few cores with aggressive ILP exploitation nor many cores with tiny ILP exploitation in a homogeneous fashion are suitable for these applications. They demand a heterogeneous organization to give aggressive ILP exploitation for those threads that request it, and some cores to execute the parallel portion of their codes.

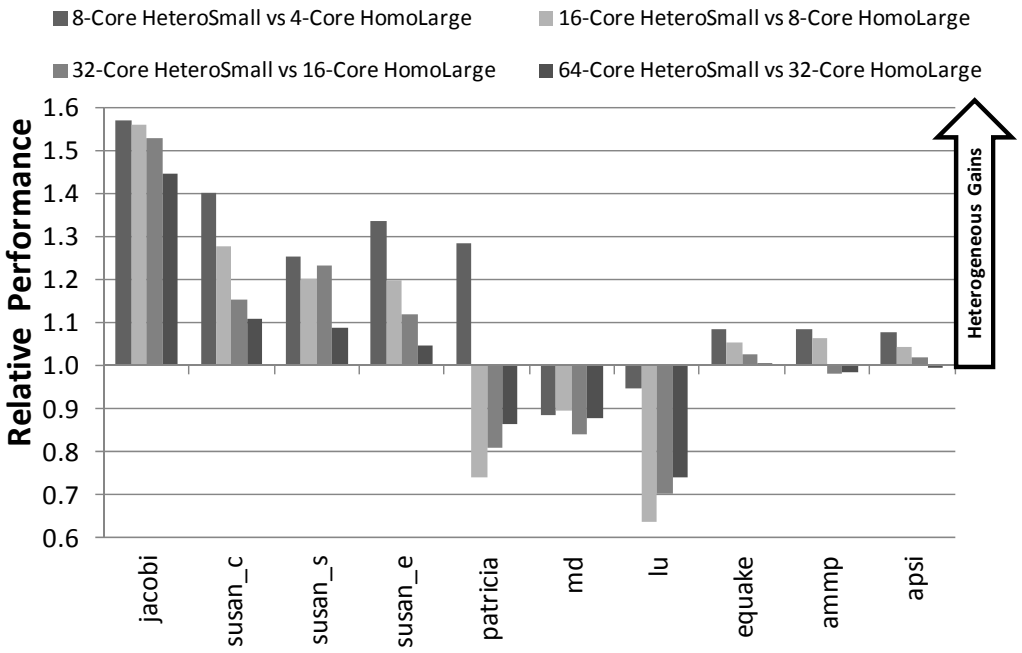


Figure 38. Relative Performance of HeteroLarge over HomoSmall CReAMS considering the Same Area #2 scheme

Figure 39 depicts the energy consumption of both 2N-DAP HeteroSmall and N-DAP HomoLarge. Comparing this Figure with Figure 37, one can notice that the energy savings provided by the heterogeneity in the Same Area #1 scheme overturn in losses when the Same Area #2 scheme is considered. The TDP of 2N-DAP HeteroSmall is 69% bigger than the N-DAP HomoLarge (refer to Table 24 (b)). Thus, even achieving, on average, an execution time 52% lower on running *jacobi*, the heterogeneous CReAMS spends 10% more energy than the homogeneous design due to its bigger TDP. For all applications, the heterogeneous CReAMS design decreases 4% of the execution time dissipating 7% more energy.

Figure 40 shows the relative Energy-Delay Product of Heterogeneous CReAMS organization over the homogeneous design when the Same Area #1 and #2 are considered. Analyzing this Figure, one can notice that the heterogeneity of the Same Area #1 scheme provides gains in the EDP metric. For instance, the heterogeneity reduces the EDP by a factor of 2.2 and 1.9 on running of *lu* and *equake* when the Same Area #1 scheme is considered. On average, considering all setups and applications, the heterogeneous CReAMS reduces 10% the EDP considering the Same Area #1 scheme. However, in the Same Area #2 scheme the heterogeneity increases the EDP in 4%. Thus, one can conclude that, for the applications considered in our experiments, it is mandatory building CReAMS as heterogeneous organization to achieve better energy delay product, but most of DAPs should have an aggressive ILP exploitation (Same Area #1).

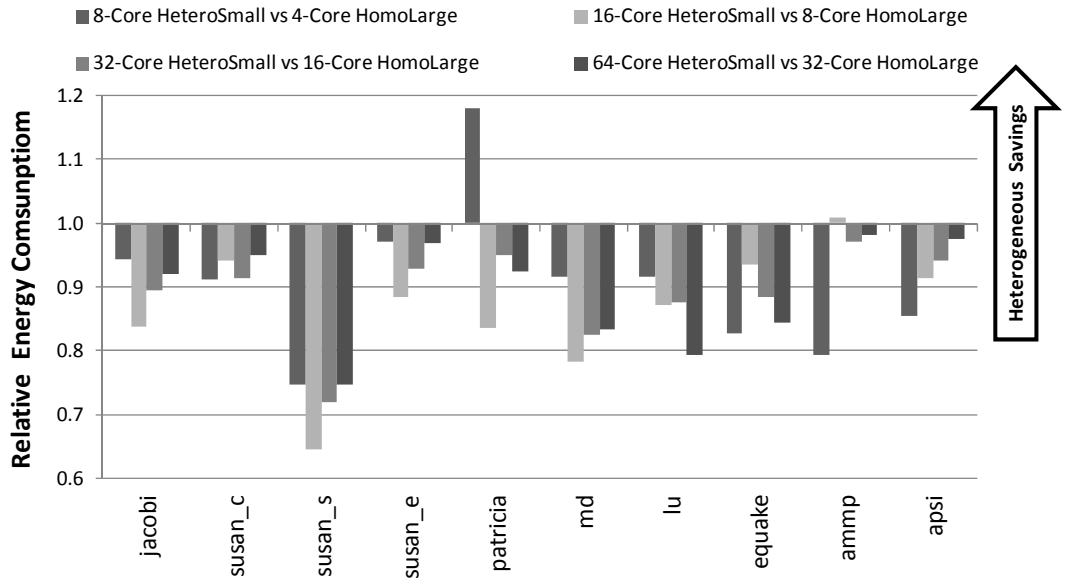


Figure 39. Relative Energy Consumption of HeteroLarge over HomoSmall CReAMS considering the Same Area #2 scheme

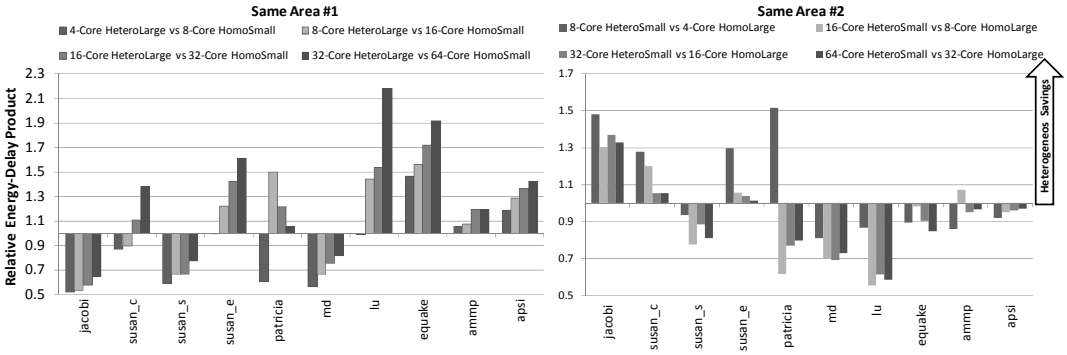


Figure 40. Relative Energy-Delay Product of HeteroLarge over HomoSmall CReAMS considering the Same Area #1 and #2 schemes

Up to now, all results shown in this subsection consider that the threads are statically schedule in the DAPs meaning that once a thread starts running in a DAP, that thread will end its execution in the same DAP. However, such an approach is not suitable when heterogeneous organization is applied, since a certain thread that needs greater ILP exploitation could be scheduled in a Small DAP, which could affect the overall performance. Dynamic scheduling is employed to verify if the performance degradation occurred on applying heterogeneity shown in Figure 36 and Figure 38 is due to the poorly thread scheduling performed. The lightweight context switch when a changing on the scheduling occurs is an advantage provided by CReAMS. The configurations stored in the reconfiguration memory of the DAP do not migrate, since threads share the same memory address space and the configurations are indexed by the memory address of the first instruction, they can take advantage of the configurations built by other threads.

The algorithm developed for the scheduling contains instruction counters for each thread and decides, based on this data, the best thread scheduling. The changing on the scheduling is performed when all threads reach a certain barrier. In this moment, the algorithm schedules, to those DAPs that have aggressive ILP exploitation, threads which ran the largest number of instructions since the last barrier. The main goal of this exploitation is not employ the best scheduling algorithm, but it is verify if performance losses provided by the heterogeneous CReAMS are due to wrong thread scheduling.

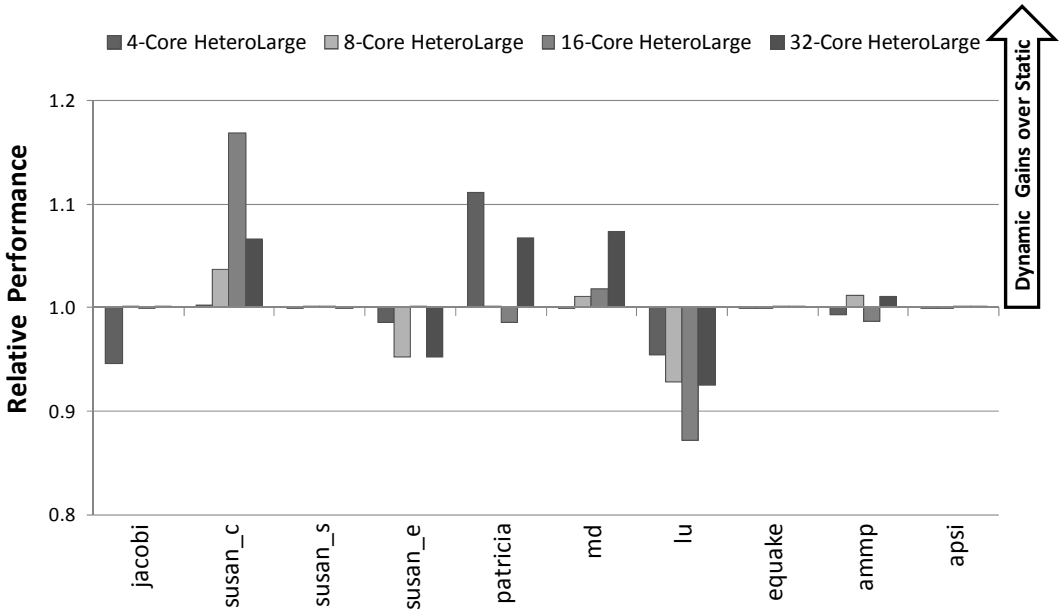


Figure 41. Relative Speedup of Dynamic over the Static Thread Scheduling considering the Same Area #1 scheme

Figure 41 shows the relative performance of the dynamic scheduling scheme in comparison with the static approach when the Same Area #1 is considered. The main purpose of using OpenMP is parallelizing loop iterations. In general, when OpenMp is applied, each loop iteration becomes a thread. For instance, let us assume a certain application where loops are completely data flow, it means that no branch is used in the body of the loop, all threads will execute the same code. The implementation of the FIR filter (refer to Figure 27) reflects this behavior. In this scenario, if the DAP that presents lowest capability of exploiting ILP already achieves the maximum gains on exploiting the parallelism of the loop body, dynamic scheduling would not produce any performance improvement. As can be seen, *susan smoothing*, *equake* and *apsi* does not present any changes on their performance on applying the dynamic scheduling algorithm due to the fact mentioned before.

On the other hand, one can notice that *susan corners* achieves performance improvement by applying dynamically scheduling. Analyzing Figure 36, one can conclude that there is no reason for 16-DAP HeteroLarge provides worst performance than 32-DAP HomoSmall on running *susan corners*, since as shown in Table 3, this application has large room for ILP exploitation and small room for TLP exploitation. Figure 41 shows that the reason for the performance losses demonstrated by heterogeneous CReAMS on running this application (Figure 36) are due to wrong thread scheduling. Performance losses of 3% on running this application in 8-DAP

HeteroLarge CReAMS become performance improvement of 13% over 16-DAP HomoSmall CReAMS when the dynamic scheduling is applied. The slowdown provided by 32-DAP HeteroLarge of 36% on running *md* is reduced to 30% by only scheduling dynamically its threads. As the algorithm used in dynamic scheduling is naïve, it produces performance losses for some applications, such as *lu* and *susan edges*, which emphasize our hypothesis to the need for a dynamic scheduling when using heterogeneous organizations together with a mixed application workload.

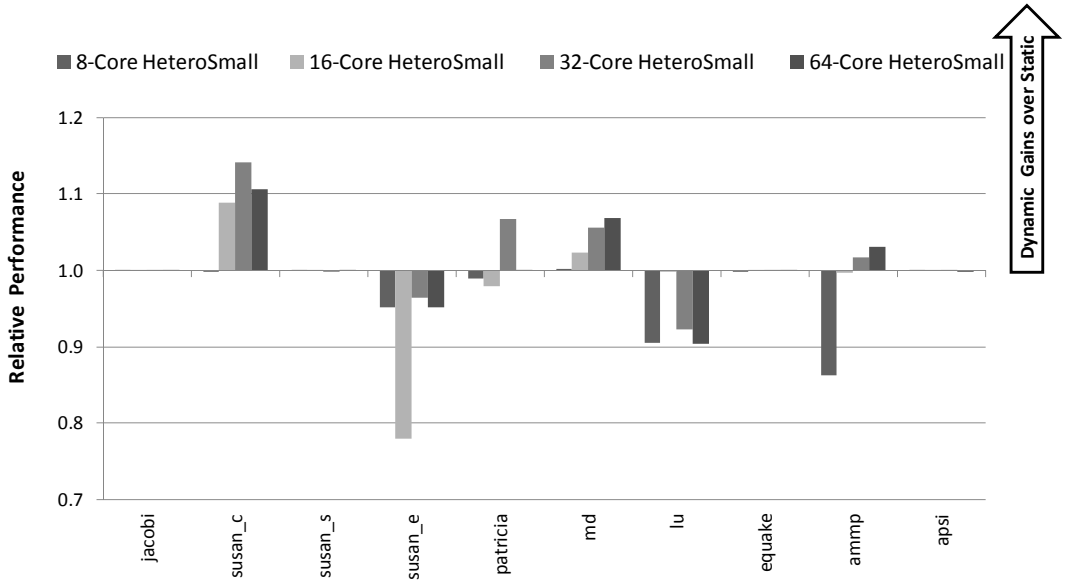


Figure 42. Relative Speedup of Dynamic over the Static Thread Scheduling considering the Same Area #2 scheme

Figure 42 shows the impact of dynamic scheduling in the Same Area #2 scheme. As occurred in the Same Area #1, dynamic scheduling does not produce any changes for some applications due to the same reasons mentioned before. *susan corners* and *md* benefit from the dynamic scheduling also in the Same Area #2 reinforcing to the need for a dynamic scheduling. Which also support our belief is the behavior of *lu*, *ammp* and *susan edges* in this Figure, as the dynamic algorithm produce some wrong scheduling, the performance of these applications are sorely affected since their threads are very heterogeneous in terms of need for ILP exploitation.

5.5 CReAMS versus Out-Of-Order Superscalar SparcV8

Up to now, we have compared CReAMS against single-issue SparcV8 multiprocessing systems. Here, we wanted to verify the feasibility of CReAMS in comparison with a state-of-the-art processor in terms of ILP exploitation. Therefore, we coupled a simulator of an Out-Of-Order SparcV8 processor (KAVVADIAS, 2001) in the framework presented in Figure 33(a). For that, some modifications in the way that the instructions are decoded and the results are generated were necessary to couple such simulator in our framework.

As there is no hardware implementation of a 4-issue OOO SparcV8 processor available in the market, we gathered data about performance in our framework based on the organization of a MIPS R10000 processors due to its similarities with SparcV8 ISA.

Besides the issue, other characteristics, such as width of reservation stations and reorder buffers, are modeled in the simulator based on the organization of MIPS R10000. To gather data about power consumption, we also apply data from the MIPS R10000. As such processor is only available at CMOS 0.35 um, a technology scaling to 90nm (the technology used on CReAMS implementation) is necessary to make a fair comparison. According to (HEINRICH, 1997), the 4-issue Out-of-Order MIPS R10000, implemented in CMOS 3.5 um, consumes 30 Watts at 3.3 Volts operating at 250 MHz. CReAMS was synthesized in a CMOS 90nm supplied by 1.0 Volt running at 600MHz. Let us scale the CMOS 3.5um to CMOS 90nm applying the power supply of the newest technology. As the power dissipated by certain circuit can be written as

$$P \approx f * C * V^2,$$

Let us scaling the power supply from 3.3 Volts to 1.0 Volts, the resulting power consumption is 2.75 Watts operating at 250MHz. Hence, to normalize the power consumption of both CReAMS and 4-issue OOO SparcV8 superscalar processor to 600 MHz, a factor of 2.4 must be applied to OOO SparcV8 processor, resulting on 6.61 Watts. Table 25 shows the TDP of different multiprocessing systems composed of 4-issue out-of-order SparcV8 processors.

Table 25. TDP of 4-issue Out-Of-Order SparcV8 multiprocessing system

TDP (mWatts)	4-OOO	8-OOO	16-OOO	32-OOO	64-OOO
4-issue	26,494	52,987	105,975	211,950	423,900

For comparison purpose, we create two different scenarios that consider power budget of 27 Watts and 53 Watts. As can be seen in Table 25 and Table 24(b), the power consumption of both 4-OOO SparcV8 superscalar processors and 32-DAP HomoLarge are similar, so these configurations compose the Power Budget #1 scheme. For the Power Budget #2 scheme, we compare 8-OOO SparcV8 superscalar processors with 64-DAP HomoLarge CReAMS, since their power consumptions are also very similar. CReAMS does not affect the power budget even if an error of 19% in our scaling process from CMOS 0.35 um to 90 nm occurs.

For these experiments, we chosen a subset of the applications presented in the previous section. This subset is composed of applications that cover the whole spectrum of opportunities to exploit TLP and ILP. As can be seen, Table 3 shows that *blackscholes*, *swaptions* and *jacobi* have perfect load balancing, so suitable for TLP exploitation. On the other hand, they have tiny room for ILP exploitation, since their mean basic block sizes are small. *susan corners* shows quite the opposite, since it has a significant load unbalancing and provides higher mean basic block size. *lu* was selected to represent applications where neither room for ILP nor TLP is available to be explored.

Table 26. Execution time of 4-issue OOO MPSparcV8 and CReAMS

	4-issue OOO MPSparcV8		***Homogeneous CReAMS***		***Heterogeneous CReAMS***	
	4SparcV8	***8SparcV8***	***32DAPs***	***64DAPs***	***32DAPs***	***64DAPs***
Execution Time (ms)						
susan_c	16.448	11.221	12.838	10.344	13.783	10.916
swaptions	7.094	3.551	1.572	0.792	2.212	1.107
blackscholes	3.408	1.710	1.048	0.535	1.578	0.812
jacobi	123.841	62.245	33.665	19.233	41.864	23.299
lu	0.352	0.265	0.279	0.310	0.315	0.367
Total Ex. Time	***151.142***	***78.992***	***49.402***	***31.214***	***59.750***	***36.500***

Table 26 shows the execution time of both CReAMS and 4-issue OOO MPSparcV8 processor by applying the Power Budget #1 and #2 schemes. As can be seen, CReAMS outperforms the 4-issue OOO MPSparcV8 in all TLP-oriented applications when both Power Budget #1 and #2 are considered. 32-DAP HomoLarge CReAMS is 4.51, 3.26 and 3.68 times faster than 4-core 4-issue OOO MPSparcV8 on running *swaptions*, *blackscholes* and *jacobi*, respectively. When the Power Budget grows to 53 Watts, the gains of 64-DAPs HomoLarge CReAMS over 8-Core 4-issue OOO MPSparcV8 remain almost in the same factors. *lu* runs 26% faster in CReAMS than a 4-issue OOO SparcV8 considering a power budget of 27 Watts. However, when the power budget grows to 53 Watts, the multiprocessing system composed of 4-issue OOO processors outperforms CReAMS by 24%. It is due to the huge load unbalancing that occurs by increasing the number of DAPs from 32 to 64 on that application, which affects the performance of CReAMS. However, it does not occur when the number of processors increases from 4 to 8, thus gains of 4-issue OOO MPSparV8 come from TLP and ILP exploitation. Even when a wide room for ILP exploration is available, CReAMS outperforms the 4-issue OOO MPSparcV8, this is the case of *susan corners* where their basic block have 17 instructions, on average. 32-DAP CReAMS outperforms the 4-Core 4-issue OOO MPSparcV8 by 28% when a power budget of 27 Watts is considered. When the power budget increases to 53 Watts, CReAMS still performs better than OOO MPSparcV8, showing an execution time 8.5% smaller.

Table 26 presents the performance of the 32-DAP and 64-DAP HeteroLarge CReAMS when static scheduling is considered. We apply the same methodology used above to compare heterogeneous CReAMS and 4-issue OOO MPSparcV8. As can be seen, in comparison with the homogeneous setups, heterogeneous CReAMS presents smaller chip area (refer to Table 23(b)) and less power consumption (refer to Table 24). However, the heterogeneous CReAMS remain almost the same gains over the 4-issue OOO MPSparcV8 provided by the homogeneous setups, and still reduce the power cap from 27 Watts and 53 Watts to 20 Watts and 40 Watts for Power Budget #1 and #2 schemes, respectively.

Summarizing, CReAMS shows performance improvement of 28% and 8.5% when a power budget #1 and #2 is applied, which shows that homogeneous CReAMS delivers higher performance per watt than a multiprocessing system based on 4-issue OOO superscalar processors. Moreover, the heterogeneous CReAMS outperforms the multiprocessing system based on OOO superscalar by a factor of 2.34, on average, and provides a power cap 33% lower.

6 CONCLUSIONS AND FUTURE WORKS

In this work, the design space exploration around thread and instruction level parallelism are investigated. In Chapter 2 we presented the related works, which elucidate and motivate the main goals of this proposal. In the Chapter 3, we discussed, using an analytical model, the limits of the TLP and ILP exploitation showing the necessity for mixed parallelism exploitation. In addition, the inter-thread communication cost is analytically studied. In order to cope with that, we have extended to a multiprocessing environment an already proposed reconfigurable architecture that transparently explores the instruction level parallelism of single-threaded applications.

The employment of the reconfigurable architecture on a homogeneous organization multiprocessor system shows that an adaptable ILP exploitation is one requisite to achieve performance improvements with energy savings. However, the strategy of replicating the same processors produces a disadvantageous tradeoff between energy, area and performance. Since most of the processors become idle in great periods of the application execution by waiting for the thread that has greater period of sequential execution. Hence, we show performance improvements and energy savings by joining in the same chip DAPs with different capabilities on exploring instruction level parallelism. Considering that hypothesis, we demonstrate that the heterogeneous organization strategy can reduce substantially the power consumption of the system, while maintaining performance of the homogeneous approach with a mandatory overhead of a dynamic thread scheduling to match the thread requirements with the DAPs capabilities. Finally, we present performance improvements of CReAMS over a multiprocessing system composed of 4-issue Out-Of-Order processors when a power budget is considered.

6.1 Future Works

6.1.1 Scheduling Algorithm

Considering the scheduling algorithm used to assign threads in a heterogeneous CReAMS organization. In this work, the scheduling process is implemented in software. Actually, there are several advantages for hardware implementation that is not only motivated by obtaining better performance than software, but aims to offer transparency to the operating system. The whole processor scheduling process is supported by the DAP's hardware, that is responsible for the fine grain parallelism detection. Once the thread is executed, the DDH stores, as intrinsic information, the level of instruction level parallelism and the number of instructions executed by the thread. This information is useful for the next thread assignments done by the processor

appointer hardware. The identification of the threads in hardware is done through a special register within each DAP that holds the thread ID generated by the operating system. A special table, indexed by the thread ID, stores the appointment metrics that are gathered in the first execution of the threads. For the next executions, the core allocator hardware fetches these metrics from the special cache and allocates a DAP to execute it, considering the best matching between the processing capability and the required metrics. The main novelty of such an approach is the complete transparency offered to the operating system. The implementation of the thread allocation is supported by the DAP's hardware, named Dynamic Detection Hardware (DDH), that is responsible for the fine-grain parallelism detection.

6.1.2 Studies over TLP and ILP considering the Operating System

Operating System is already present in the embedded systems, such as Android and iOS. The software layer added over the hardware cannot be overlooked since it brings a significant overhead to the system performance. However, as the code of the operating systems is split in threads, one can explore CReAMS to achieve performance improvements. In this work, only applications were explored, but one can investigate the efficiency of CReAMS to accelerate both application and operating system code. This measurement has a significant importance to verify how many DAPs are necessary to provide, in a concurrent way, threads of the operating system and the application. Moreover, one important question to be answered is: Is the heterogeneity also efficient in an environment where the operating system is present?

6.1.3 Behavioral of CReAMS on a Multitask Environment

In the multitask environment, there are many threads of different applications running at the same time in the chip multiprocessor. As shown in this work, applications behave in their own way and require different amount of resources to achieve efficient execution. This work can be extended by considering multitask environment, where many applications are running over a homogeneous/heterogeneous CReAMS. Such investigation approaches to the real scenario of current embedded systems, where operating systems provide multitask support.

6.1.4 Automatic CReAMS generation

In (RUTZIG, 2009), the authors propose an automatic tool, named ARISE, for generating an optimized reconfigurable data path considering the application execution. When many processors are encapsulated in a single die, an investigation of the amount of reconfigurable resources needed to produce the best performance on exploiting TLP and ILP is mandatory to reduce area occupied. Besides, this work shows that heterogeneous organization is efficient when area, power and performance are taken into account. However, there is no investigation of the best heterogeneity for a multiprocessor environment, which could be achieved by extending ARISE for CReAMS.

6.1.5 Area reductions by applying the Data Path Virtualization Strategy

Reductions on chip area will be also explored using a data path virtualization technique proposed in (BERTICELLI LO, BECK, *et al.*, 2010). The authors propose a virtual execution using the DIM technique. Currently, the replication on the number of levels dictates the limits of the sequential execution in the reconfigurable data path. The results shown in Section 5 employ a reconfigurable data path composed of eight levels. However, to achieve the same performance results shown in this section, there is need

for only three data path levels by using the virtualization technique. In this technique, on each execution cycle a certain reconfigurable data path works in a specific mode that can be: reconfiguring, running or propagating results. To support the virtualization mode a finite state machine is necessary to control the switching of modes between the three levels. However, the chip area is extremely reduced, since there is no need for huge functional units and interconnection replication to execute configurations longer than 3 cycles. The employment of the data path virtualization does not affect the original performance, since the number of execution cycles is dictated by the width of a reconfiguration memory slot. Despite the huge area saving by avoiding replication of functional units, the virtualization does not produce any savings on the area and of the reconfiguration memory. Currently, this component is responsible for 15% of the total area spent in the DAP implementation, as the area of the data path is drastically reduced with the employment of the virtualization process this number becomes more significant, reaching almost 32% of the total chip area, which becomes a important point to investigate optimizations when heterogeneous CReAMS is considered.

6.1.6 Boosting TLP performance with Heterogeneous Multithread CReAMS

Simultaneous Multithreaded (SMT) processors already have shown an efficient strategy to obtain an advantageous tradeoff between performance improvements and area overhead. However, there are few works using the simultaneous multithread strategy in reconfigurable computing. These works use such an approach in a homogenous fashion, meaning that all processors have the same degree on exploiting ILP and TLP. Heterogeneous multithread DAPs in the same CReAMS platform is a point to be investigated. In a heterogeneous CReAMS platform, some DAPs have more functional units than others, so higher degree of SMT could be applied in those DAPs, achieving performance improvement with the low area overhead. Such overhead comes from supporting the concurrent execution of multithreads in the DAP' data path.

7 PUBLICATIONS

7.1 Book Chapters

1. Wong, Stephan, Carro, Luigi, RUTZIG, Mateus Beck, MATOS, D. M., GIORGI, R., Puzovic, N, Kaxiras, S., DESOLI, G., GAI, P., CINTRA, M., MCKEE, S. A., ZAKS, A.

ERA – Embedded Reconfigurable Architectures In: Reconfigurable Computing: From FPGAs to Hardware/Software Codesign.1 ed.New York : Springer, 2011, v.1, p. 239-260.

2. CARRO, L., RUTZIG, Mateus Beck

Multi-Core System on Chip In: Handbook of Signal Processing Systems.1 ed.New York : Springer, 2010, v.1, p. 485-514.

7.2 Journals

1. Rutzig, Mateus B., Beck, Antonio C. S., Madruga, Felipe, Alves, Marco A., Freitas, Henrique C., Maillard, Nicolas, Navaux, Philippe O. A., Carro, Luigi, RUTZIG, Mateus Beck

Boosting Parallel Applications Performance on Applying DIM Technique in a Multiprocessing Environment. International Journal of Reconfigurable Computing (Print). , v.2011, p.1 - 13, 2011.

7.3 Conferences

1. RUTZIG, Mateus Beck, BECK FILHO, Antonio Carlos Schneider, CARRO, L.

CReAMS: An Embedded Multiprocessor Platform In: International Symposium on Applied Reconfigurable Computing, 2011, Belfast.

2. FAJARDO JUNIOR, J., RUTZIG, Mateus Beck, BECK FILHO, Antonio Carlos Schneider, CARRO, L.

A Dynamically Reconfigurable Architecture with a Two-Level Binary Translation Mechanism In: 5th Workshop on Reconfigurable Computing, 2011, Heraklion.

3. JUNQUEIRA, A., RUTZIG, Mateus Beck, ITTURRIET, F. P., CARRO, L.

A Reconfigurable Fabric Supporting Full C/C++ Input In: International Workshop on Reconfigurable Communication-centric Systems-on-Chip, 2011, Montpellier.

4. FAJARDO JUNIOR, J., RUTZIG, Mateus Beck, BECK FILHO, Antonio Carlos Schneider, CARRO, L.
A Transparent and Adaptable Multiple-ISA Embedded System In: Engineering of Reconfigurable Systems and Algorithms, 2011, Las Vegas.

5. FAJARDO JUNIOR, J., RUTZIG, Mateus Beck, BECK FILHO, Antonio Carlos Schneider, CARRO, L.
Towards an Adaptable Multiple-ISA Reconfigurable Processor In: International Symposium on Applied Reconfigurable Computing, 2011, Belfast.

6. LO, T.B., BECK FILHO, A.C.S., RUTZIG, Mateus Beck, CARRO, L.
A Low-Energy Approach for Context Memory in Reconfigurable Systems In: IEEE International Parallel And Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW), 2010, 2010, Atlanta.

7. LO, T.B., BECK FILHO, Antonio Carlos Schneider, RUTZIG, Mateus Beck
Decreasing the Impact of the Context Memory on Reconfigurable Architectures In: HiPEAC Workshop on Reconfigurable Computing, 2010, Pisa.

8. SILVA, M.G., HECKTHEUER, B., MATTOS, J. C. B., BECK FILHO, A.C.S., RUTZIG, Mateus Beck, CARRO, L.
Floating point unit implementation for a reconfigurable architecture In: South Symposium on Microelectronics, 2010, Porto Alegre.

9. SILVA, M.G., HECKTHEUER, B., MATTOS, J. C. B., BECK FILHO, A.C.S., RUTZIG, Mateus Beck, CARRO, L.
Implementação de uma Unidade de Ponto Flutuante para uma Arquitetura Reconfigurável. In: XVI IBERCHIP Workshop, 2010, Foz do Iguagu.

10. RUTZIG, Mateus Beck, MADRUGA, F.L., COSTA, H., BECK FILHO, A.C.S., MAILLARD, N, B., NAVAUX, P.O.A.
TLP and ILP exploitation through Reconfigurable Multiprocessing System In: IEEE International Parallel And Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW), 2010, Atlanta.

11. FERREIRA, R.S., LAURE, M., BECK FILHO, A.C.S., LO, T.B., RUTZIG, Mateus Beck, CARRO, L.
A Low Cost and Adaptable Routing Network for Reconfigurable Systems In: IEEE International Parallel And Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW), 2009,, 2009, Roma.

12. RUTZIG, Mateus Beck, BECK FILHO, A.C.S., CARRO, L.
Dynamically Adapted Low Power ASIPs In: International Workshop on Reconfigurable Computing, 2009, 2009, Karlsruhe.

REFERENCES

ALBONESI, D.; WATKINS, M. A. ReMAP A Reconfigurable Heterogeneous Multicore Architecture. PROCEEDINGS OF 43RD ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE. Washington: ACM. 2010. p. 497-508.

ANANTARAMAN, A. et al. Virtual simple architecture (VISA) exceeding the complexity limit in safe real-time systems. PROCEEDINGS OF 30TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE. New York: ACM. 2003. p. 350-361.

ANDRE, L. et al. Piranha a scalable architecture based on single-chip multiprocessing. SIGARCH Comput. Archit. News 28. New York: ACM. 2000. p. 282-293.

BECK, A. C. S. et al. Transparent reconfigurable acceleration for heterogeneous embedded applications. PROCEEDINGS OF DESIGN, AUTOMATION AND TEST IN EUROPE. New York: ACM. 2008. p. 1208-1213.

BERTICELLI LO, T. et al. A low-energy approach for context memory in reconfigurable systems. PROCEEDINGS OF 17TH RECONFIGURABLE ARCHITECTURES WORKSHOP. [S.l.]: [s.n.]. 2010. p. 19-23.

BIENIA, C. et al. The PARSEC benchmark suite: characterization and architectural implications. PROCEEDINGS OF 17TH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES. New York: ACM. 2008. p. 72-81.

BLAKE, G. et al. Evolution of thread-level parallelism in desktop applications. SIGARCH Comput. Archit. News 38. New York: ACM. 2010. p. 302-313.

BRANDAO, R.; WYNN, M. Product Lifecycle Management Systems and Business Process Improvement – A Report on Case Study Research. [S.l.]: [s.n.]. 2008. p. 113--118.

CHEN, T. et al. Cell broadband engine architecture and its first implementation: a performance view. IBM J. Res. Dev. 51. [S.l.]: [s.n.]. 2007. p. 559-572.

CHEN, X. et al. Speedup analysis of data-parallel applications on Multi-core NoCs. PROCEEDINGS OF INTERNATIONAL CONFERENCE ON ASIC. [S.l.]: [s.n.]. 2009. p. 105-108.

CLARK, N. et al. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. PROCEEDINGS OF 37TH ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE. Washington: ACM. 2004. p. 30-40.

DIEFENDORFF, K. Hal Makes Sparcs Fly. [S.l.]. 1999.

DIXIT, K. M. The SPEC benchmarks. In Computer benchmarks. [S.l.]: Elsevier. 1993. p. 149-163.

DORTA, A. J. et al. The OpenMP Source Code Repository. PROCEEDINGS OF 13TH EUROMICROCONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING. Washington: IEEE Computer Society. 2005. p. 244-250.

FUJITSU MICROELECTRONICS, I. New TurboSPARC Processor Sets New Performance Level For Low-End, Mid-Range Workstations. [S.l.].

GAISLER. Gaisler, 2006. Disponivel em: <http://www.gaisler.com>. Acesso em: 12 December 2010.

GARCIA, P.; COMPTON, K. Kernel sharing on reconfigurable multiprocessor systems. PROCEEDINGS OF INTERNATIONAL CONFERENCE ON ICECE TECHNOLOGY. [S.l.]: IEEE. 2008. p. 225.

GOLDSTEIN, S. C. et al. PipeRench: A Reconfigurable Architecture and Compiler. **Computer 33**, Los Alamitos, April 2000. 70-77.

GONZALEZ, R. E. Xtensa: a configurable and extensible processor. **IEEE Micro**, v. 20, n. 2, p. 60-70, March 2000.

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. PROCEEDINGS OF IEEE INTERNATIONAL WORKSHOP ON WORKLOAD CHARACTERIZATION. [S.l.]: [s.n.]. 2002.

HAMMOND, L. et al. The Stanford Hydra CMP. **IEEE Micro 20**, Los Alamitos, March 2000. 71-84.

HAUCK, K.; COMPTON, K. Reconfigurable computing: a survey of systems and software. ACM Comput. Surv. 34. New York: ACM. 2002. p. 171-210.

HAUCK, S. et al. The chimaera reconfigurable functional unit. IEEE Trans. Very Large Scale Integr. Syst. Piscataway: ACM. 2004. p. 206-217.

HEINRICH, J. MIPS R1000 User Manual. **MIPS R1000 User Manual**, 1997. Disponivel em: <http://techpubs.sgi.com/library/manuals/2000/007-2490-001/pdf/007-2490-001.pdf>.

HENKEL, J. Closing the SoC design gap. **Computer**, v. 36, p. 119--121, 2003.

INTEL. Inside an 80-core chip: the on-chip communication and memory bandwidth solution, 2007. Disponivel em: <http://blogs.intel.com/research/2007/07/inside_the_terascale_many_core.php>. Acesso em: 15 March 2011.

JOHNSON, T.; NAWATHE, U. An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2). PROCEEDINGS OF 2007 INTERNATIONAL SYMPOSIUM ON PHYSICAL DESIGN. New York: ACM. 2007. p. 2-7.

KAVADIAS, S. Limited Priority-Thread Based Sharing of Simultaneous MultiThreaded Processor Resources. Master dissertation of. [S.l.]: [s.n.].

KAVVADIAS, S. A Trace Driven Configurable SparcV8 ISA Simulator, 2001. Disponivel em:

<http://www.ics.forth.gr/~kavadias/SMT_Page/trace_driven_configurable_simulator.htm>. Acesso em: 6 May 2010.

KOENIG, R. et al. KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-SetMulti-grained-Array Architecture. PROCEEDINGS OF DESIGN, AUTOMATION & TEST IN EUROPE CONFERENCE. [S.l.]: [s.n.]. 2010. p. 819-824.

KUMAR, R. et al. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. PROCEEDINGS OF 36TH ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE. [S.l.]: [s.n.]. 2003. p. 81 - 92.

KUMAR, R. et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. SIGARCH Comput. Archit. News 32. New York: ACM. 2004. p. 64-80.

KUMAR, R.; JOUPPI, N. P.; TULLSEN, D. M. Core architecture optimization for heterogeneous chip multiprocessors. PROCEEDINGS OF THE 15TH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES. [S.l.]: [s.n.]. 2006.

LINDHOLM, E. et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro 28. [S.l.]: IEEE. 2008. p. 39-55.

LYSECKY, R.; STITT, G.; VAHID, F. Warp Processors. PROCEEDINGS OF 41ST ANNUAL DESIGN AUTOMATION CONFERENCE. New York: ACM. 2004. p. 659-681.

MAGNUSSON, P. S. et al. Simics: A Full System Simulation Platform. Computer 35. Los Alamitos: ACM. 2002. p. 50-58.

MAK, J. . Limits of instruction-level parallelism. PROCEEDINGS OF INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS. New York: ACM. 1991.

MATOS, D. et al. A NOC closed-loop performance monitor and adapter. **Microprocessors and Microsystems**, 2011. 1-10.

MENON, L. OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Comput. Sci. Eng. 5. Los Alamitos: IEEE. 1998. p. 46-55.

NVIDIA. **Whitepaper of Variable SMP**, 2011. Disponivel em: <http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf>. Acesso em: 15 December 2011.

PATTERSON, D.; HENNESSY, J. **Computer Organization and Design**. [S.l.]: Elsevier, 2010.

RUTZIG, M. B. B. F. A. C. S. . C. L. Dynamically Adapted Low Power ASIPs. PROCEEDINGS OF INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMPUTING. [S.l.]: [s.n.]. 2009. p. 110-122.

RUTZIG, M. B.; BECK, A. C.; CARRO, L. Dynamically Adapted Low Power ASIPs. PROCEEDINGS OF 5TH INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMPUTING: ARCHITECTURES, TOOLS AND APPLICATIONS. Berlin: Springer-Verlag. 2009. p. 110-122.

SANKARALINGAM, K. et al. TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. **ACM Trans. Archit. Code Optim.**, New York, March 2004. 62-93.

SEILER, L. et al. Larrabee: a many-core x86 architecture for visual computing. ACM SIGGRAPH 2008 Papers. Los Angeles: ACM. 2008. p. 1-15.

SEMICONDUCTORS, I. T. R. F. ITRS. **ITRS**, 2009. Disponivel em: <http://www.itrs.net/>. Acesso em: 12 Maio 2010.

SHI, K.; HOWARD, D. Challenges in Sleep Transistor Design and Implementation in Low-Power Designs. [S.l.]: [s.n.]. 2006. p. 113 – 116.

SMIT, G. J. M. Multi-core Architectures and Streaming Applications. PROCEEDINGS OF INTERNATIONAL WORKSHOP ON SYSTEM LEVEL INTERCONNECT PREDICTION. New York: ACM. 2008. p. 35-42.

STITT, G.; VAHID, F. Thread warping: a framework for dynamic synthesis of thread accelerators. PROCEEDINGS OF ACM INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS. New York: ACM. 2007. p. 93-98.

SWANSON, S. The WaveScalar architecture. **ACM Trans. Comput. Syst.**, New York, May 2007. 54.

VANGAL, S. et al. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. PROCEEDINGS OF SOLID-STATE CIRCUITS CONFERENCE. [S.l.]: IEEE explorer. 2007. p. 98-589.

VASSILIADIS, S. et al. The MOLEN Polymorphic Processor. **IEEE Trans. Comput.**, Washington, November 2004. 1363-1375.

WATKINS, M. A.; CIANCHETTI, M. J.; ALBONESI, D. H. Shared reconfigurable architectures for CMPS. PROCEEDINGS OF INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATION. [S.l.]: IEEE. 2008. p. 299 - 304.

WAWRZYNEK, J. R. H. Garp: a MIPS processor with a reconfigurable coprocessor. PROCEEDINGS OF IEEE SYMPOSIUM ON FPGA-BASED CUSTOM COMPUTING MACHINES. Washington: ACM. 1997.

WILTON, S. J. E.; JOUPPI, N. P. CACTI: an enhanced cache access and cycle time model. **IEEE Journal of Solid-State Circuits**, May 1996. 677-688.

WOO, D. H.; LEE, H. Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era. **Computer**, December 2008. 24-31.

WOO, S. C. et al. The SPLASH-2 programs: characterization and methodological considerations. PROCEEDINGS OF ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE. New York: ACM. 1995. p. 24-36.

YAN, L. et al. A Reconfigurable Processor Architecture Combining Multi-core and Reconfigurable Processing Unit. PROCEEDINGS OF IEEE INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION TECHNOLOGY. West Yorkshire: [s.n.]. 2010. p. 2897-2903.

APPENDIX A

Introdução

Atualmente, a grande demanda por sistemas embarcados pelo mercado faz com que o projeto destes dispositivos torne-se cada vez mais complexo.Telefones celulares são capazes de embarcar aplicações durante seu tempo de vida, fato que torna o projeto de um hardware para um dispositivo atual um grande desafio visto que o mesmo deve ser capaz de executar eficientemente todos os comportamentos de software com baixo consumo de energia.

No cenário embarcado atual, dois tipos de abordagens são utilizados para aumentar o desempenho de processadores embarcados: extensões no conjunto de instruções e a implantação de ASICs. A primeira abordagem tem como objetivo resolver alguns gargalos criados pela execução massiva de certas aplicações. Assim, quando é observado um montante de aplicações embarcadas que compartilham o mesmo comportamento, extensões no conjunto de instruções são realizadas para que este comportamento possa ser executado de forma eficiente em termos de desempenho e consumo de energia. ASICs são frequentemente utilizados em plataformas embarcadas para solucionar questões de desempenho de aplicações que já estão fortemente embarcadas e consomem um grande tempo de execução em software. Entretanto, tanto extensões no conjunto de instruções quanto ASICs afetam a produtividade de software, visto que para cada nova plataforma é necessário mudanças na ferramenta de geração de código e, conseqüentemente, a recompilação de todas as aplicações, possibilitando que estas aplicações explorem os novos recursos da plataforma. Este fato afeta o time-to-market e aumenta o tempo de projeto do dispositivo embarcado, requisitos muito exigidos no neste domínio.

Devido às razões explicitadas acima, é necessária uma mudança no paradigma de concepção de hardware para o mercado embarcado. A utilização de sistemas multiprocessados provêem diversas vantagens, o tempo de execução pode ser claramente beneficiado visto que diversas partes do programa podem ser executadas concorrentemente nos diversos processadores da plataforma. O tempo de validação destas plataformas é extremamente beneficiado visto que um único processador deve ser validado para posterior replicação. Em um sistema multiprocessado o projetista de hardware é responsável por encapsular o número máximo de elementos de processamento em um único chip. Por outro lado, o projetista de software é responsável pela árdua tarefa de distribuir o software entre os diversos elementos de processamento. Assim, produtividade de software surge como o principal desafio quando ambientes multiprocessados são considerados, visto que as aplicações devem ser lançadas no

mercado rapidamente e o código binário das mesmas deve ser genérico suficiente para ser executado em qualquer plataforma que seja lançada no futuro. Adicionalmente, o infraestrutura de interconexão entre os processadores deve ser suficientemente eficiente para que os ganhos em desempenho da execução paralela não seja afetado pelo processo de intercomunicação das threads.

Assim, um sistema multiprocessado ideal para sistemas embarcados deve ser composto pela replicação de elementos de processamento, sendo que cada um destes elementos possam se adaptar as particularidades das aplicações, esta adaptação tem de ocorrer mesmo após a fabricação. A plataforma deve emular o comportamento, em termos de desempenho e energia, dos ASICs que estão sendo implementados com sucessos nas plataformas embarcadas atuais. No mesmo momento, o uso do mesmo conjunto de instruções para todos os elementos de processamento é mandatório para manter a produtividade de software e evitar a modificação das ferramentas que geram código e, conseqüentemente, a recompilação do código fonte para cada nova versão de plataforma. Esta plataforma deve atacar de forma eficiente todo o espectro de comportamento de aplicações: aquelas que contêm paralelismo em nível de threads massivo e as aplicações onde este paralelismo é inexistente. Ainda, a plataforma deve ser concebida com uma organização heterogênea para fornecer o melhor compromisso entre a heterogeneidade das aplicações e a área ocupada em chip pelo sistema multiprocessado. Por fim, o exato número de processadores deve ser investigado, visto que o desempenho do sistema concebido com muitos processadores pode ser afetado pelos custos providos pela comunicação entre threads.

Objetivos

Considerando todas as motivações demonstradas anteriormente, o primeiro objetivo deste trabalho é focado em reforçar, pelo uso de um modelo analítico, que a aplicação da exploração de paralelismo em um único nível não prove um compromisso vantajoso em relação ao desempenho obtido e a energia consumida pelo sistema. Ainda, este estudo fornece alguns rumos sobre a fatia de hardware que deve ser empregada para explorar o paralelismo em nível de instrução e thread. Um modelo de comunicação de uma rede em chip é criado para investigar o impacto da comunicação entre as threads.

Neste cenário, é proposto uma plataforma baseada em Custom Reconfigurable Arrays for Multiprocessor System (CReAMS), pelo acoplamento de dois diferentes conceitos: arquiteturas reconfiguráveis e sistemas multiprocessados. Em um primeiro passo, CReAMS é concebido em uma organização homogênea. Entretanto, esta plataforma virtualmente se comporta como uma organização heterogênea devido a sua capacidade de se adaptar em tempo de execução.

O sistema é capaz de explorar de forma transparente, ou seja, sem modificações no código binário original, o paralelismo em nível de instrução das threads em execução, oferecendo uma alta habilidade em se adaptar as demandas de paralelismo das diferentes aplicações. O paralelismo em nível de threads não depende de nenhuma ferramenta que faça investigação do código em tempo de projeto, visto que o paralelismo em nível de thread é explorado pelas interfaces de programação (OpenMP e Pthreads) conhecidas e suportadas pelos compiladores do mercado, tornando a execução de CReAMS independente de qualquer processo proprietário. Dinamicamente é possível balancear a melhor exploração do paralelismo em nível de instrução e thread. Assim, qualquer tipo de código, tanto aquele que apresenta alto TLP e baixo TLP quanto o código que tem características inversas é acelerado. CReAMS prove menor consumo de

energia e mantém a produtividade de software das organizações homogêneas. Um único conjunto de ferramenta é necessário para toda a plataforma e qualquer modificação no hardware não requer nenhuma modificação no código binário.

CReAMS

Um visão geral de CReAMS é apresentada na Figura 1. O paralelismo em nível de thread é explorado pela replicação de DAPs (Dynamic Adaptive Processors). A comunicação entre os DAPs é feita através de uma NoC com topologia mesh.

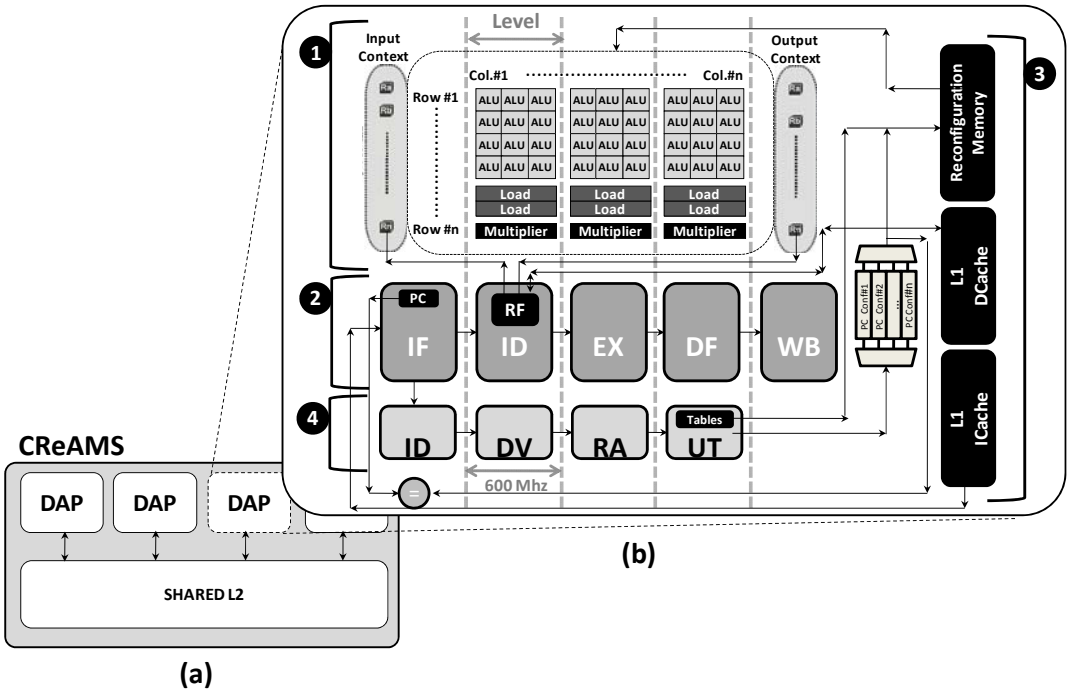


Figura 1. (a) Organização de CReAMS (b) DAP

DAP

Um Dynamic Adaptive Processor é divido em quatro blocos ilustrados na Figura 1. Estes blocos são discutidos nas seções a seguir:

Processor Pipeline

Um processador baseado na arquitetura SparcV8 é utilizado como o processador base da plataforma. Este processador contém cinco estágios de *pipeline*, refletindo um processador RISC tradicional.

Detector dinâmico em Hardware

Uma das restrições impostas pelo projeto de um dispositivo embarcado é o tempo de projeto. A concorrência das empresas para disponibilizar no mercado o primeiro dispositivo com novas funcionalidades força o projeto embarcado ser cada vez mais curto. Assim, os projetistas devem utilizar técnicas que ajudem a modelagem e reaproveitamento do software já escrito para o dispositivo anterior.

A tradução binária é largamente utilizada em processadores de propósito geral para prover compatibilidade de software. Em (BECK, RUTZIG, *et al.*, 2008), foi

demonstrado que a utilização da técnica de tradução binária também alcança ótimos resultados de desempenho e energia em um processador que executa o conjunto de instruções PISA (BURGER, 1997). Assim, para este estudo foi herdada a técnica aplicada neste último trabalho.

A idéia básica do mecanismo é prover compatibilidade de software a partir da tradução binária de seqüências de instruções, em tempo de execução, para que as mesmas possam ser futuramente executadas num mecanismo mais eficiente, neste caso, em uma unidade funcional reconfigurável.

Basicamente, o mecanismo avalia cada instrução executada pelo processador, agrupando-as em blocos chamados de configuração da unidade funcional reconfigurável (UFR). A cada instrução executada pelo processador é verificada a possibilidade da execução desta na unidade reconfigurável. Caso positivo, a mesma é alocada na configuração corrente da UFR. Ao final da construção de uma configuração da UFR, esta é armazenada em uma cache de reconfigurações, indexada pelo valor do contador de programa da primeira instrução. Quando este valor novamente for alcançado pelo contador de programa, o mecanismo reconfigurável é ativado seguindo os seguintes passos:

- Efetua-se a busca da configuração, da seqüência de instruções em questão, na cache de reconfigurações;
- Configura-se a UFR com os bits fornecidos pela cache;
- Os valores dos registradores necessários para executar a configuração são carregados no contexto de entrada da UFR;
- A execução é realizada na UFR;

Após o término da execução na UFR o valor do contador de programa é atualizado, assim como os valores dos registradores modificados no banco de registradores. Do mesmo modo as escritas na memória de dados são realizadas. É importante destacar que enquanto o mecanismo reconfigurável está ativo o processador não realiza nenhuma operação.

Diferentemente dos processadores superescalares, esta abordagem não realiza repetidas vezes, para a mesma seqüência de instruções, a verificação das dependências entre as instruções. A utilização da técnica de reuso de rastros (do inglês *trace reuse*) evita a repetição desta tarefa. Portanto, se não houver falha na cache de reconfiguração, o mecanismo de TB somente será aplicado uma única vez em cada seqüência de instruções.

Em relação a mecanismos utilizados em outros sistemas reconfiguráveis, que somente aplicam as técnicas de reconfiguração em partes mais executadas da aplicação, a abordagem de TB descrita neste trabalho fornece uma maior flexibilidade. A aplicação inteira sofre análise do TB, não se limitando a apenas partes específicas da aplicação.

O hardware de TB é composto basicamente por tabelas e mapas de bits que armazenam temporariamente dados da configuração corrente. O algoritmo é composto por seis tabelas e dois mapas de bits que serão especificadas a seguir:

- Mapas de Escrita – a função deste mapa é armazenar o número do registrador de escrita de cada instrução alocada na UFR. Este mapa é utilizado na verificação das dependências entre as instruções no momento de alocação

das mesmas na arquitetura. Em cada linha da UFR existe um mapa de escrita, sendo o número de bits igual ao número de registradores existentes no contexto de entrada. Este mapa de bits não será armazenado na configuração final, pois não tem utilidade no momento da execução.

- Mapas de Recurso – este mapa foi inserido no mecanismo para gerenciar a alocação de recursos na UFR. Assim, no momento da inserção de uma nova instrução em uma configuração, é realizada a busca por uma unidade funcional ociosa neste mapa. Analogamente ao mapa de escrita, os dados do mapa de recurso não serão inseridos na configuração.
- Tabela de Contexto Atual – armazena uma referência a todos os registradores que serão utilizados pelas instruções executas por uma configuração.
- Tabela de marcadores de contexto atual – faz a caracterização dos registradores da tabela de contexto atual, distinguindo entre registradores de leitura e de escrita.
- Tabela de Contexto Inicial de Leitura – nesta tabela são inseridas as referências a todos os operandos de leitura armazenados na tabela de marcadores de contexto atual. A inserção é feita na posição correspondente à posição do operando na tabela de contexto atual. A função desta tabela é indicar quais os registradores devem ser carregados no contexto de entrada no inicio da execução.
- Tabela de Imediatos – as instruções do tipo-I e do tipo-J trazem em seu corpo operandos imediatos. Para que estas possam ser executadas na UFR é necessário armazenar o valor destes operandos. Assim, a função desta tabela é armazenar os valores imediatos, para que no momento da execução, sejam carregados para o contexto de entrada.
- Tabela de Leituras – a função desta tabela é armazenar quais registradores serão entradas de cada unidade funcional. Especificamente, os dados desta tabela serão inseridos, na hora da execução, nos bits de controle dos multiplexadores de entrada de cada unidade funcional. O modo que esta tabela indica os registradores é referenciando a posição dos mesmos na tabela de contexto inicial.
- Tabela de Escrita – analogamente a tabela de leituras, a tabela de escrita armazena a referência para a coluna em que o recurso foi alocado na UFR. A definição da posição de escrita nesta tabela corresponde à mesma posição em que este operando está na tabela de contexto. Estes valores servirão como controle dos demultiplexadores alocados após as unidades funcionais, com o objetivo de realizar a escrita nos registradores do contexto de saída.

A Figura 1 demonstra o hardware de tradução binária acoplado ao processador SparcV8. O hardware foi dividido em quatro estágios para não infringir o caminho crítico do processador. Os estágios que estão em cinza escuro são estágios do processador e os estágios que estão em cinza claro são estágios do hardware de TB. Cada instrução executada pelo processador é analisada pelo TB que realiza a alocação destas nas tabelas e mapas de bits que compõe o hardware de detecção. Abaixo é demonstrada a análise de execução de cada estágio do algoritmo.

- Decodificação (ID) – neste estágio é realizada a decodificação dos campos da instrução, estas informações servirão para identificar características como:
    - Grupo da Instrução: em qual grupo da unidade reconfigurável a instrução deve ser alocada.
    - Tipo e Função da Instrução: em qual unidade funcional do grupo a instrução deve ser alocada, além de qual função a unidade em questão deve desempenhar para executar a instrução.
    - Registradores de Leitura e Escrita: identificam quais são os registradores de leitura e escrita da instrução.
    - Operandos Imediatos: identifica, se a instrução possuir, operandos imediatos.
- Dependência (DV) – após a instrução estar decodificada e seus campos identificados, o papel deste é verificar as dependências de dados existentes entre as instruções. Este trabalho será útil para o próximo estágio alocar as instruções nas unidades funcionais. Dependências de dados verdadeiras são bastante comuns entre instruções em um fluxo de execução, então se deve tomar o devido cuidado na execução destas garantindo a consistência dos dados.
- Recursos (RA) – no estágio anterior são detectadas as dependências verdadeiras existentes no fluxo de execução entre as instruções. Neste estágio é verificada a disponibilidade de unidades funcionais na UFR e realizada a alocação de uma destas para que a instrução em questão possa ser executada.
- Atualização de Tabelas e Mapas de Bits (UT) – neste estágio todas as tabelas e mapas demonstrados anteriormente são atualizados. Assim, a cada instrução adicionada à UFR, é realizada a atualização nas tabelas com as devidas informações necessárias para que a instrução seja executada. Deste modo, é garantido que, no momento da execução, a instrução seja alocada, executada e o seu resultado seja armazenado no registrador destino. Ao final da construção de uma configuração, todos os dados das tabelas necessárias serão empacotados, e a mesma será armazenada na cache de reconfigurações.

A detecção de um bloco de instruções para montagem de uma configuração ocorre de forma seqüencial, em momento de execução. Entretanto, alguns fatores podem interromper a formação deste bloco e, conseqüentemente, a montagem de uma configuração. O primeiro fator são as instruções que não tem suporte de execução na UFR, exemplos destas são as instruções que realizam operações de divisão. No momento em que este tipo de instrução é encontrado pelo TB, a configuração corrente é concluída, e armazenada na cache de reconfigurações. Posteriormente, quando uma instrução com suporte de execução na UFR é encontrada, uma nova configuração é iniciada.

Outro fator que interrompe a formação de uma configuração são as instruções de salto. Assim, a cada instrução de salto executada pelo processador a configuração corrente é concluída e uma nova configuração é iniciada. Em (BECK, RUTZIG, *et al.*, 2008) foi proposto um mecanismo de especulação de saltos, este cria árvores de execução que são utilizadas na formação das configurações da UFR.

A utilização de especulação possibilita a formação de configurações que ultrapassam a execução de instruções de salto, impactando diretamente no desempenho da execução da aplicação pela inclusão de um número maior de instruções em uma única configuração. O número de saltos executados em uma única configuração da UFR pode ser definido pelo projetista. Entretanto, a penalidade por erro de especulação é proporcional ao crescimento do nível especulado.

Memória de Reconfiguração

O sistema reconfigurável, fundamentalmente, explora três técnicas amplamente difundidas no meio científico: reconfigurabilidade, tradução binária e reuso. Esta última explora a natureza de execução do modelo Von-Neumann, onde o contador de programa é o elemento que dirige o fluxo de execução. Assim, a existência de um laço ou de saltos remete a repetição de certo trecho de código da aplicação.

A abordagem proposta explora esta característica das aplicações, realizando tradução binária destes trechos de códigos e armazenando as configurações realizadas em uma cache, evitando a repetitiva análise de código realizada pelos processadores superescalares.

Cada posição da cache de reconfigurações armazena dados necessários para a execução de uma configuração na UFR. Como já explicitado anteriormente, cada configuração armazena os bits necessários para controlar os multiplexadores e demultiplexadores, bits de controle das funções de cada unidade funcional, além dos dados imediatos contidos nas instruções.

Na extração dos resultados de (BECK, RUTZIG, *et al.*, 2008) o algoritmo de substituição FIFO (do inglês *first in, first out*) foi utilizado. Como contribuição para esta dissertação uma maior exploração desta característica foi realizada. Algoritmos tradicionais de substituição foram implementados; LRU (do inglês *least recently used*); LFU (do inglês *least frequently used*) e o algoritmo randômico. Em alguns casos o algoritmo LRU e FIFO obtiveram o mesmo número de faltas na cache. Entretanto, na maioria das aplicações o algoritmo FIFO demonstrou melhores resultados, ou seja, este algoritmo é capaz de abranger a região exata da execução temporal das configurações.

Metodologia

Para extrair resultados de desempenho, energia e potência do sistema proposto foram selecionadas algumas aplicações que refletem comportamentos distintos em relação ao paralelismo em nível de instrução e thread. Todas as aplicações foram paralelizadas com as interfaces de programação OpenMP e Pthreads.

Um ambiente de simulação foi criado onde é agregado o simulador Simics, scripts realizados para este trabalho e um simulador desenvolvido que emula o comportamento de um DAP. Pontos de sincronização são precisamente calculados pelo ambiente de simulação que trabalha com precisão de ciclo.

Para extração de dados de potência, área e caminho crítico, todos os componentes do DAP foram descritos em VHDL. O consumo de potência e a área dos componentes de memória foram extraídos com a ferramenta CACTI 6.5.

Resultados

Os primeiros resultados demonstram a comparação de CReAMS e um sistema multiprocessado composto de processadores SparcV8, chamado MPSparcV8. Um DAP ocupa a área de quatro SparcV8. Assim, criou-se um cenário de comparação de 4-DAP CReAMS e 16-SparcV8, que refletem plataformas com a mesma área. Os resultados demonstraram que CReAMS produz um tempo de execução menor em 6 aplicações das 10 simuladas. CReAMS prove um maior tempo de execução nas aplicações onde o TLP é massivo, visto que no cenário de mesma área CReAMS tem 4 vezes menos processador do que MPSparV8. Quando um cenário de mesma área é criado, onde o número de processadores é maior (16-DAP CReAMS e 64-SparcV8), CReAMS produz um tempo de execução menor do que MPSparcV8 em 7 aplicações, perdendo nas aplicações onde o TLP é extremamente massivo. O consumo de energia de CReAMS é menor em todas as aplicações visto que a arquitetura possui algumas abordagens que evita este consumo: menos acessos a memória de instrução, modo inteligente de reconfiguração da unidade funcional reconfigurável e a utilização de Sleep Transistors. Em média o tempo de execução é 30% menor em CReAMS consumindo menos 32% de energia do que o MPSparcV8.

Nos resultados explicitados anteriormente o custo de comunicação entre as threads é desconsiderado. Assim, a latência de uma rede em chip com topologia mesh é modelada para verificar qual o impacto da comunicação em CReAMS e em MPSparcV8. Os resultados demonstraram que a interconexão afetou mais o MPSparcV8 do que CReAMS visto que o primeiro, em um cenário de mesma área, tem mais processadores do que CReAMS. Sem comunicação CReAMS produz um tempo 20% menor na execução da aplicação *apsi* do que MPSparcV8, quando a comunicação é considerada os ganhos aumentaram para 37%. O consumo de energia de CReAMS foi beneficiado em relação ao consumo de MPSParcV8, sem comunicação os ganhos do primeiro é de 32% em relação ao segundo, quando comunicação é inserida os ganhos sobem para 49%. Isto se dá pela maior latência de comunicação provida por um sistema multiprocessado que contém mais processadores.

Resultados interessantes foram obtidos concebendo CReAMS com ouma organização homogênea, com o objetivo de reduzir ainda mais o consumo de energia, CReAMS foi concebida com uma organização heterogênea. Desta forma, três diferentes configurações de DAPs foram criadas e acopladas em um mesmo chip. Os resultados obtidos em um cenário com chips de mesma área demonstraram que organizações heterogêneas com menor exploração de TLP e maior exploração de ILP do que as homogêneas provem ganhos nas aplicações onde ILP é predominante e perdas onde o TLP é dominante. Ganhos de energia são obtidos pelas organizações heterogêneas visto que o seu Termal Design Power (TDP) é sempre menor do que o das organizações homogêneas.

Entretanto, em outro cenário de comparação de mesma área, onde as organizações heterogêneas possuem maior nível de explorar TLP e menor nível de explorar ILP do que as homogêneas, a heterogeneidade mostrou-se mais eficiente em aplicações onde o TLP é massivo e também em aplicações onde o ILP é massivo. Conclui-se que os poucos processadores que possuem alta capacidade de explorar ILP já exploram significativamente este tipo de paralelismo. O consumo de energia das organizações heterogêneas é maior do que as homogêneas devido ao seu maior TDP.

A partir dos resultados obtidos na exploração das organizações heterogênea verificou-se a necessidade de inclusão de um escalonador dinâmico de threads no sistema multiprocessado. Algumas threads que necessitavam de grande exploração de ILP estavam alocadas em processadores que proviam fraca exploração deste nível de paralelismo, fazendo com que o tempo de execução de algumas aplicações fosse afetado. Assim, um escalonador que detectasse, em tempo de execução, a falha de alocação das threads e realocasse as mesmas para um processador com alto grau de exploração de ILP poderia diminuir as perdas providas pelas organizações heterogêneas. Os resultados demonstraram que um simples escalonador dinâmico de threads consegue diminuir o tempo de execução em até 15%, dado que prova a necessidade de escalonamento quando este tipo de organização é utilizada.

Por fim, uma comparação de CReAMS com um sistema multiprocessado composto por processadores superescalares provido de execução fora-de-ordem foi realizada. Foram criados dois cenários de comparação, ambos com o mesmo orçamento de potência. No primeiro cenário foi comparado 4-OOO 4-issue MPSparcV8 e 32-DAP CReAMS. CReAMS se mostrou mais eficiente em todas as aplicações, em média o ganho foi de 3 vezes sobre a plataforma composta por quatro processadores superescalares. Em um segundo cenário, foi comparado 8-OOO 4-issue MPSparcV8 e 64-DAP CReAMS. CReAMS também foi mais eficiente em todas as aplicações, e em média alcançou-se um tempo de execução 2.5 vezes menor. Estes dados mostram que CReAMS entrega mais desempenho por watt consumido do que um sistema multiprocessador composto de processadores 4-issue com execução fora de ordem.

Conclusões

Neste trabalho o espaço de projeto sobre o paralelismo em nível de instrução e thread é explorado. O uso de um modelo analítico demonstrou os limites de ambos os níveis de paralelismo e a necessidade de explorar ambos de forma conjunta. O mesmo modelo analítico mostrou o grande impacto que a comunicação entre threads produz em um sistema onde vários processadores são encapsulados em um mesmo chip. Levando as conclusões do modelo analítico em conta, foi proposto um sistema multiprocessado que agrega a exploração transparente de ILP por arquitetura reconfigurável com a produtividade de software provida pelas interfaces de programação OpenMP e Pthreads. Este sistema multiprocessado organizado homogeneamente demonstrou-se ganhos em desempenho e energia em comparação com um sistema multiprocessado tradicional que ocupa a mesma área em chip. Após, a organização heterogênea deste sistema se demonstrou ainda mais eficiente em termos de desempenho e energia, alcançando resultados ainda mais significantes. A adaptabilidade em explorar o paralelismo em nível de instrução do sistema proposto se mostrou mais eficiente em termos de desempenho do que um sistema multiprocessado composto por processadores superescalares com execução fora de ordem levando em conta um mesmo orçamento de potência para ambas as plataformas.