UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

JECKSON DELLAGOSTIN SOUZA

Evaluation of Heterogeneous CReAMS

Paper presented as partial requirement for the degree of Bachelor in Computer Engineering.

Advisor: Prof. Dr. Antonio Carlos Schneider Beck
Co-advisor: Prof. Mateus Beck Rutzig

Porto Alegre
2014

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do Curso de Engenharia de Computação: Prof. Marcelo Götz
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço a minha família, pelo apoio durante toda a minha diplomação. Ao meu pai, Valerio, que sempre incentivou, mesmo que de forma implícita, todas as minhas decisões. À minha mãe, Genelci, por não ter me expulsado de casa mesmo perante a falta de organização explícita de um concluinte de graduação. E ao meu irmão, Jonatas, por estar sempre disponível para ajudar.

Agradeço aos meus amigos – os que conheci antes, durante ou fora da faculdade – que ajudaram a tornar o todo o processo mais simples. Em especial, agradeço a Alessandra Leonhardt, Arthur Rauter, Eduardo Conto, Gennaro Rodrigues, Ingrid Lopes, Lucas Schons, Luís Henrique Reinicke, Luis Henrique Grassi e Marcelo Brandalero que tornaram todo esse final do curso mais especial. Aos meus amigos de Glasgow, que foram como uma família por um ano e com os quais eu dividi alguns dos melhores momentos da minha vida, e, por isso, estarão sempre presentes comigo, mesmo que distantes. Em especial, ao Eduardo Couto, Juliana Versiane, Lucas de Souza e Aline Carvalho por terem me recebido em suas casas e feito viagens fantásticas comigo. Esses dias roubados de férias inventadas foram muito importantes para o andamento deste trabalho.

Aos colegas do laboratório de sistemas embarcados, que ajudaram neste projeto não apenas tecnicamente, mas também com amizade e fartas ofertas de pizza (muitas das quais eu tive que recusar de má vontade por causa deste projeto). Agradeço ao Cláudio e Daniel Walter, que além de me proporcionarem minha primeira experiência profissional quando eu era apenas técnico em informática, ainda me deram todo o tipo de apoio para que eu ingressasse na graduação e continuasse minha formação.

Ao meu orientador e meu coorientador, por estarem sempre dispostos a ajudar e incentivar a minha produção intelectual. Em especial ao Caco, que me convenceu de que ainda pode existir vida acadêmica após a graduação. Também ao meu orientador de Glasgow, Iain Thayne, por ter me inserido no ambiente de pesquisa e em um projeto inspirador e que, mesmo que não tenha se tornado minha área, me fez querer continuar na produção científica.

Agradeço à CAPES por financiar todo o meu intercâmbio e o CNPQ pela oportunidade de realizar a iniciação científica. À UFRGS, por fornecer a educação de qualidade, e principalmente àqueles professores que não apenas estiveram dispostos a compartilhar conhecimento, mas também me inspiraram no aprendizado.

Por fim, agradeço também a todos aqueles que fizeram parte da minha vida durante esses anos de pleno amadurecimento e aprendizagem. Obrigado a todos.

RESUMO

Arquiteturas reconfiguráveis são uma alternativa às clássicas organizações superescalares para explorar o paralelismo em nível de instruções em aplicações (ILP, do inglês Instruction Level Parallelism), enquanto que as organizações multinúcleos são a estratégia normalmente adotada para tirar proveito do paralelismo no nível de threads (TLP, do inglês *Thread Level Parallelism*). Este trabalho é baseado no CReAMS, um sistema reconfigurável que explora os benefícios de ambos o ILP – através de um array reconfigurável – e TLP – por multinúcleos. Contudo, o CReAMS foi inicialmente projetado como um sistema homogêneo, cujos núcleos são exatamente iguais, o que é uma estratégia ineficiente quando considerado que aplicações atuais possuem grande variação de carga entre suas threads. O objetivo deste trabalho é introduzir o conceito de CReAMS heterogêneo, uma versão cujos núcleos possuem recursos distintos. Mostraremos as motivações para usar um sistema heterogêneo ao invés de um homogêneo, especialmente no aspecto das aplicações atualmente executadas nos sistemas embarcados. Este trabalho propõe encontrar algumas configurações heterogêneas de CReAMS que ganhem, em desempenho, de versões homogêneas de mesma área. Mostraremos a metodologia usada para criar tais configurações, o processo de simulação e os resultados, os quais sugerem que sistemas heterogêneos baseados em CReAMS tem melhor performance que versões homogêneas quando as aplicações executadas possuem baixa distribuição de carga entre as threads ou alto potencial para exploração de ILP.

ABSTRACT

Reconfigurable architectures are an alternative for classic superscalar organizations to the exploitation of instruction level parallelism (ILP) on applications, while the multicore organizations are the most commonly used strategy to exploit thread level parallelism (TLP). This work is based on CReAMS, a multicore reconfigurable system that explores the advantages of both ILP – through a reconfigurable array – and TLP – through multicores. However, CReAMS was conceived as a homogeneous system, in which cores are exactly the same. This is an inefficient strategy when we consider current applications that have great variability through its threads. The main goal of this work is to introduce the concepts of heterogeneous CReAMS, a version in which cores have distinct resources. We will show the motivations for using a heterogeneous system over a homogeneous one, especially in the aspect of the current applications executed in the embedded systems. This work proposes to find some heterogeneous configurations of CReAMS that outperform a homogeneous configuration of same area. We will show the methodology for creating such configurations, the simulation processes and the results, which suggest that heterogeneous systems based on CReAMS have better performance than homogeneous versions when the executed applications have poor load balance between threads or have high potential for exploiting ILP.

Keywords: Reconfigurable architectures. Multicore systems. Heterogeneous systems. Homogeneous systems.

LIST OF FIGURES

Figure 1 CReAMS of 8 cores (DAPs) and a L2 shared memory (SM).. 19
Figure 2 (a) DAP blocks (b) Assembly of a loop (c) Allocation inside of reconfigurable datapath..... 20
Figure 3 Interconnection mechanism 21
Figure 4 (a) Heterogeneous threads running on big cores. (b) Parts of the array are wasted............... 24
Figure 5 (a) Heterogeneous threads running on small cores. (b) Bigger threads have to be split 25
Figure 6 Heterogeneous threads running on heterogeneous cores 25
Figure 7 An eight-core homogeneous version of CReAMS and a quad-core heterogeneous version .. 26
Figure 8 Methodology in steps. (A) Configurations creation thought area spreadsheet. (B) Scripts for simulation of each configuration. (C) Simulation of scripts. (D) Compilation of results. (E) Comparison and chart anylisis. 28
Figure 9 Hetero1 vs Homo1: Best and worst cases... 39
Figure 10 Hetero1 vs Homo2: Best and worst cases.. 40
Figure 11 Hetero10 vs Homo5: Heterogenous version is smaller than the homogeneous.................... 42
Figure 12 Hetero8 vs Homo5: Both configurations have the same area.. 43
Figure 13 Hetero8 vs Homo4: Heterogenous version is bigger than the homogeneous 44

LIST OF TABLES

Table 1 Load balancing and mean basic block size of selected applications 30
Table 2 Average number of hops for cores 31
Table 3 Configurations of small cores 33
Table 4 Configurations of medium cores 33
Table 5 Configurations of big cores 34
Table 7 New homogeneous configurations 35
Table 6 Homogeneous configurations 35
Table 8 New heterogeneous configurations (4-7) 36
Table 9 New heterogeneous configurations (8-11) 37

LIST OF ABBREVIATIONS AND ACRONYMS

OS	Operating System
SMS	Short Message Service
UFRGS	Universidade Federal do Rio Grande do Sul
ILP	Instruction Level Parallelism
ARM	Advanced RISC Machine or Acorn RISC Machine
RISC	Reduced Instruction Set Computing
TLP	Thread Level Parallelism
CReAMS	Custom Reconfigurable Arrays for Multiprocessor Systems
DAP	Dynamic Adaptive Processor
SM	Shared Memory
DDH	Dynamic Detection Hardware
ASIP	Application Specific Instruction Set Processors
ASIC	Application Specific Integrated Circuits
GPP	General Propose Processor
FPGA	Field Programmable Gate Array
ALU	Arithmetic and Logic Unit
ISA	Instruction Set Architecture
VLIW	Very Large Instruction Word
API	Application Programming Interface
CPI	Cycles per Instruction
DDH	Dynamic Detection Hardware
CSV	Comma-Separated Values
LUT	Look-Up Table
MLP	Memory-Level Parallelism

TABLE OF CONTENTS

1. INTRODUCTION 11

1.1 Contributions 12

2. RECONFIGURABLE ARCHITECTURES 13

2.1 Basic Principles 13

2.2 Classification 14

3. RELATED WORK 15

3.1 Multithreaded Reconfigurable Systems 15

3.2 Heterogeneous Systems 17

4. CUSTOM RECONFIGURABLE ARRAYS FOR MULTIPROCESSOR SYSTEMS 18

4.1 CReAMS 18

4.2 DAP - Dynamic Adaptive Processors 20
4.2.1 Block 1 – Reconfigurable data path 20
4.2.2 Block 2 – Processor pipeline 22
4.2.3 Block 3 – Storage components 22
4.2.4 Block 4 – Dynamic detection hardware 22

5. HETEROGENEOUS CREAMS 24

6. METHODOLOGY 26

6.1 Simulation environment 29

6.2 Benchmarks 30

6.3 Communication overhead 31

6.4 Dynamic Scheduler 31

7. RESULTS 32

7.1 The configurations created 32

7.2 Simulations Results 38
7.2.1 Simulations without scheduler 38
7.2.2 Simulations with scheduler 40
7.2.2.1 Heterogeneous configuration smaller than the homogeneous 41
7.2.2.2 Heterogeneous and homogeneous configurations with same area 42
7.2.2.3 Heterogeneous configuration bigger than the homogeneous 43

8. CONCLUSION AND FUTURE WORK 45

REFERENCES 46

APPENDIX A 1

APPENDIX B 1

1. INTRODUCTION

For many years, the majority of embedded systems was designed to execute tasks that were specific to the device it was built in. For instance, the system inside an MP3 player was designed to decode digital music files and little else. However, the advance of industry technology pushed by the constant market competition has brought the project of embedded processors to a completely new level of complexity. Nowadays, these systems execute many tasks and comprise as many features as possible in just one device. Current smartphones are a clear example of these devices. Smartphones running Google's Android, Apple's iOS or Microsoft's Windows Mobile OS are capable of executing applications that can not only make calls and send SMS, but also play music, browse the web, monitor user activity and location, take pictures and movies, suggest activities for the user and much more. However, all these embodied features require significant amounts of energy and chip area. The latter is an important issue, considering that the user wants to charge the device only once a day. Thus, each new generation of embedded hardware is expected to not only provide more functionalities and be faster, but also to be more energetically and area efficient.

A processor's performance can be improved in many ways: from its manufacturing process to its organization. This work focuses on optimizing the organization, by exploiting the parallel execution of a program's instructions, as it is a common strategy used to increase performance and efficiently use the resources of a processor. If a sequence of independent instructions (that do not operate over the same data set in a read-after-write fashion) is dispatched, the processor can allocate them to different functional units and process them concurrently. This provides performance gains due to the exploitation of the instruction level parallelism (ILP). The superscalar approach is widely used to exploit ILP in both general purpose – as the Intel x86 architectures – and embedded processors – like the ARM architecture. However, it is expensive in terms of power consumption, since for each incoming instruction block, the superscalar processor has to evaluate repeatedly the data dependencies for the dependence analysis. If the same block is processed multiple times – as in a loop – the superscalar processor needs to evaluate the data dependencies at each execution, because it does not keep any sort of history of these dependencies.

Another possible strategy to exploit ILP is dynamic reconfigurable architectures. They are projected to adapt themselves according to the application at hand, reconfiguring their datapath so that the stream of data through a large set of functional units is optimized. The great advantage of these architectures is that they can also optimize data dependent instructions,

besides executing the non-dependent ones concurrently. Another advantage is that some of these organizations are able to store sequences of instructions with their dependencies already resolved. Thus, differently from the superscalar, the processor does not need to analyze the dependencies each time a block is executed.

Moreover, it is also possible to exploit the parallel execution of program threads. The employment of multicore organizations coupled with the support of the scheduler from the operating system, enables the simultaneous execution of application threads through many processor cores. This is defined as thread level parallelism (TLP) and can be used in any organization, as long as precautions with data consistency in cache and synchronization between cores are taken, which involve hardware support. Therefore, it is also possible to increase performance on reconfigurable organizations through the exploitation of TLP, by the replication of cores that have their own reconfigurable datapath each. CReAMS (Custom Reconfigurable Arrays for Multiprocessor Systems) is an example of such a system.

CReAMS is a reconfigurable system that exploits ILP through an array of functional units capable of adapting itself to execute many instructions concurrently. CReAMS also exploits TLP by replicating its basic core (composed of the processor, the array and additional circuitry to control them), called DAP. Additionally, CReAMS is homogeneous: all cores are the same, which means that the reconfigurable array is also identical on every core.

However, as already mentioned, current applications on embedded processors are quite heterogeneous. Running them on homogeneous multicore systems leads to inefficient usage of the aforementioned architecture. When the cores are not under full usage, many of the functional units on the reconfigurable array are not used, wasting resources and power.

1.1 Contributions

Considering the motivations discussed above, this work will study the potential of using a heterogeneous configuration of CReAMS. A heterogeneous CReAMS differs from the homogeneous by the size of the reconfigurable array in each DAP that composes the system. Although CReAMS was originally created to be homogeneous, two different configurations heterogeneous CReAMS were already studied in [1]. This work will further explore the variation of performance on heterogeneous CReAMS by creating many other versions.

In the first step, we have created many heterogeneous versions of CReAMS considering an area parity with a few fixed homogeneous configurations. We have varied the number of functional units, the input context size and the reconfiguration memory to achieve a good

diversification of configurations. Our goal was to verify the potential of heterogeneous CReAMS by identifying points of performance improvements. Then, we have selected a few of these proposed configurations and simulate them on a variety of benchmarks – chosen to cover a wide range of applications. We show that one can find heterogeneous configurations that have better performance on some applications, but worst in others.

This work is organized as following: in section 2 we discuss about the reconfigurable architectures and their main classifications; section 3 has a collection of related works both in multithreaded reconfigurable systems and heterogeneous architectures; section 4 introduces CReAMS and its organization; section 5 discusses about the heterogeneous version of CReAMS; section 6 shows the methodology used in this work including the simulation environment and the tools used to reach the results; section 7 discusses and shows some of the most interesting results; and, finally, section 8 gives conclusions and the expected future work for this project.

2. RECONFIGURABLE ARCHITECTURES

The architectures that can adapt themselves to provide a hardware expertise for a specific application are known as reconfigurable systems. Because of this specialization, these architectures are expected to provide performance and energy saving improvements. However, these systems are still built aiming flexibility to execute many kinds of tasks, which means that they have smaller gains if compared to dedicated circuits, like Application-Specific Instruction Set Processors (ASIPs) and Application-Specific Integrated Circuits (ASICs) [2].

2.1 Basic Principles

Reconfigurable systems are usually composed of a reconfigurable logic, a controller – to control the logic and the communications – and a context memory – to store the configurations – that are usually coupled to a General Propose Processor (GPP). Pieces of code are executed in the reconfigurable logic while others are executed in the GPP. The more code is executed in the reconfigurable logic, the better, as this will be processed in a more efficient

way. However, this could lead to higher costs for the circuitry: there is need for bigger area and memory to implement the reconfigurable logic.

Reconfigurable systems usually implement the following six steps:

1) *Code Analysis*: The identification of the parts of code that can be transformed to execute in the reconfigurable logic. Usually, these are the pieces of code that are most used, like loops, and are named as hot spots or kernels.
2) *Code transformation:* The kernels are replaced by reconfigurable instructions, which are handled by the control unit in the reconfigurable logic.
3) *Reconfiguration:* The reconfigurable logic is then reorganized to perform the function that the current reconfiguration instruction was designed for. The configuration of the logic is stored in a special memory, called *configuration context*. The time needed to configure the whole system is called *reconfiguration time*, while the memory required for storing the reconfiguration data is called *context memory*. Both these parameters constitute the reconfiguration overhead.
4) *Input Context Loading:* The input operands are fetched. These could come from the register file, a shared memory or transmitted by message passing.
5) *Execution:* The actual execution of the reconfiguration instruction.
6) *Write Back:* The results from the reconfiguration logic execution are written back in the register file or shared memory, or transmitted by message to the reconfigurable control unit or the GPP.

Steps 3 – 6 are repeated while reconfigurable instructions are found in the code.

2.2 Classification

A reconfigurable system can have many classifications, as shown in [2]. Here we will discuss the ones that are most important for our work.

1) *Code Analysis:* The code analysis can be done directly in the binary/source code or in the trace generated by the execution of the program on the GPP. The trace method has the advantage of keeping dynamic information. For instance, the system designer cannot know if loops with non-fixed bounds are the most used ones by only analyzing the static source code. However, the designer can use tools that

dynamically analyze the trace and indicate which are the kernels of the application that are being most executed.

2) *Coupling:* This determines how the reconfigurable unit (RU) is coupled to the GPP. The way the coupling is applied determines how the data transmission and synchronization are done. There are three main places where the RU can be placed relative to the main processor:

- **Attached to the processor:** The reconfigurable logic communicates with the processor by an I/O bus, having to pass through the main memory, which generates high overhead.
- **Coprocessor:** The RU is placed next to the GPP. The communication is usually done using a protocol similar to those used by coprocessors.
- **Functional Unit:** The logic is placed inside the main processor. This way, the RU has full access to the register file of the processor, greatly reducing the communication overhead, but increasing the chip area.

3) *Granularity*: This defines the level of data manipulation of the RU. For fine-grained logic, the smallest blocks that can be configured are usually gates (like on LUTs of FPGAs, they are efficient for bit level operations). On the other hand, coarse-grained RUs have larger blocks (e.g.: ALUs), better suited for bit parallel operations (like bytes, or words).

3. RELATED WORK

In this section, we will show some of the works that were developed and are considered the state of the art in reconfigurable systems with multi core processors. However, as by our knowledge, there are no currently solidified systems that combine reconfigurable architectures and heterogeneous cores. Thus, we present current works on thread scheduling and power consumption on heterogeneous systems.

3.1 Multithreaded Reconfigurable Systems

One can find different single- and multi- processing environments which apply some kind of adaptability to improve the performance of applications [3] [2] [4]. They can be

homogeneous or heterogeneous, considering their architecture (i.e. what ISA is implemented) and organization (i.e. if the processors that comprise the system are the same or not).

Watkins [5] presents a procedure for mapping functions in the ReMAPP system, which is composed of a pair of coarse-grained reconfigurable arrays that is shared among several cores. As an example of a system with homogeneous architecture and heterogeneous organization, one can find Thread Warping [6]. It extends the Warp Processing [7] system to support multiple-thread execution. In this case, one processor is totally dedicated to execute the operating system tasks needed to synchronize threads and to schedule their kernels in the accelerators. ARM's big-LITTLE [8] also implements a heterogeneous organization and homogeneous architecture by grouping a Cortex A7 and a Cortex A15 within the same chip. This strategy aims to provide high performance as well as power efficiency by selecting at run-time the right processor to execute the task at hand according to its requirements.

KAHRISMA [9] is another example of a totally heterogeneous architecture. It supports multiple instruction sets (RISC, 2- 4- and 6-issue VLIW, and EPIC) and fine and coarse-grained reconfigurable arrays. Software compilation, ISA partitioning, custom instructions selection and thread scheduling are made by a design time tool that decides, for each part of the application code, which assembly code will be generated, considering its dominant type of parallelism and resources availability. A run time system is responsible for code binding and for avoiding execution collisions in the available resources.

Both ReMAPP and KAHRISMA are able to optimize multiple threads, but they break the binary compatibility.

CReAMS have the following advantages:

- Unlike KAHRISMA, Thread Warping and big-LITTLE ARM's strategy, this proposal is physically homogeneous in both architecture and organization. Heterogeneity is achieved on the fly, without any human intervention, by employing a binary translation mechanism that will be explained later. It eases the software development process since a well-known tool chain (i.e. gcc) is used for any of its versions. Neither source code modifications nor additional libraries are necessary if new processing elements are inserted.
- KAHRISMA, Thread Warping and ReMAPP rely in special and particular tool chains to extract thread-level parallelism and to prepare the platform for execution. CReAMS approach does not change the current development flow, so well known application programming interfaces (e.g. OpenMP) can be used. This way, the programmer can extract TLP in a friendly interface, since such

APIs are already coupled to a great number of compilers (e.g. gcc and icc), which makes the software development and the binary generation process easier than the aforementioned approaches.

- In contrast to ReMAPP and Thread Warping, CReAMS employs a coarse-grained reconfigurable fabric instead of a fine-grained one. Fine-grained architectures provide higher acceleration levels, but their scope is narrowed to applications that have few kernels responsible for a large part of the execution time. Coarse-grained reconfigurable architectures reduce the reconfiguration time and memory footprint due to the small context size, which increases its field of applications, because they are capable of accelerating the entire application.
- In contrast to ARM´s big.LITTLE strategy, CReAMS does not waste energy rediscovering parallelism like a superscalar does, but rather redefine the data path on the fly for ILP exploitation with minimum energy dissipation.

3.2 Heterogeneous Systems

Most of the works on heterogeneous systems focus on efficient thread schedule, as the real gains in performance and power that these systems present are directly related to the correct assignment of jobs to the cores – as discussed later in section 5. Thus, we show the current work being made on scheduling in heterogeneous systems.

An early work on heterogeneous systems [10] shows that by simply applying the heterogeneity, one can reach significant energy savings with small overhead in performance and area. In this work, the authors have replicated a number of different Alpha processors and the threads are reallocated (by OS decision) accordingly to their necessities. This is a very simple approach and used only as a proof of concept, as only one core is active, so no TLP is exploited.

In [11] the authors propose an algorithm that is implemented directly in the scheduler of the OS. The algorithm uses some metrics to make dynamic scheduling decisions, biasing each thread to be allocated in the most compatible core. However, this approach needs to modify the OS kernel. Moreover, the OS needs to be aware of the resources of each core so it can correctly allocate the threads.

Metrics to decide which core a thread must be allocated to can also be applied in hardware, as in [12]. On that work, the authors use hardware counters to create a profile based

on ILP, MLP and the CPI stack for each thread. This profile is then used to determine whether a thread should be swapped to another core, a decision that is constantly made at small periods of time.

Heterogeneity can be exploited, also, in runtime, as proposed in [13]. On this work, the authors create an out-of-order multicore superscalar processor in which cores are able to polymorph themselves into in-order cores whenever determined to be performance/Watt efficient. Using this approach, many units in a core can be switched off (such as the reorder buffer, load/store queue, etc) at runtime, creating heterogeneity inside the core, which avoids the performance overhead created by thread swapping and reduces energy consumption. Nonetheless, this comes at the cost of leaving many resources of the processor turned off, wasting chip area.

None of these approaches, however, addresses heterogeneity using a reconfigurable organization. The heterogeneous CReAMS brings together all the advantages of the homogeneous version – which includes simultaneous TLP and ILP exploitation, binary compatibility and OS transparency – with the power and area efficiency provided by heterogeneity.

4. CUSTOM RECONFIGURABLE ARRAYS FOR MULTIPROCESSOR SYSTEMS

4.1 CReAMS

The Custom Reconfigurable Arrays for Multiprocessor Systems (CReAMS) is an architecture based on the DAP and was presented in [14]. It is, actually, an extension of the later created to exploit TLP through the replication of the number of DAPs in a system. In this way, the reconfigurable system can now support (and take advantage of) multithreaded applications.

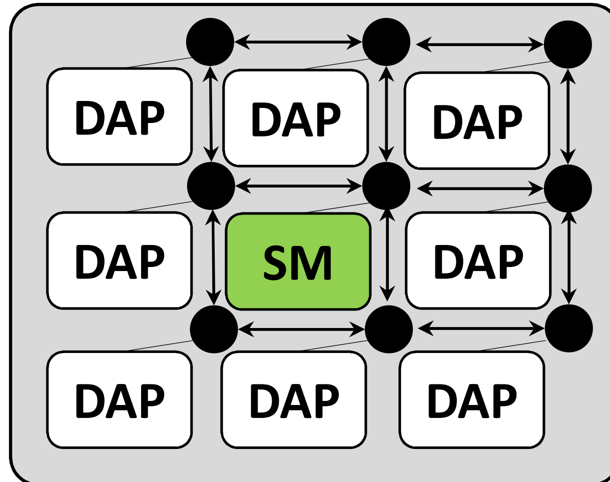


Figure 1 CReAMS of 8 cores (DAPs) and a L2 shared memory (SM)

The communication among the DAPs is done through a 2D-mesh Network on Chip (NoC) using a XY routing strategy. CReAMS also includes an on-chip unified L2 shared memory, illustrated as SM in the Figure 1. As in any multithreaded system, the communication between the cores produces an extra timing overhead. When a core needs to exchange information with another one or with the SM, the data must travel through the hops of the network, which can be just one (if the destination core/SM is just next to the origin) or many. The mean number of hops the data need to go through to reach the destination increases with the total number of cores the system has (as the network becomes bigger), so the higher the number of cores in a configuration, the higher the communication overhead will be. Thus, a communication model must be considered as well during the simulation of CReAMS.

The CReAMS system was originally created to be homogeneous between its cores, which means that all the DAPs on the configuration are exactly the same. They all have the same size in area, the same memory size (L1 cache size, reconfiguration memory table size, number of input and output context...) and the same functional units. This configuration represents the traditional approach to multicore systems – where the DAPs would be the generic cores – and it is simple to implement, as no special scheduling is necessary (a thread is simply allocated to the next free DAP). However, this is not the most efficient approach, whereas a low-duty thread will execute on the same environment as a heavy-duty one.

4.2 DAP - Dynamic Adaptive Processors

The DAP is a reconfigurable architecture tightly coupled to the processor and was presented in [15]. It is a transparent coarse-grained architecture – it is a reconfigurable datapath composed of functional units for word-level operations. To better explain the DAP, we divided it into four blocks, as shown in Figure 1(a).

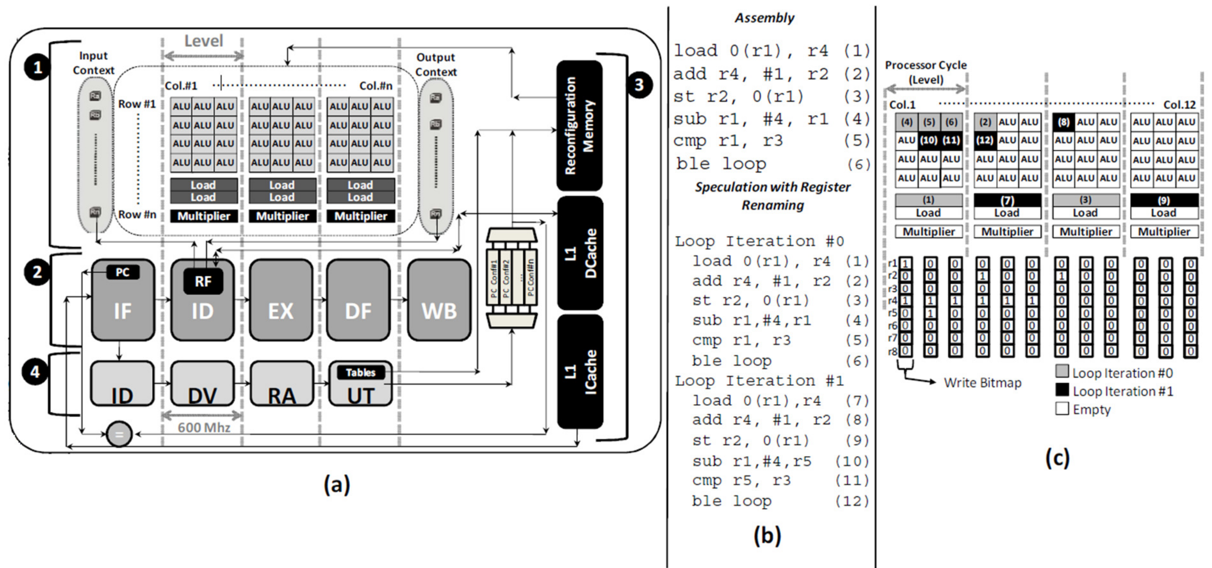


Figure 2 (a) DAP blocks (b) Assembly of a loop (c) Allocation inside of reconfigurable datapath

4.2.1 Block 1 – Reconfigurable data path

Block 1 shows the structure for the reconfigurable data path. It is coarse-grained and tightly coupled to the processor's pipeline (there is no shared bus between them), which removes the necessity of external access to the memory and, consequently, saves power and reduces the reconfiguration time. As we can see in Figure 2(a), the data path is organized in a matrix structure, where the number of rows is the maximum number of instructions that can be executed in parallel. Independent instructions can be allocated on the same column. Therefore, the number of columns limits the ILP exploitation. Additionally, the number of columns dictates the maximum number of dependent instructions that can be executed in sequence in a configuration – the columns in one level are executed in sequence as a big combinational block. For example, the configuration in Figure 2 performs up to four arithmetic and logic operations, two memory accesses on cache and one multiplication at the same time, if all the data instructions are independent. As the critical path (the piece of combinational circuit that takes longer to produce a correct result) is the multiplier, it is possible to have other faster units in the

same level. In the above example, three arithmetic and logic units (ALUs) compose a level, while the multiplier and the memory access take the equivalent to one cycle of the processor. In other words, this configuration can execute twelve arithmetic and logic operations, two memory accesses and one multiplication on each level (within one clock cycle) at the very best case.

The entire structure of the reconfigurable data path is combinational, meaning that there is no temporal barrier (registers) between the functional units. The only registers present are at the entry point – the input context – and at the exit point – as temporary storage of the results. The feeding of the input context with the necessary data is the first step to configure the data path before starting the execution. The results are sent to the processor's register file on demand. It means that if any value is produced at any data path level (a cycle of the processor) and if it will not be changed in the next levels, this value can be written back on the next cycle. If the number of writes produced by the array is greater than the number of available write ports in the register file, than the excess instructions are forwarded to the next level. In the example shown in Figure 2(a), the maximum number of ports available is two.

Figure 2 shows a simplified overview of the interconnection structure of the reconfigurable data path. Bus lines connect the input context with the functional units and the output context, while multiplexers are responsible for choosing the path this data will run. The input multiplexers – two for each functional unit – will decide which are going to be the input registers for a specific functional unit (in the Figure 3, an ALU). The output multiplexers, on the other hand, will select if the output will be provided directly from the input context registers or from a previous functional unit. The control signals for these multiplexers are stored on the reconfiguration cache.

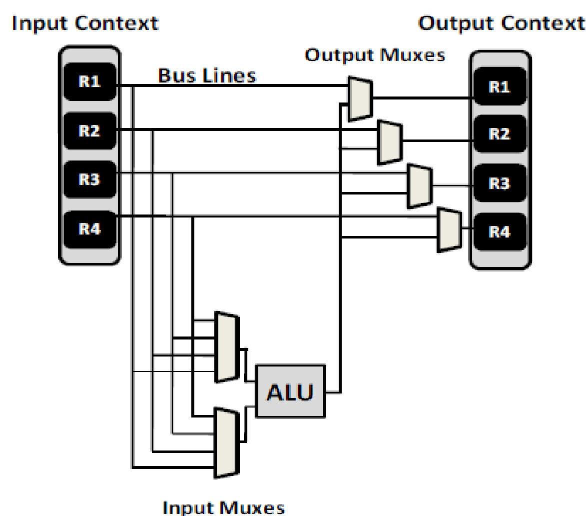


Figure 3 Interconnection mechanism

4.2.2 Block 2 – Processor pipeline

Block 2 is the basic processor coupled to the array. It is also the reference for comparisons on the simulation performance. In this work, the baseline processor is a SparcV8-based architecture. Its five stage pipeline reflects a traditional RISC design (instruction fetch, decode, execution, data fetch and write back) and is similar to other RISC processors used in well-known embedded platforms (e.g. MIPS, ARM).

4.2.3 Block 3 – Storage components

There are two memory units specialized for the reconfiguration system: the address cache and the reconfiguration memory. The first holds the memory address of the first instruction of every configuration built by the dynamic detection hardware (explained later). This cache is implemented as a 4-way set associative table containing 64 entries (which means that the system can hold up to 64 configurations). An address cache hit indicates that a configuration was found, therefore this cache is used to verify the existence of a configuration and to point where it is stored on the reconfiguration memory.

The reconfiguration memory holds the bits for each configuration saved. Each bit is a control bit for the output and input multiplexers. They indicate which functional unit will be active and which register will be read as operators.

4.2.4 Block 4 – Dynamic detection hardware

The DDH is a binary translation mechanism that turns the instructions from SparcV8 ISA to reconfigurable array configurations. This block is responsible for instruction detection and allocation in the data path and it is implemented as a 4-stage pipelined circuit. The stages of the circuit are divided in Instruction Decode (ID), Dependence Verification (DP), Resource Allocation (RA) and Update Tables (UT). The translation process is performed as the processor executes the instruction (at the same time and independently), so there is no extra performance overhead, which means that it does not increase the processors critical path, leaving operating frequency unchanged.

For each column of the reconfiguration data path (Figure 2(a)), there is a bitmap responsible for storing in the target operands of the already allocated instructions in the

respective column, named as Write Bitmap (Figure 2(c)). Thus, for each incoming instruction, its source operands will be compared to the target operands in this bitmap to decide in which column this instruction will be allocated, according to data dependencies.

Figure 2(b) has an assembly code as an example. On Figure 2(c), the allocation of this code on the reconfigurable data path is shown. The first incoming instruction, a memory access, is allocated on the highest functional unit of the leftmost data path column. However, as in this case this type of operation takes an entire level (a processor cycle), the fourth bit of the write bitmap (representing the r4 register) of the columns 1, 2 and 3 are set to maintain the allocation consistency.

The dependency detection starts from the second instruction. In the example, the instruction number two reads the r4 register. As it is written by the previous instruction, a read after write (RAW) dependence is found. The DDH detects it (through the write bitmap) and allocates the instruction number two at the later column of instruction number one. The second bit of the fourth column of the write bitmap is set since this instruction has the register r2 as the target operand. The dependency analysis keeps these steps until instruction number five, where a loop is found.

The DDH supports speculation, so when the branch instruction is found, a speculation flag is set and the configuration continues the allocation of the following iterations. In other words, it is possible (if there is enough space on the array) to keep multiple iterations of a basic block on the same configuration. The instructions in black on Figure 2(c) represents the instructions allocated for the second iteration of the loop code of Figure 2(b).

This hardware is capable of performing register renaming, resolving false dependencies. In instruction number ten, the register r1 could be read by the incoming instruction in the second column, but could not write in this same register at this column. This is detected by the DDH and the register is renamed to r5 (the next empty register of the input context). All subsequent instructions that contain a reference to r1 are modified accordingly.

5. HETEROGENEOUS CREAMS

A heterogeneous version of CReAMS is a configuration where each DAP has a different number of functional units, input and output context length and memory size. This allows for some DAPs to be bigger than others (i.e.: some will be more efficient to execute threads that can exploit higher levels of instruction parallelism). Similarly, the smallest DAPs would be allocated to run jobs with low ILP.

For instance, in Figure 4(a) we illustrate two heterogeneous threads. The one in the left is bigger, meaning that it has larger basic blocks and has more potential for ILP exploitation. The one in the right is smaller, so its kernels are composed of groups of fewer instructions. On this case, both threads are being scheduled on cores that have large arrays. The processor will be able to execute both threads; however, the core that is receiving the smaller thread will not use most of its functional units on the reconfigurable datapath – as illustrated on Figure 4(b) – simply because the configurations this thread generates do not require all the resources available. This waste of resources also generates power leakage. Similarly, on Figure 5(a) we show an example where the same heterogeneous threads are executed in small cores. Again, the CReAMS processor will be able to run both threads, however – as illustrated in Figure 5(b)

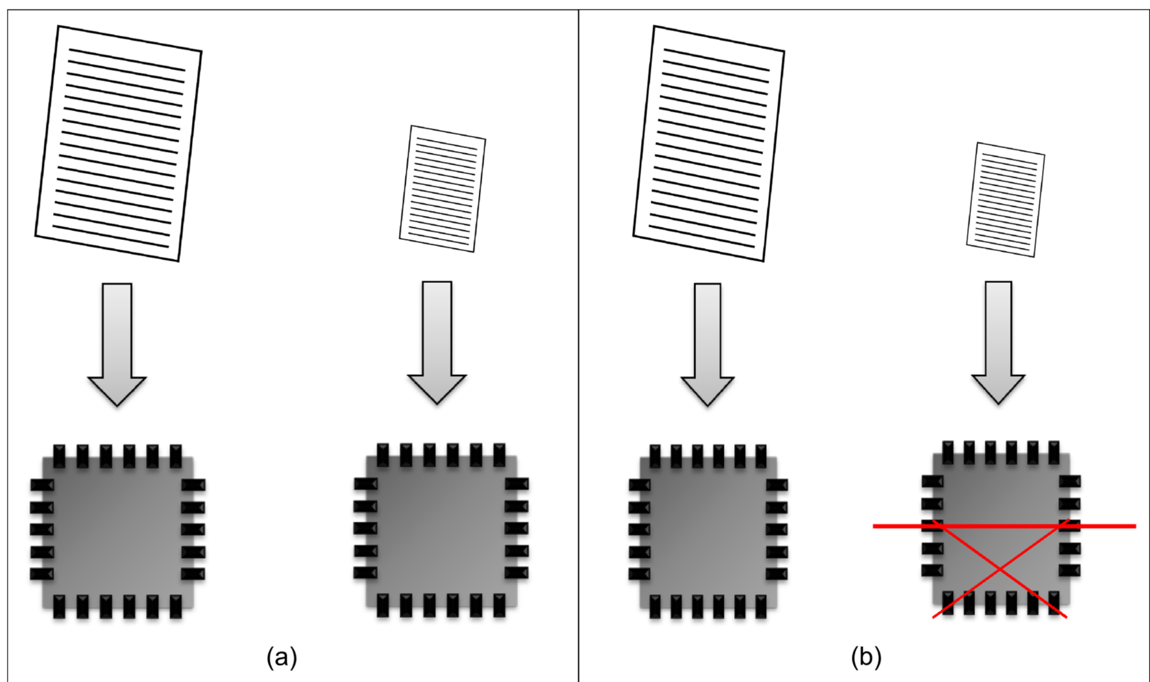


Figure 4 (a) Heterogeneous threads running on big cores. (b) Parts of the array are wasted.

–, the DDH will have to split the kernels of the larger thread. As this thread has larger basic blocks, it will generate bigger configurations. The reconfigurable array, however, does not have enough resources to execute the configuration in one cycle, so they have to be split to fit the smaller array. This thread will need more cycles to finish, which means that it will take more time to execute and, consequently, will consume more power.

We aim to achieve a configuration like the one illustrated on Figure 6, where the larger threads are scheduled to bigger cores and smaller threads to smaller cores. This will lead to increased efficiency in processing heterogeneous applications.

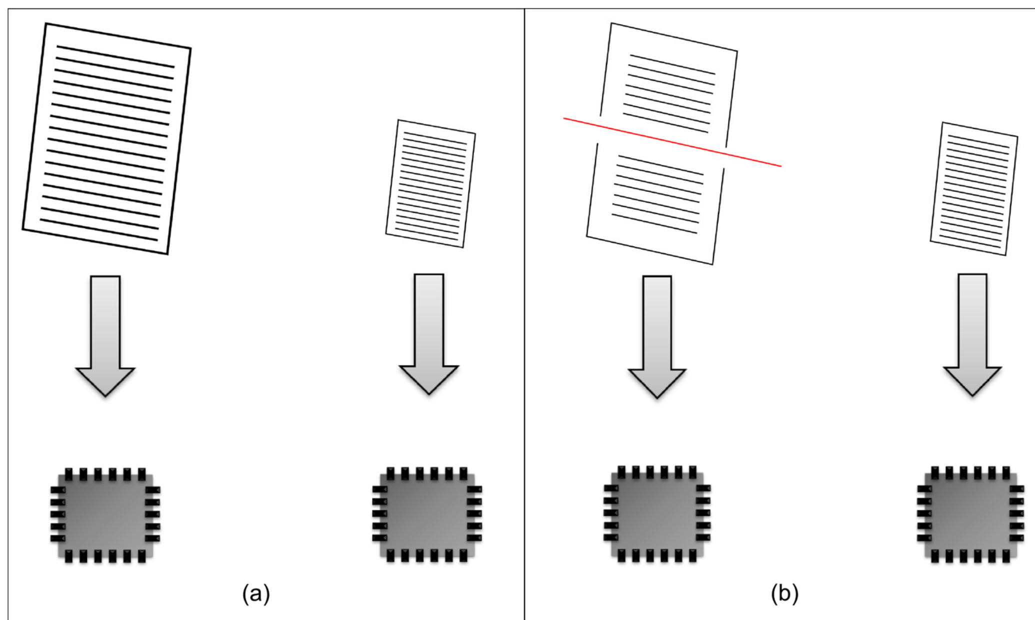


Figure 5 (a) Heterogeneous threads running on small cores. (b) Bigger threads have to be split .

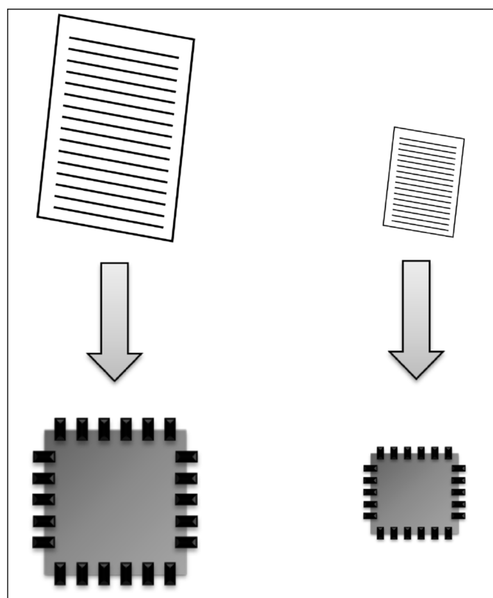


Figure 6 Heterogeneous threads running on heterogeneous cores

Figure 7 shows an example comparing an eight-core homogeneous CReAMS of medium cores with a quad-core heterogeneous CReAMS composed of two big cores, one medium and one small. On this illustration, both processors have the same area, as the large cores on the heterogeneous configuration compensate the area occupied by the extra medium cores on the homogeneous.

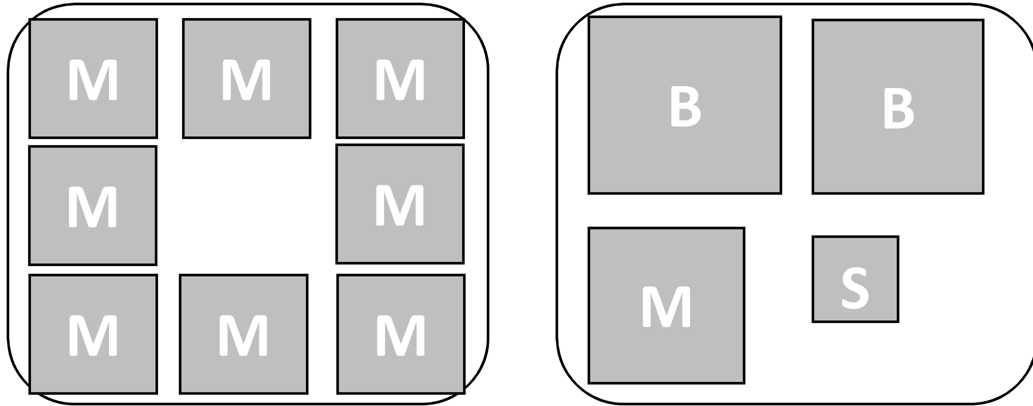


Figure 7 An eight-core homogeneous version of CReAMS and a quad-core heterogeneous version

This configuration, however, leads to some problems that need to be addressed carefully. For instance, how to determine the best sizes of DAPs to be used and the best combination – should we have many small sized DAPs or more big sized ones – that leads to better performance if compared to the homogeneous configuration.

6. METHODOLOGY

The first step of this work was to create many heterogeneous configurations and then choose the most suitable for testing. As the CReAMS simulator is already very generic when it comes to the resources configurations, no changes on it were necessary. The simulator works by emulating each DAP, using configurations that are passed as parameters. Then, it compiles the information (e.g. cycles taken, cache access) of each core and outputs a result file. Therefore, at first, to simulate heterogeneous configurations we only needed to create them and forward them to the simulator accordingly. Later on, a modification was made to simulate a dynamic scheduler as well. This scheduler is important to allocate the threads accordingly to their ILP requirements.

To create the configurations, a spreadsheet was used. This spreadsheet is part of the CReAMS project and it contains the area occupied for each of the CReAMS components. It

allows the user to vary the number of functional units on the datapath, the dimensions of the array and the input context length and receive the expected area occupied by this configuration – considering the number of cores as well. This spreadsheet was also modified to be able to vary the size of the reconfiguration cache – giving how many configurations the cache can hold –, a resource needed to create finer area parity combinations, as we will discuss later.

The latter tools discussed were already developed and from now on we show what was added on the process through this work. The next step was to create many scripts to run automatically the simulations. Each configuration needs a script for each application and number of cores. Thus, if there is a combination of five applications with five number of cores, each configuration needs a total of 25 scripts. Furthermore, to run all these jobs automatically, another script was created to iterate over the number of cores and call all the necessary simulations.

The second phase of tests was focused on two points: to compare versions of heterogeneous CReAMS that would be smaller than the homogeneous configurations; and to insert a dynamic scheduler, so we could analyze the impact of intelligent allocation on performance

The former is easily achieved by creating new heterogeneous and homogeneous configurations. The CReAMS simulator, also, already supported dynamic scheduling, so the only need was to create new simulation scripts with the correct parameters.

Figure 8 shows the steps from the creation of each configuration to the analysis of the results. On step (A), we use the spreadsheet that calculates the area of a CReAMS system to create the many configurations of same area. Step (B) uses each of these configurations to create many scripts for the selected applications, which are then simulated on step (C). The results of all the simulations are then compiled in a single file in step (D). Finally, we compare the configurations that have same area and format this data in a chart, for analysis.

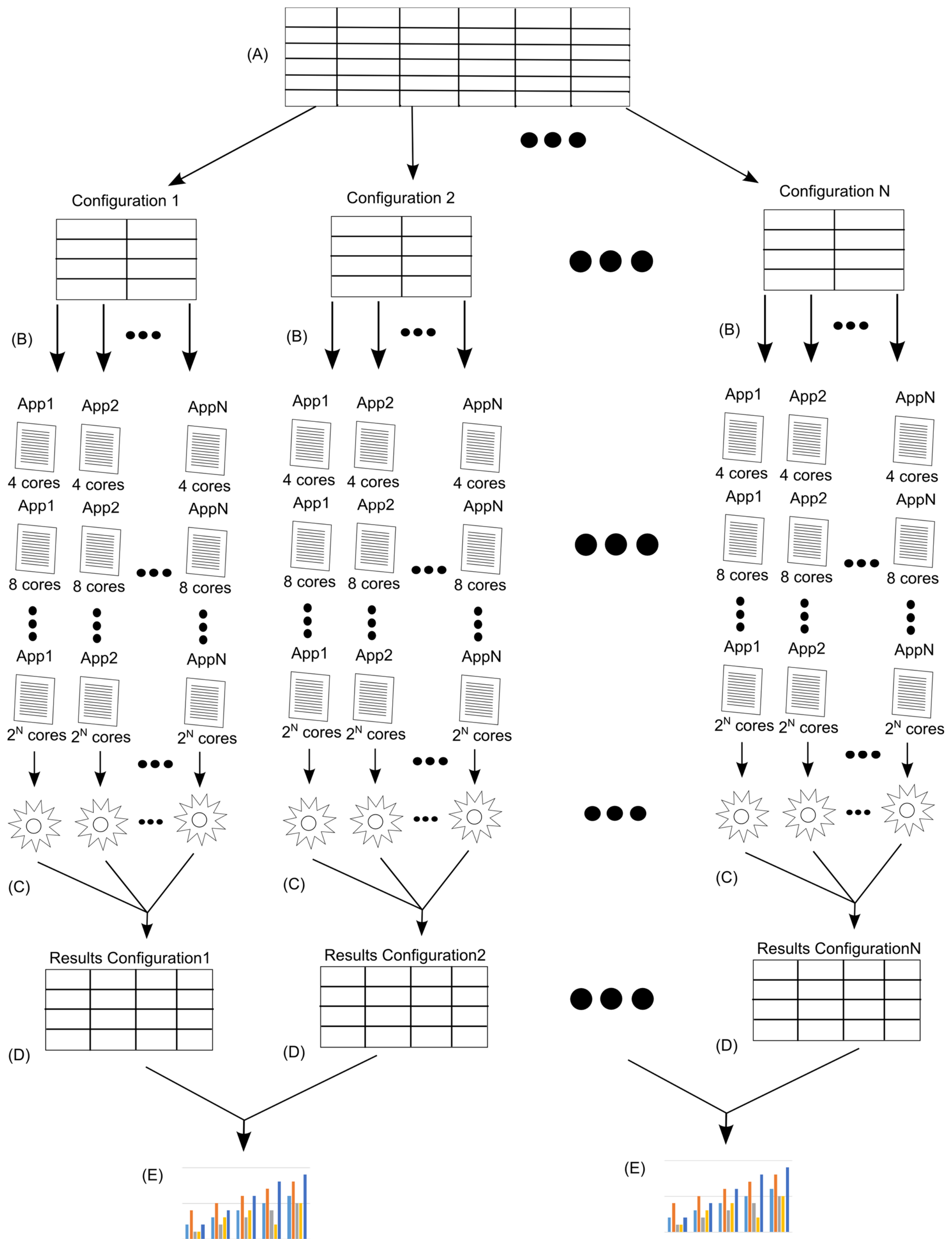


Figure 8 Methodology in steps. (A) Configurations creation thought area spreadsheet. (B) Scripts for simulation of each configuration. (C) Simulation of scripts. (D) Compilation of results. (E) Comparison and chart anylisis.

6.1 Simulation environment

This work has used the Simics simulator [16], an instruction level simulator. Simics uses an environment, which is a set of configurations to specify the characteristics of the hardware and the software of the simulated system, to extract the instructions executed by a specific processor. For this work, we have used an environment with a Linux Ubuntu operating system running over a single SparcV8 processor. The benchmarks used were created using OpenMP and Pthreads, thus the number of threads spawned at run-time are user arbitrary, even if the system uses a single processor.

Simics produces a sequencial trace of instructions and data accesses for all threads. As they are all mixed by the simulator, special instructions marks were added to indicate where each thread trace starts and ends. The trace generated by Simics is then sent to a python script, named as Splitter. This script recognizes the instruction marks and splits the trace file, generating many new files, one for each running thread of the application.

Mkfifo is a UNIX process that manages automatically a first-in-first-out (FIFO) behavior. The Simics simulator and the Splitter communicate in a producer-consumer way, so this FIFO process was inserted to control such communication. The split files created by the Splitter are also mkfifo processes, as they communicate with the DAPs in a producer-consumer behavior as well.

For each thread that was created, an instance of a DAP simulator is used (in this case, a CReAMS simulator). Each DAP consumes the instructions sent by the Splitter and produces a file with results for performance, communication among threads and energy consumption. The Backward process is activated at the end of the simulation. It compiles the results that are generated for each thread in a single file, providing the results of the whole application simulation.

As the simulations are usually made for many core variations (from 4 to 64 cores) and each of them outputs results for the best and worst case of communication, a final python script is executed to merge all the results. It receives the name of the application and a code to identify the simulation batch as parameters. With this information, the script recursively searches for all the results of this batch and compiles a single, CSV formatted, file that can be easily imported to a spreadsheet software.

6.2 Benchmarks

In order to measure the performance of CReAMS, 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 five benchmarks chosen were *lu* and *fft* from parallel suites [17] [18], *equake* from the SPEC OMPM2001 [19] package and *susan edges* and *susan corners* from the MiBench suite [20]. In [1] a study was made to characterize the potential of these applications on obtaining performance improvements when TLP or ILP exploitation is applied. The mean basic block size gives us some clues about the limits of ILP that the selected application may provide. In addition, the percentage of the entire application code that is executed in parallel, when multithreaded application environment is considered, is an important metric to obtain the upper bound for TLP exploitation in the applications. This metric is also called load balancing and it gives the TLP potential for the application: the more an application can distribute its work through its threads, the more it will be able to take advantage of a multicore system. Table 1 shows the results of this study.

FFT is not shown on the table study, however, as will be discussed later, the results suggests that it has good load balancing, as for the homogeneous simulations, it has gradually improved performance with the increase of cores.

Table 1 Load balancing and mean basic block size of selected applications

Benchmark	Mean BB Size(#instr)	Load Balancing (%) in threads			
		4	8	16	64
equake	4,80	18,49	10,32	5,10	0,92
susan_e	16,60	39,80	24,90	4,80	0,90
susan_c	17,36	67,58	49,18	34,94	12,50
lu	8,32	82,20	56,77	29,35	7,03

Table 1 shows that equake is a benchmark with small basic blocks, meaning that ILP is poorly exploited on this application. Moreover, it has also a very poor load balancing between threads, which worsens each time the number of cores is increased. That also suggests that equake is not a TLP oriented application. Overall, poor or none advantages are expected when running equake on a system that focus on ILP and TLP exploitation.

Similarly, lu has small basic blocks but good load balancing when the number of cores is small. Susan edges and susan corners have both big basic blocks, however susan corners has a better load balancing, meaning that we should see it performing better on configurations that have more cores. Susan edges, however, should perform better on systems with more resources on the arrays – consequently, which have less cores, as we work with area parity.

6.3 Communication overhead

As any multicore system, the cores of CReAMS must communicate between themselves to perform synchronization. In section 4.1 we have discussed that the communication between the DAPs are done through a NoC using the XY strategy, which introduces an overhead each time the cores have to change information. This overhead depends on the size of the mesh (which is directly dependent on the number of cores) and the distance between cores, i.e., the number of router the information must go through between origin and destination.

In [1] a model for the corner cases of communication in CReAMS is proposed. It consider the best and worst cases of communication overhead, being the best case when the traffic of data is uniformly distributed among the NoC nodes and the worst when data is concentrated in a specific node. This is the average number of hops and the model calculates it 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
$$AvgHops = \frac{k^2}{k+1}$$

where $h = \sqrt{N}$

Table 2 shows the number of hops for a single communications process considering the formulas presented above.

Table 2 Average number of hops for cores

	4 Cores	8 Cores	16 Cores	32 Cores	64 Cores
Distributed	1.33	1.88	2.66	3.77	5.33
Centralized	1.33	2.09	3.20	4.81	7.11

6.4 Dynamic Scheduler

On the first extracted results, the simulator used a static scheduler for the DAPs, meaning that once a thread starts running in a core, it will end its execution on the same core. Dynamically scheduling threads to work on the most efficient cores have been proved to be one

of the main advantages of heterogeneous systems [21]. Thus, for the second phase of simulations, a dynamic thread scheduler (proposed on [1]) was applied.

This scheduler keeps instruction counters for each thread on the reconfiguration memory and based on this data, it changes the threads to the cores that most suits their needs. Thus, threads that contain more instructions on their configurations will be allocated to cores that have more resources, while threads that have small load will be sent to cores with small arrays. The main goal of this algorithm is not to employ the best scheduling possible, but to verify if performance losses provided by the heterogeneous CReAMS in some applications are due to wrong thread scheduling.

7. RESULTS

In this section, we will present the results obtained by this work. We start with all the heterogeneous and homogeneous configurations that were created. Then, we show the ones that were chosen to be simulated. Finally, we present the results of the simulations, both with and without a scheduling mechanism.

7.1 The configurations created

To create new configurations of CReAMS, one can change the resources on the array, the input context size or the length of the reconfiguration memory. However, these characteristics are also connected. If the array is greatly expanded, but the input context stays short, just a few registers will be able to be addressed by the array at each cycle, so the system will take many cycles to fill the context. Moreover, if the reconfigurable cache is not expanded as well, the system will not be able to hold big configurations, in other words, the array will not be fully used during a reconfiguration process. Thus, to select the best configurations, we had to consider a good balance between these characteristics of the system, as we will discuss later.

A great number of configurations were created considering area parity. We started creating configurations that would have parity with a certain number of SparcV8 processors (the basic processor of CReAMS) and then combine these cores to compare them with homogenous versions of CReAMS. This strategy lead to 15 configurations of small cores arrays, 22 configurations of medium cores and other 22 of large cores, shown in tables 3, 4 and 5. Combined over, we could use 15x44 variations to create configurations using to sizes of core and 15x22x22 to create configurations with the three sizes. This would lead to almost 8000

configurations to simulate. Furthermore, ours simulations were made on five benchmarks (discussed later) from with five different core sizes, so for each configuration, a total of 25 simulations were needed. As some of the benchmarks can take several hours to execute, the simulation of all the configurations created was not practicable and just a few of them were selected.

Table 3 Configurations of small cores

Small Core	SC1	SC2	SC3	SC4	SC5	SC6	SC7	SC8	SC9	SC10	SC11	SC12	SC13	SC14	SC15
Lines	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
ALU	4	2	1	2	3	8	3	2	4	6	10	4	2	5	8
Load/Store	1	7	2	2	2	4	12	5	3	4	4	16	6	5	6
Multiplier	1	1	6	2	1	1	3	9	3	2	2	4	12	5	4
Input Context	4	6	4	10	8	8	12	12	20	12	16	16	16	24	16
Sparcs (Parity)	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4

Table 4 Configurations of medium cores

Medium Cores	MC1	MC2	MC3	MC4	MC5	MC6	MC7	MC8	MC9	MC10	MC11
Lines	15	15	15	15	15	15	15	15	15	15	15
ALU	2	1	4	3	1	2	4	7	3	1	3
Load/Store	1	2	2	8	3	2	3	3	10	4	2
Multiplier	1	2	1	1	6	2	1	1	3	10	2
Input Context	15	15	15	15	15	28	15	15	15	15	28
Parity	2	2	3	3	3	3	3	4	4	4	4
	MC12	MC13	MC14	MC15	MC16	MC17	MC18	MC19	MC20	MC21	MC22
	15	15	15	15	15	15	15	15	15	15	15
	6	12	6	2	4	8	12	5	2	5	10
	4	6	18	8	4	6	8	20	10	5	8
	2	4	2	15	4	4	2	5	16	5	4
	15	15	15	15	38	20	22	22	22	46	24
	4	6	6	6	6	6	8	8	8	8	8

Table 5 Configurations of big cores

Big Cores	BC1	BC2	BC3	BC4	BC5	BC6	BC7	BC8	BC9	BC10	BC11
Lines	24	24	24	24	24	24	24	24	24	24	24
ALU	2	1	5	2	1	2	4	8	2	2	3
Load/Store	1	2	1	6	3	2	3	2	12	4	3
Multiplier	1	2	1	2	6	2	1	1	2	8	3
Input Context	24	24	24	24	24	42	24	24	24	24	42
Parity	4	4	6	6	6	6	6	8	8	8	8

Big Cores	BC12	BC13	BC14	BC15	BC16	BC17	BC18	BC19	BC20	BC21	BC22
Lines	24	24	24	24	24	24	24	24	24	24	24
ALU	6	10	3	1	4	8	16	4	1	5	11
Load/Store	4	4	16	4	4	6	4	20	8	5	9
Multiplier	2	1	2	14	4	3	1	4	18	5	4
Input Context	24	32	32	32	54	32	32	32	32	62	32
Parity	8	12	12	12	12	12	16	16	16	16	16

We have analyzed and selected the best configurations on the three tables to create a few heterogeneous versions of CReAMS. To select these configurations, we have considered those which had a good balance between the functional units. For instance, SC3 in Table 1 has many multipliers and just a few ALUs and memory access units. It is well known that most programs use more ALU operations than multiplications, furthermore, the lack of load/store units can highly compromise the performance of an application, as each time those units fully occupied, the reconfiguration datapath will need to wait for the next cycle to execute them. On the other hand, SC7 has too many load/store units, which will probably be wasted and just occupy a great amount of area, as most of the applications do not need to access memory so often [20].

The final heterogeneous configurations tested were based on a small core version (SC6), a medium core (MC7), and a big core (BC7). Those configurations have a good balance between their functional units and input context length. From these, we have created other three version of heterogeneous cores. One of them with 50% of big cores, 25% of medium cores and 25% of small cores (called Hetero1), another with 25% of big cores, 50% of medium cores and 25% of small cores (the Hetero2) and the last one with 25% of big cores, 25% of medium cores and 50% of small cores (the Hetero3). For instance, a 4-core version of Hetero1 would have two big cores, one medium core and one small core, while the Hetero2 would have one big, two medium and one small. Those three configuration allowed us to analyze the impact of the sizes of the cores over the applications.

Homogeneous versions of CReAMS were also simulated, so we could compare the results. However, these results were already done on the first part of this project (see

APPENDIX A), hence they were reused. Nonetheless, for convenience, we show these configurations on Table 7. Homo1 is a small version of CReAMS – if compared with the size of the cores on the heterogeneous versions, it would be classified as small – while the Homo2 is a medium version.

For the second phase, which we inserted the scheduler in the simulations, we have created three new homogeneous configurations of CReAMS focusing on having more size variation on the cores. These configurations are listed on Table 6.

Similarly, we have created five new heterogeneous (Hetero 4, 5, 6, 7 and 8) configurations and re simulated the same heterogeneous versions from the first phase (Hetero 9, 10 and 11), but now with the scheduler. We wanted to analyze the behavior of these configurations and what changed when the scheduler was inserted. These versions are shown in Table 8 and Table 9. These were fine adjusted to have similar area if compared with the homogeneous versions of Table 6. On the bigger cores, we have increased and decreased the size of the reconfiguration cache, because theses cores already have enough functional units to exploit ILP and increasing them even further would not result in great performance increase (as shown on the first part of this work, in APPENDIX A). However, medium and smaller cores still have room for ILP exploitation, so we have fine adjusted their functional units.

In Table 6 the line "Total Sparcs" represents the number of SparcV8 processors that would fit (that have the same area as) one core of the homogeneous configuration. In Table 8 and Table 9 the line "Total 4Core Configuration Sparc" represents the number of SparcV8 processors that would fit a 4-Core version of the heterogeneous configuration. These numbers are used as basis for our area parity.

Table 7 Homogeneous configurations

Configuration	Lines	ALU	Load/Store	Multipliers	Input Context
Homo1	9	3	1	2	8
Homo2	15	4	4	2	16

Table 6 New homogeneous configurations

Homogeneous					
Name	Homo3	Name	Homo4	Name	Homo5
Total Sparcs	2	Total Sparcs	4	Total Sparcs	8
Lines	6	Lines	15	Lines	21
ALUs	4	ALUs	5	ALUs	6
Load/Stores	3	Load/Stores	4	Load/Stores	5
Muls	2	Muls	2	Muls	4
Input C.	8	Input C.	16	Input C.	24
Cache Rec	32conf	Cache Rec	64conf	Cache Rec	128conf

Table 8 New heterogeneous configurations (4-7)

Hetero4							
Subname	BigCores		Subname	MedCores		Subname	SmallCores
Proportion	50%		Proportion	25%		Proportion	25%
Total Sparcs	12		Total Sparcs	6		Total Sparcs	2
Lines	27		Lines	15		Lines	6
ALUs	7		ALUs	8		ALUs	3
Load/Stores	5		Load/Stores	5		Load/Stores	2
Muls	4		Muls	4		Muls	1
Input C.	28		Input C.	16		Input C.	8
Cache Rec	128conf		Cache Rec	64conf		Cache Rec	32conf
						Total 4Core Configuration Sparcs:	32
Hetero5							
Subname	BigCores		Subname	MedCores		Subname	SmallCores
Proportion	25%		Proportion	50%		Proportion	25%
Total Sparcs	6		Total Sparcs	4		Total Sparcs	2
Lines	21		Lines	15		Lines	6
ALUs	5		ALUs	5		ALUs	4
Load/Stores	4		Load/Stores	4		Load/Stores	3
Muls	2		Muls	2		Muls	2
Input C.	24		Input C.	16		Input C.	8
Cache Rec	64conf		Cache Rec	64conf		Cache Rec	32conf
						Total 4Core Configuration Sparcs:	16
Hetero6							
Subname	BigCores		Subname	MedCores		Subname	SmallCores
Proportion	25%		Proportion	25%		Proportion	50%
Total Sparcs	8		Total Sparcs	4		Total Sparcs	2
Lines	21		Lines	15		Lines	6
ALUs	6		ALUs	5		ALUs	4
Load/Stores	5		Load/Stores	4		Load/Stores	3
Muls	4		Muls	2		Muls	2
Input C.	24		Input C.	16		Input C.	8
Cache Rec	128conf		Cache Rec	64conf		Cache Rec	32conf
						Total 4Core Configuration Sparcs:	16
Hetero7							
Subname	BigCores		Subname	MedCores		Subname	SmallCores
Proportion	25%		Proportion	50%		Proportion	25%
Total Sparcs	14		Total Sparcs	7		Total Sparcs	4
Lines	21		Lines	15		Lines	6
ALUs	9		ALUs	9		ALUs	10
Load/Stores	7		Load/Stores	7		Load/Stores	8
Muls	5		Muls	4		Muls	5
Input C.	32		Input C.	20		Input C.	16
Cache Rec	128conf		Cache Rec	64conf		Cache Rec	32conf
						Total 4Core Configuration Sparcs:	32

Table 9 New heterogeneous configurations (8-11)

Hetero8					
Subname	BigCores	Subname	MedCores	Subname	SmallCores
Proportion	25%	Proportion	25%	Proportion	50%
Total Sparcs	16	Total Sparcs	8	Total Sparcs	4
Lines	21	Lines	15	Lines	6
ALUs	10	ALUs	10	ALUs	10
Load/Stores	8	Load/Stores	8	Load/Stores	8
Muls	6	Muls	5	Muls	5
Input C.	32	Input C.	20	Input C.	16
Cache Rec	128conf	Cache Rec	64conf	Cache Rec	32conf
				Total 4Core Configuration Sparcs:	32
Hetero9					
Subname	BigCores	Subname	MedCores	Subname	SmallCores
Proportion	50%	Proportion	25%	Proportion	25%
Total Sparcs	6	Total Sparcs	3	Total Sparcs	2
Lines	24	Lines	15	Lines	9
ALUs	4	ALUs	4	ALUs	3
Load/Stores	3	Load/Stores	3	Load/Stores	2
Muls	1	Muls	1	Muls	1
Input C.	24	Input C.	12	Input C.	8
Cache Rec	128conf	Cache Rec	64conf	Cache Rec	32conf
				Total 4Core Configuration Sparcs:	17
Hetero10					
Subname	BigCores	Subname	MedCores	Subname	SmallCores
Proportion	25%	Proportion	50%	Proportion	25%
Total Sparcs	6	Total Sparcs	3	Total Sparcs	2
Lines	24	Lines	15	Lines	9
ALUs	4	ALUs	4	ALUs	3
Load/Stores	3	Load/Stores	3	Load/Stores	2
Muls	1	Muls	1	Muls	1
Input C.	24	Input C.	12	Input C.	8
Cache Rec	128conf	Cache Rec	64conf	Cache Rec	32conf
				Total 4Core Configuration Sparcs:	14
Hetero11					
Subname	BigCores	Subname	MedCores	Subname	SmallCores
Proportion	25%	Proportion	25%	Proportion	50%
Total Sparcs	6	Total Sparcs	3	Total Sparcs	2
Lines	24	Lines	15	Lines	9
ALUs	4	ALUs	4	ALUs	3
Load/Stores	3	Load/Stores	3	Load/Stores	2
Muls	1	Muls	1	Muls	1
Input C.	24	Input C.	12	Input C.	8
Cache Rec	128conf	Cache Rec	64conf	Cache Rec	32conf
				Total 4Core Configuration Sparcs:	13

7.2 Simulations Results

7.2.1 Simulations without scheduler

On the first part of this work, we have simulated the homogeneous versions Homo1 and Homo2 and the heterogeneous versions Hetero1, Hetero2 and Hetero3. The main goal was to analyze the potential of having heterogeneous cores of CReAMS without worrying about the thread scheduling. The simulations are presented with best and worst case of communication and are compared with the percentage of speedup a configuration has achieved over another. Positive speedups means that the heterogeneous configuration was faster, while negative percentages means that the homogeneous version was faster. We have selected the most interesting cases to discuss and those that do not have similar behaviors. The complete list of results can be found on APPENDIX B.

Figure 9 shows the results for the worst and best cases of communication between the Hetero1 and Homo1 configurations. It can be seen that the communication overhead negatively affects the performance of the configuration with more cores (in this case, the homogeneous), so the difference between the cases are a small increase in performance of the heterogeneous version on the worst scenario.

While the communication overhead is directly dependent on the number of cores, the overall performance of the configurations varies accordingly to the characteristics of the application being executed. We can clearly see that the application that have high levels of TLP – previously discussed in section 7.2 – are faster on the versions of CReAMS with more cores, the homogeneous in this case. On the other hand, applications that are more heterogeneous (have low load balancing) or that have high ILP levels are benefited by the assorted environment provided by the heterogeneous CReAMS. On this case, where no scheduler is assigned, these applications can take advantage of the bigger cores that are available on heterogeneous configurations and, as they have four times less cores, they do not suffer with the communication overhead as the homogeneous versions do.

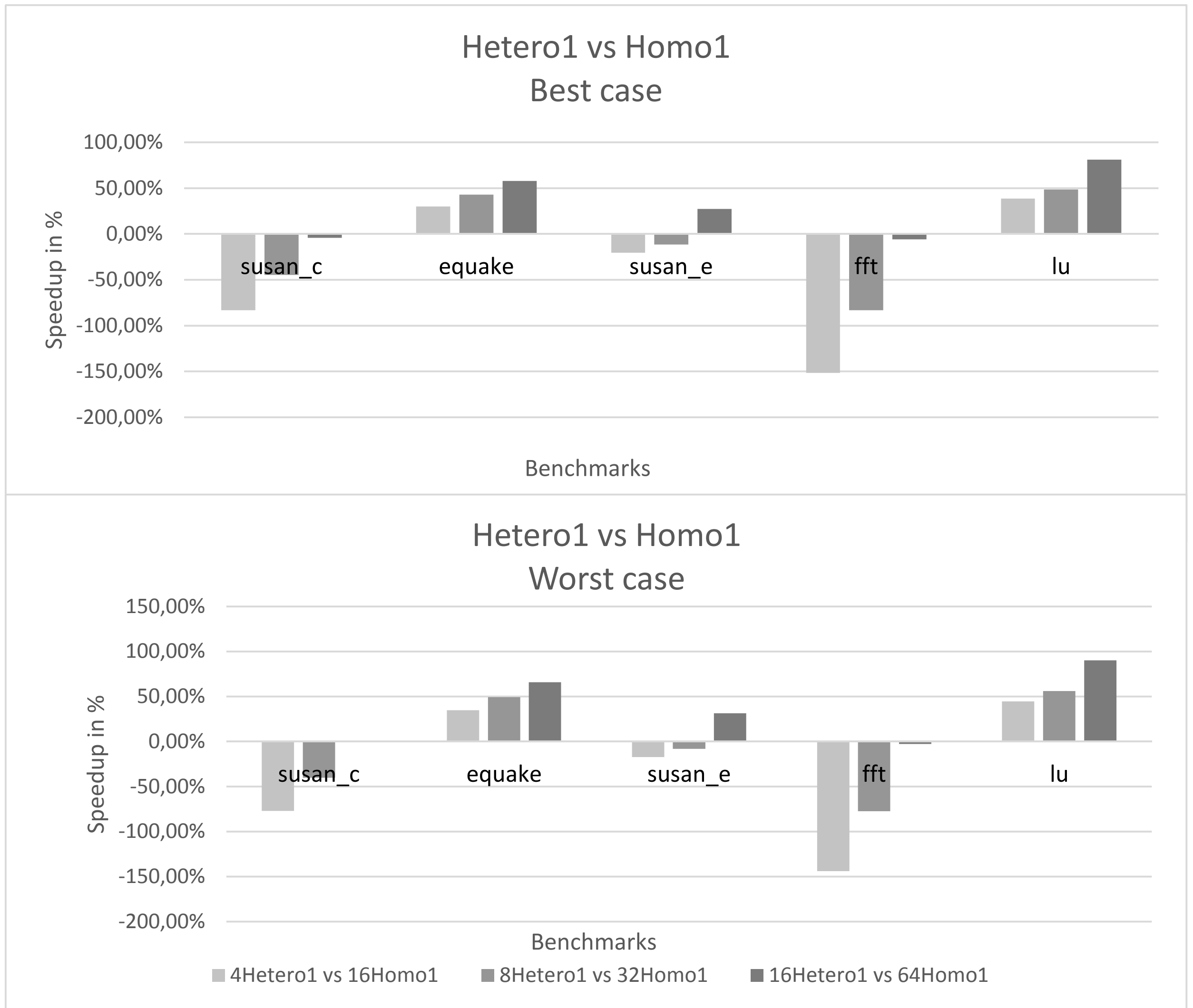


Figure 9 Hetero1 vs Homo1: Best and worst cases

Figure 10 shows the results comparing the Hetero1 with the Homo2 configurations. On this scenario, the effects of the worst case in communication are more evident. This scenario shows the same behavior we have observed on Figure 9 for the applications. However, as the Homo2 configurations are bigger than the Homo1, we see worst results on the high TLP benchmarks and better on the high ILP ones. Furthermore, in both figures, we can observe a tendency of getting better results when the number of cores is increased. In some applications that have high TLP levels, we switch from a scenario where the heterogeneous was slower to one that it becomes faster. For instance, in the worst case of Figure 10, both *susan_c*, *susan_e* and *fft* switch from losing in 4Hetero1 to winning in 32Hetero1. This suggests that the TLP gains provided by these applications reach a threshold point where they are smaller than the communication overhead added by the extra number of cores.

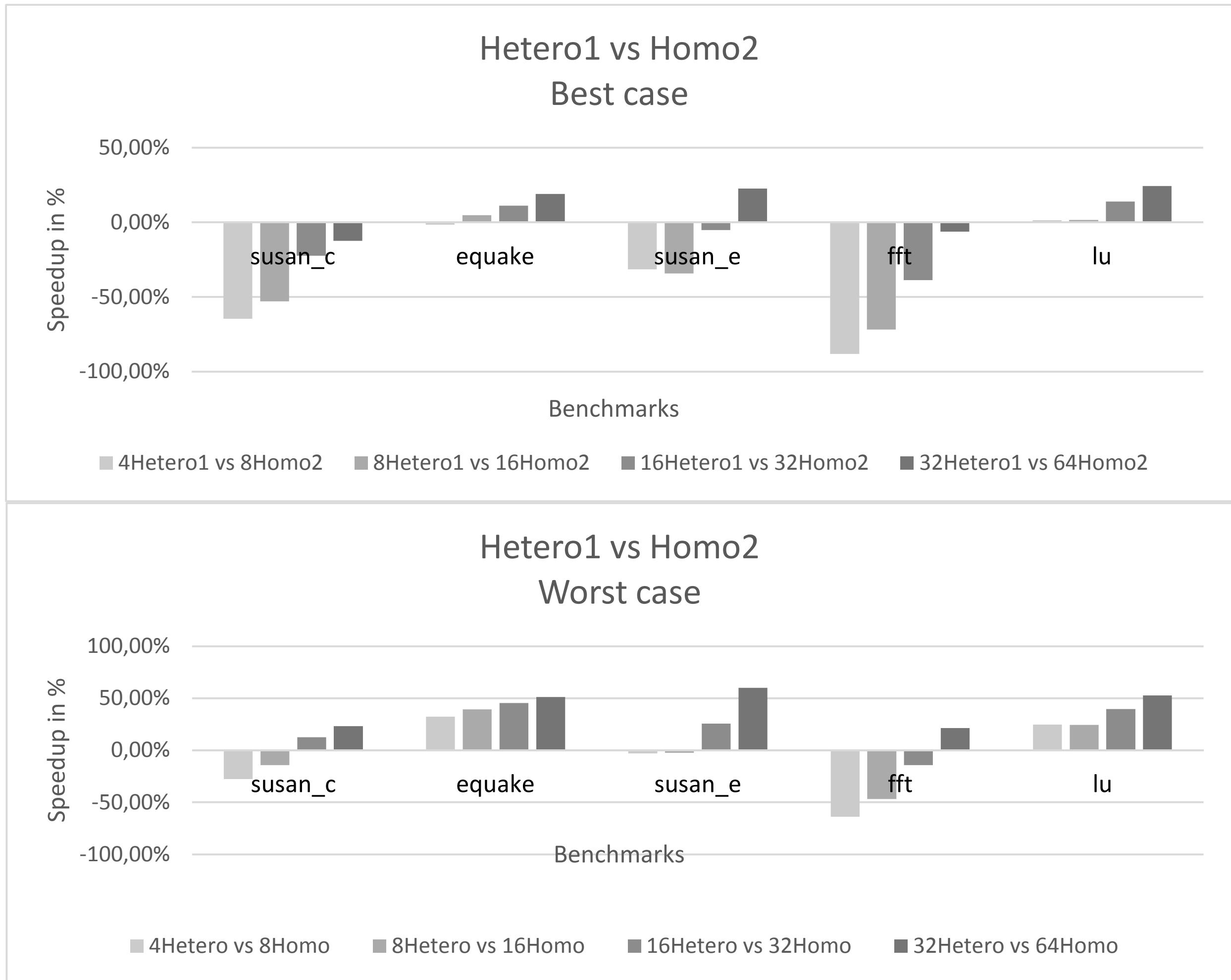


Figure 10 Hetero1 vs Homo2: Best and worst cases

7.2.2 Simulations with scheduler

On the second phase of simulations, we have added a simple scheduler to the CReAMS system in order to select the proper core for each thread based on its levels of ILP. We have also taken care to create new configurations that would represent three area scenarios: one in which the heterogeneous version would be smaller than the homogeneous, one in which they would have the same area and one in which the heterogeneous would be bigger than the homogeneous. However, area parity is still kept by the multiplication of the cores.

As in the section 8.2.1, we will only discuss the most significant results. All the comparisons can be found on the APPENDIX B.

7.2.2.1 Heterogeneous configuration smaller than the homogeneous

Figure 11 shows the Hetero10 configuration compared with the Homo5. Accordingly to Table 6, the Homo5 configuration is composed of big arrays (if compared with the other homogeneous versions), while the Hetero10 configuration has 25% of large cores, 50% of medium and 25% of small, as shown in Table 9. This produces a heterogeneous version of CReAMS that is smaller than the homogeneous.

The results show that the heterogeneous version is superior on the applications *susan_c*, *susan_e* and *lu*. *Susan_c* has good load balance even when the number of cores highly increases, and for up to eight cores the TLP exploitation benefits the heterogeneous version that has more cores. However, from sixteen cores the communication overhead starts to harm the heterogeneous performance. Load balancing in *lu* decreases in a higher rate from eight cores. This, combined with the overhead of the extra cores, produces the decline in performance observed in the Figure 11. *Equake* has very low load balancing, so it is natural to observe better performance on the homogeneous version, that has less cores (consequently, less overhead on communication) and bigger arrays to exploit ILP.

FFT shows high declines in performance when the number of cores is increased. And, as we will discuss later, this behavior is observed in all the simulations that were run with the scheduler. However, this was not the behavior that the application has shown in the Figure 9 and Figure 10 (when the simulation was run without scheduler), which suggests that the scheduler algorithm actually harms the performance on the *FFT*.

This performance loss might be explained due to changes in load of a thread during execution. If a thread is allocated to a small core, but it has big load (it could use much more resources if the array could provide), it will be reallocated on a bigger core. However, this allocation only occurs on synchronization points of the threads and all the other cores will be stalled while waiting for this particular thread to reach this point. If a thread is then allocated to a more suitable core, but then changes its load, it will need to be, again, reallocated. The process of changing context between cores might be to sparse and that might be causing such losses.

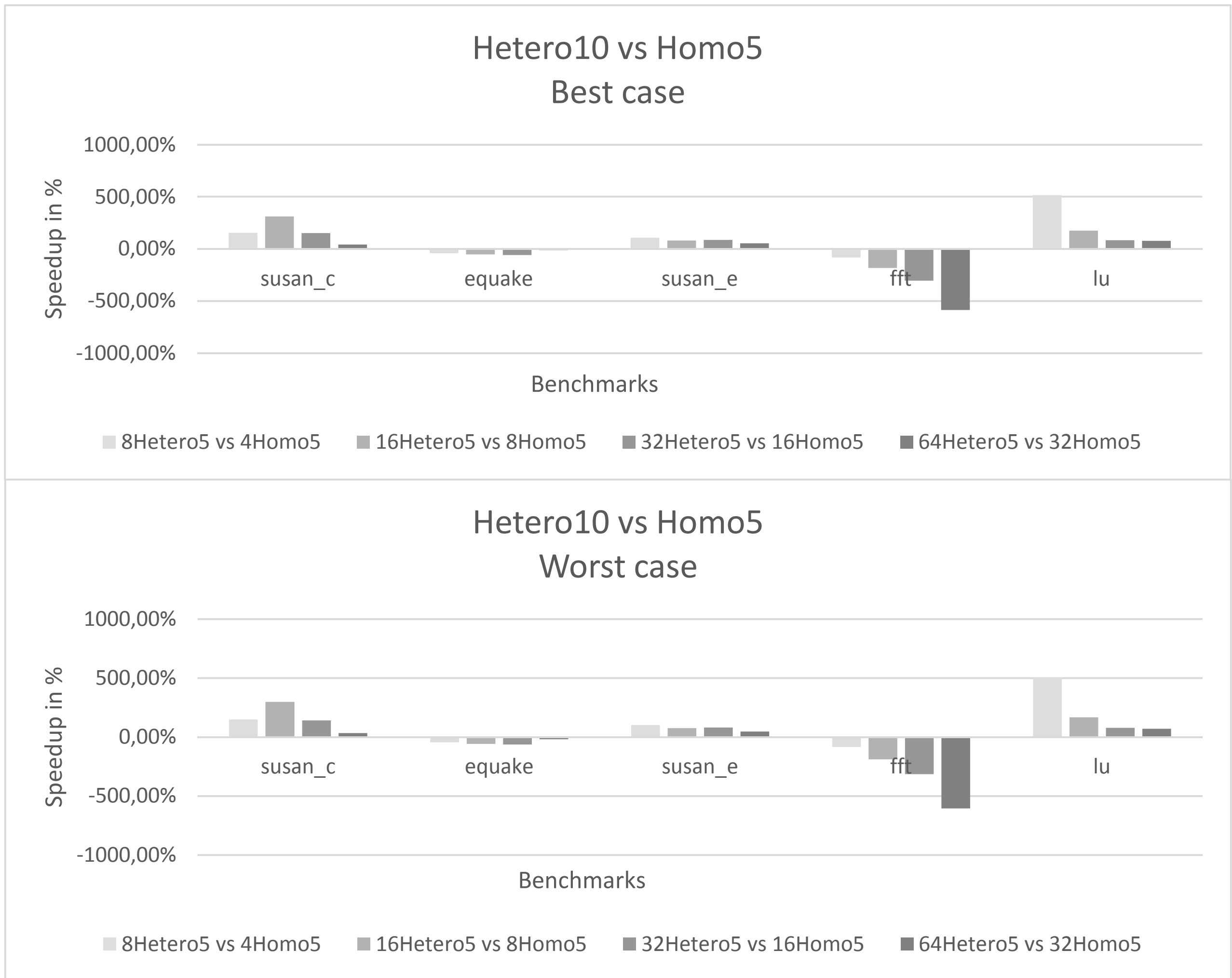


Figure 11 Hetero10 vs Homo5: Heterogenous version is smaller than the homogeneous

7.2.2.2 Heterogeneous and homogeneous configurations with same area

In the Figure 12 we present the results for the comparation of configurations Hetero8 and Homo5. Although Homo5 is considered to have large cores, we can observe in the Table 9 that the medium cores of Hetero8 are actualy the same size of the Homo5 ones and the large cores of Hetero8 are twice the size as the medium. However, this heterogeneous configuration has mostly small cores (50% of them), thus we achieve a version that has the same area as the homogeneous.

These results should provide an insight on the performance given exclusively by the diverse environment of the heterogeneous DAPs combined with the scheduler algorithm, as the number of cores are the same and, thus, the communication overhead on both configurations are equivalent.

As we can see, *susan_e, susan_c* and *lu* that have bigger basic blocks take advantage of the bigger cores of the heterogeneous version. In addition, the scheduler is work on their favor by allocating the haviest threads on the cores that have more resources.

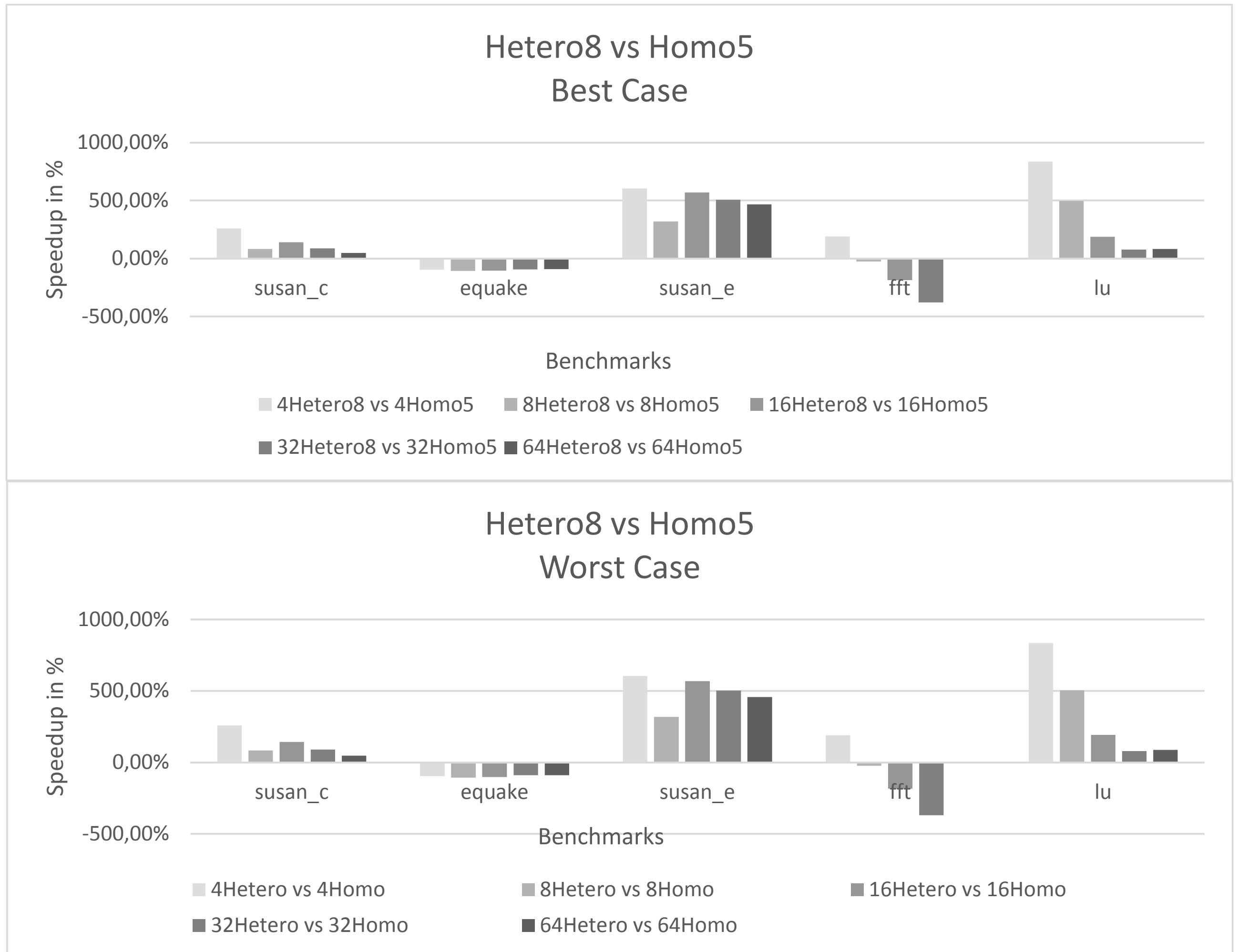


Figure 12 Hetero8 vs Homo5: Both configurations have the same area

7.2.2.3 Heterogeneous configuration bigger than the homogeneous

Figure 13 shows the scenario where the Hetero8 version is bigger than the Homo4, so we have to compare the homogeneous version with more cores than the heterogeneous to reach area parity. This situation is similar to the one on the first phase of the work, when we have simulated the applications without scheduler, and is good to reach conclusions about the scheduler performance.

As can be seen, the applications that are more heterogeneous (with low load balancing or high ILP) like *susan_e* and *lu* shows better performance when executed on the heterogeneous CReAMS. In addition, the homogeneous version is harmed by the communication overhead as

it has more cores, which increases the performance gains in those applications. *Susan_c*, however, has advantages of the exploitation of TLP, so the homogeneous version is more suitable for this benchmark.

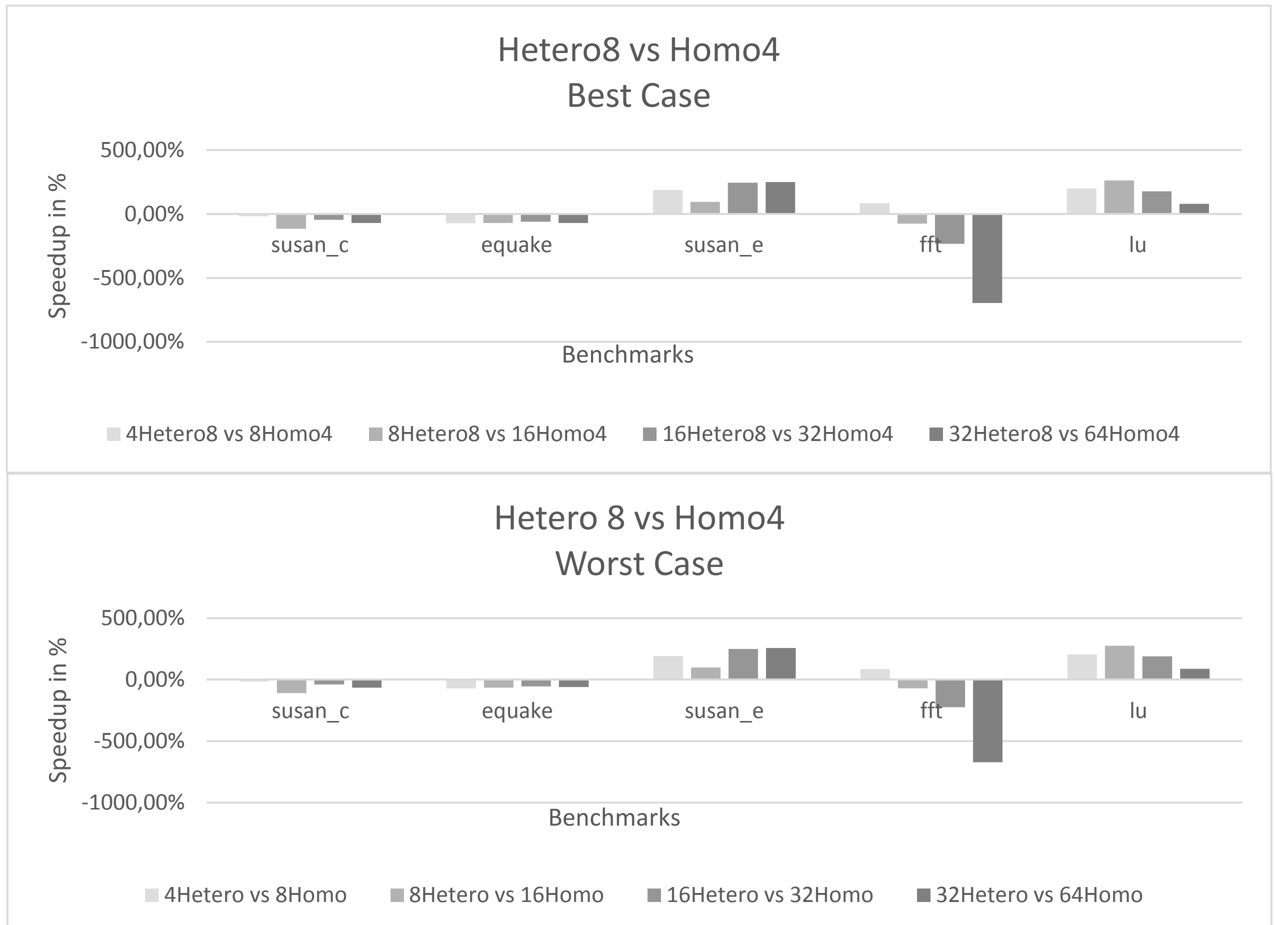


Figure 13 Hetero8 vs Homo4: Heterogenous version is bigger than the homogeneous

We can also make additional observations over the results, regarding all the studied scenarios. For instance, on any of the simulations done with the scheduler we were able to see the same behavior that we have observed on the simulations without scheduler. The steady increase on performance of the heterogeneous version when the number of cores is also increased and the heterogeneous has less cores than the homogeneous is theoretically expected. This is because the TLP exploitation has a limit for all applications (even the ones with good load balancing) and the communication overhead is a significant issue for performance. The results suggests that the scheduler not only harms the *FFT* benchmark, but all other applications. This is especially observed when the number of cores increases and there is no common behavior between the applications and the compared configurations as we have seen in the Figure 9 and Figure 10, which shows that the scheduler does not escalate well.

8. CONCLUSION AND FUTURE WORK

This work shows how simple are the steps to create heterogeneous reconfigurable processors using the CReAMS organization. The system array of resources is easily customizable and – as the ISA is always the same, independently of the array – no special hardware is necessary to mix different DAPs inside a processor. However, this simplicity does not mean that finding the best combination of resources is a trivial process. Results shows that small variations of the array resources can change the performance of applications appreciably.

The simulations also demonstrate that the heterogeneous CReAMS can reach better performance than the homogeneous when the application has varied load balance. Our best results suggests that having less cores, but with more resources, are especially more efficient for such applications.

Although a dynamic scheduler is essential for heterogeneous systems to reach their full efficiency, this work also shows that if this scheduler is not well designed, it can actually be harmful to the performance of applications. On our results with scheduling of threads, the benchmarks have shown mostly expected results, but without a predictable behavior.

Thus, as future work for this project, we propose a better version of the scheduling algorithm. We will track the points of performance overhead introduced by the current version and improve them in order to reach results that are more consistent for the heterogeneous CReAMS.

Another important metric for any embedded system is the power consumption. Todays embedded system are usually mobile, or have very strict power constrains. CReAMS, as a homogeneous system, already introduced power analysis, so we propose to add this metric on the heterogeneous version as well. In some scenarios, the ideal configuration may not be the one with best performance, but the one with best energy efficiency, or even a combination of both.

REFERENCES

[1] M. Rutzig, *A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Efficient ILP and TLP Exploitation,* Porto Alegre, Doctor Thesis of Computer Science, 2012.

[2] A. C. S. Beck, L. Carro and C. A. Lisboa, Adaptable Embedded Systems, New York: Springer, 2013.

[3] A. C. S. Beck and L. Carro, Dynamic Reconfigurable Architectures and Transparent Optimization Techniques, Springer-Verlag, 2010.

[4] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev and L. Carro, "Run-Time Adaptable Architectures for Heterogeneous Behavior Embedded Systems," *In Proceedings of the 4th international workshop on Reconfigurable Computing: Architectures, Tools and Applications,* 2008.

[5] M. A. Watkins and D. H. Albonesi, "Enabling Parallelization via a Reconfigurable Chip Multiprocessor," *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures,* 2010.

[6] J. Lee, H. Wu, M. Ravichandran and N. Clark, "Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications.," *ISCA,* pp. 270-279, 2010.

[7] L. Roman, S. Greg and V. Frank, "Warp Processors.," in *41st annual Design Automation Conference*, New York, 2004.

[8] "big.LITTLE Technology," ARM, [Online]. Available: http://www.arm.com/products/processors/technologies/bi. [Accessed 17 11 2014].

[9] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker and J. Henkel, "KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-SetMulti-grained-Array Architecture," *Design, Automation & Test in Europe Conference,* pp. 819-824, 2010.

[10] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan and D. Tullsen, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 81 - 92, 2003.

[11] D. Koufaty, D. Reddy and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," *Proceedings of the 5th European conference on Computer systems,* pp. 125-138, 2010.

[12] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez and J. Emer, "Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE)," *Proceedings of the 39th Annual International Symposium on Computer Architecture,* pp. 213-224, 2012.

[13] S. Srinivasan, R. Rodrigues, A. Annamalai, I. Koren and S. Kundu, "A study on polymorphing superscalar processor dynamically to improve power efficiency," *IEEE Computer Society Annual Symposium on VLSI,* pp. 46 - 51, 2013.

[14] M. B. Rutzig, A. C. S. Beck and L. Carro, "A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Simultaneous ILP and TLP Exploitation.," *Design, Automation & Test in Europe Conference & Exhibition (DATE),* pp. 1559-1564, 2013.

[15] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," *In Proceedings of the conference on Design, automation and test in Europe (DATE'08),* pp. 1208-1213, 2008.

[16] M. P. S., C. M., E. J., H. G. Forsgren D., H. J., L. F., M. A. and W. B., "Simics: A full system simulation platform," 2002.

[17] B. C., K. S., S. J.P. and L. K., "The PARSEC benchmark suite: Characterization and architectural implications," 2008.

[18] A. Dorta, C. Rodriguez, S. F. and G.-E. A., "The OpenMP source code repository," *Proceedings of the 13th Euromicro conference on Parallel, Distributed and Network-Based Processing.*

[19] K. M. Dixit, "The SPEC benchmarks," *Computer benchmarks,* pp. 143-163, 1993.

[20] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization,* 2001.

[21] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," *Proceedings of the 3rd conference on Computing frontiers,* pp. 29-40, 2006.

[22] "CACTI 5.3," Hewlett-Packard, [Online]. Available: http://quid.hpl.hp.com:9081/cacti/. [Accessed 11 11 2014].

[23] A. C. S. Beck, C. A. Lisboa and L. Carro, Adaptable Embedded Systems, Springer-Verlag, 2012.

APPENDIX A

Heterogeneous CReAMS: A study on performance gain from a heterogeneous multicore system over a homogeneous system

Jeckson Dellagostin Souza

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

jeckson.dellagostin@gmail.com

Abstract. *Reconfigurable architectures are an alternative for classic superscalar organizations to the exploration of instruction level parallelism, while the multicore organizations are the most communly used strategy to exploit thread level parallelism. This work is based on CReAMS, a multicore reconfigurable system that explores both TLP and ILP. Moreover, we present the concept of the heterogeneous CReAMS, which cores have distinct resources. We will be introducing the basics of this organization and results of simulations made to explore it. Also, we will show the motivations for using a heterogeneous system over a homogeneous one. This work proposes to find a heterogeneous configuration of CReAMS that out performs a homogeneous configuration of same area. The first simulations were executed on homogeneous cores and they illustrate the potential of the system on exploring both TLP and ILP.*

1. INTRODUCTION

For many years, the majority of embedded systems were designed to execute specific and specialized tasks. However, with the advancement of both transistor technologies and integrated circuit design,, embedded processors are now able to perform many kinds of operations. Moreover, users now demand to be able to execute their daily routine tasks using as few devices as possible. Thus, each new generation of embedded hardware is expected to do even more.

On program execution, it is usual for some instructions to be independent of each other, so they can be executed at the same time, but on different functional units inside the processor. This is known as exploiting parallelism on instruction level and is commonly explored by superscalar processors. On the other hand, modern operating systems can distribute threads through many processors, so they can be executed simultaneously. This is known as exploiting the parallelism at thread level and is largely explored by multicore organizations.

Until the beginning of the 2000's, the performance gains were all concentrated on ILP exploitation and this led to die size problems, as the strategies used greatly raised the complexity of circuitry. The tradeoff between area and performance was not positive and became an issue. Thus, it was clear that a new strategy to create faster processors was necessary, and designers started to explore the TLP by replicating the cores.

The superscalar strategy is the most common approach to explore ILP and it is used on both general purpose – as the Intel x86 architectures – and embedded processors – like the ARM architecture. However, this is expensive, as for each word of instructions read, the superscalar processor has to evaluate the data dependencies for exploiting the parallelism. Hence, another possible strategy to explore ILP are the reconfigurable architectures.

This work suggests the use of Custom Reconfigurable Arrays for Multiprocessor Systems (CReAMS), which is a proposed organization capable of exploiting both Instruction Level Parallelism (ILP) – using a dynamic adaptable array of processing units on each core – and Thread Level Parallelism (TLP) – through the use of many processing cores. This organization has already shown to have better performance if compared to a standalone processor when all the cores are homogeneous – have the same size and resources.

A study to compare the effects of ILP and TLP in CReAMS over a base processor will be made. We will be using four different benchmarks – chosen to cover a wide range of applications – to measure their performance. CReAMS is currently available as a homogeneous systems. Therefore, this work proposes to find a set of heterogeneous organization CReAMS – containing cores of different processing capabilities – on which their performance is superior to a homogeneous configuration of same area. The main goal is to exploit the diversity in

processing to achieve this better performance. We will be simulating a variety of different combinations of reconfigurable arrays sizes and comparing them to an equivalent area homogeneous CReAMS.

It is expected for a well explored heterogeneous system to be more efficient than a homogeneous because of the smarter use of chip area and energy resources. However, a heterogeneous configuration must have a good thread scheduling and distribution over the different cores to fully exploit this efficiency.

2. RECONFIGURABLE SYSTEMS

The architectures that can adapt themselves to provide a hardware expertize for a specific application are known as reconfigurable systems. Because of this specialization, these architectures are expected to provide performance and energy saving improvements. However, these systems are still built aiming flexibility to execute many kinds of tasks, which means that they have smaller gains if compared to dedicated circuits, like Application-Specific Instruction Set Processors (ASIPs) and Application-Specific Integrated Circuits (ASICs) [1].

One can find different proposed works on reconfigurable systems on the literature, like the ReMAPP system [9] which is composed of a pair of coarse grained (it implements word-level operations on the functional units) reconfigurable arrays that is shared among several cores. There is also the KAHRISMA [10] which supports multiple instructions sets (it is a heterogeneous architecture) and has fine-(it implements bit-level operations) and coarse-grained reconfigurable arrays. This system is, however, heavily dependent on software compilation.

This work will be based on CReAMS, which is a reconfigurable multiprocessing system created by our group of research and still has room for improvements. Furthermore, we have access to all the source code and tools for this system, making it suitable to adapt its configuration to our needs. Moreover, the DAP (discussed later) has an easily modifiable binary translator for different architectures (instruction sets), which is independent to the variation of the functional units on the reconfigurable array.

2.1. DAP - Dynamic Adaptive Processors

The DAP is a reconfigurable architecture coupled to the processor and was presented in [2]. It is a transparent coarse grained architecture – the processor is not aware of the array and, also, the latter's functional units implement word-level operations. The DAP is used by the CReAMS – in fact, it is the basic core of CReAMS –, thus we will give more details about this architecture. To better explain the DAP, we divided it in four blocks, as shown in Figure 1(a).

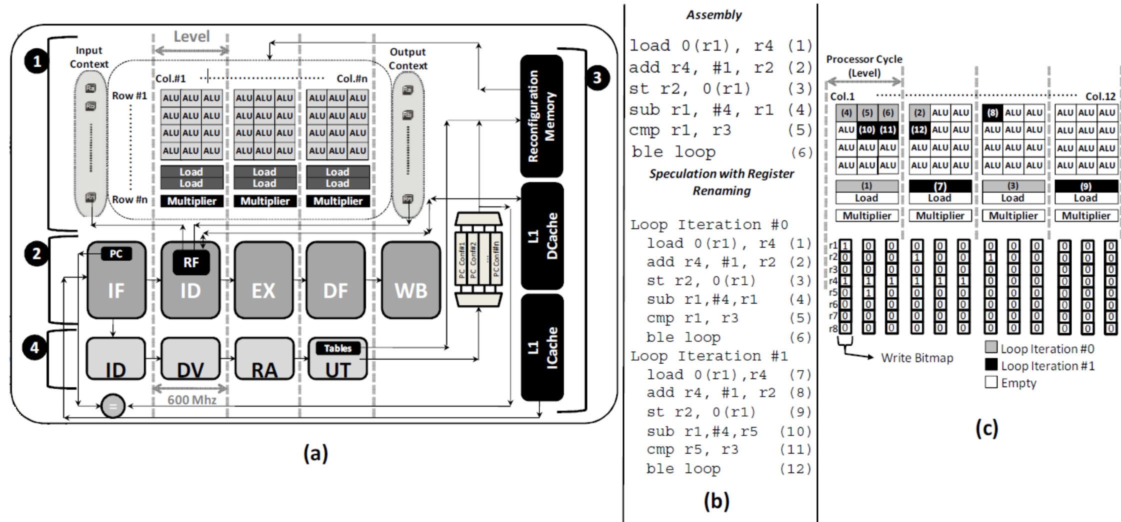


Figure 1. (a) DAP blocks (b) Assembly of a loop (c) Allocation inside of reconfigurable data path

2.1.1 Block 1 – Reconfigurable data path

Block 1 shows the structure for the reconfigurable data path. It is coarse-grained and tightly coupled to the processor's pipeline (there is no shared bus between them), which avoids external access to the memory and, consequently, saves power and reduces the reconfiguration time. As we can see in Figure 1(a), the data path is organized in a matrix structure, where the number of rows is the maximum number of instructions that can be executed in parallel – independent instructions can be allocated on the same column, in other words, the number of columns limits the ILP exploitation –, while the number of columns dictates the maximum number of dependent instructions that can be executed in sequence in a configuration – the columns in one level are executed in sequence as a big combinational block. For example, the configuration on Figure 1 is able to perform four arithmetic and logic operation, two memory accesses on cache and one multiplication at the same time if all the data instructions are independent. As the critical path (the piece of combinational circuit that takes longer to produce a correct result) is the multiplier, it is possible to have other faster units on the same level. On the example, three arithmetic and logic units (ALUs) compose a level, while the multiplier and the memory access take the equivalent to one cycle on the processor. In other words, this configuration can execute twelve arithmetic and logic operations, two memory accesses and one multiplication on each level at the very best case of data dependency.

The entire structure of the reconfigurable data path is combinational, meaning that no temporal barrier (registers) are added between the functional units. The only registers are present on the entry point – the input context – and the exit point – the storage of the results. The feeding of the input context with the necessary data is the first step to configure the data path before starting the execution. The results are sent to the processor's register file on demand. It means that if any value is produced at any data path level (a cycle of the processor) and if it will not be changed in the next levels, this value can be written back on the next cycle. If the number of writes produced be the array is greater than the number of available write ports in the register file, than the excess instructions are forwarded to the next level – on the example shown in Figure 1(a), the maximum number of ports available is two.

Figure 2 shows a simplified overview of the interconnection structure of the reconfigurable data path. Bus lines connect the input context with the functional units and the output context, while multiplexers are responsible for choosing the path this data will run. The input multiplexers – two for each functional unit – will dictate if a value will be used by a determined

functional unit (on the Figure 2, a ALU), while the output multiplexers – one for each output – select if the output (next column) will come directly from an input or from a functional unit.

The input and output sizes are important parameters of the systems as they may limit the number of instructions allocated in a single data path configuration. If the input context is full, a new configuration will be created, so a small input context size might reduce performance, as a new configuration will start even if there are still functional units available. However, a large input context could lead to a huge overhead in the data path, as the size of the input multiplexers and the number of output multiplexers are directly dependent of the input context length.

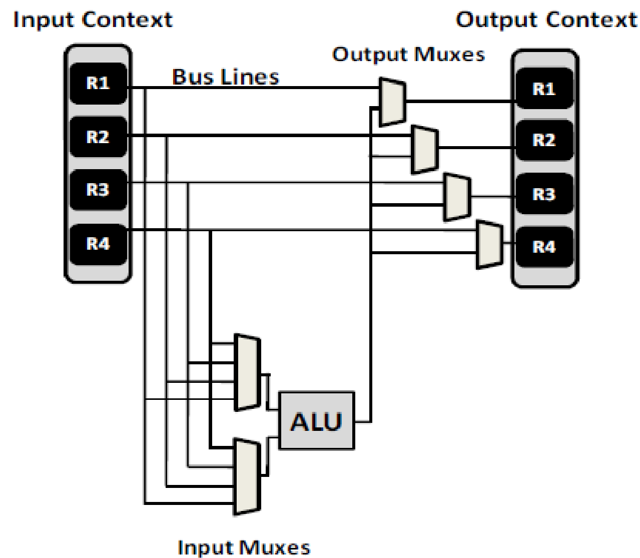


Figure 2. Interconnection mechanism

2.1.2 Block 2 – Processor pipeline

Block 2 is the basic processor to be used coupled with the array. It is also the reference for comparisons on the simulation performance. In this work, the baseline processor is a SparcV8-based architecture. Its five stage pipeline reflects a traditional RISC design (instruction fetch, decode, execution, data fetch and write back) and is similar to other RISC processors used in well-known embedded platforms (e.g. MIPS, ARM).

2.1.3 Block 3 – Storage components

There are two memory units specialized for the reconfiguration system: the address cache and the reconfiguration memory. The first holds the memory address of the first instruction of every configuration built by the dynamic detection hardware (explained later). This cache is implemented as a 4-way set associative table containing 64 entries (which means that the system can hold up to 64 configurations). An address cache hit indicates that a configuration was found, therefore this cache is used the verify the existence of a configuration on the reconfiguration memory – where the configuration bits are kept.

2.1.4 Block 4 – Dynamic detection hardware

This block is responsible for instruction detection and allocation in the data path and is implemented as a 4-stage pipelined circuit. The Dynamic Detection Hardware (DDH) does not increase the critical path of the processor and it is a binary translation mechanism that translates

the instructions from SparcV8 ISA to data path configurations. The stages of the circuit are divided in Instruction Decode (ID), Dependence Verification (DP), Resource Allocation (RA) and Update Tables (UT). The translation process is performed as the processor executes the instruction (at the same time and independently), so there is no extra performance overhead.

For each column on the reconfiguration data path (Figure 1(a)), there is a bitmap responsible for storing in the target operands of the already allocated instructions in the respective column, named as Write Bitmap (Figure 1(c)). Thus, for each incoming instruction, its source operands will be compared to the target operands in this bitmap to decide in which column this instruction will be allocated, according to data dependencies.

Figure 1(b) has an assembly code as an example. On Figure 1(c), the allocation of this code on the reconfigurable data path is shown. The first incoming instruction, a memory access, is allocated on the highest functional unit of the leftmost data path column. However, as on this process this type of operation takes an entire level (a processor cycle), the fourth bit of the write bitmap (representing the r4 register) of the columns 1, 2 and 3 are set to maintain the allocation consistency.

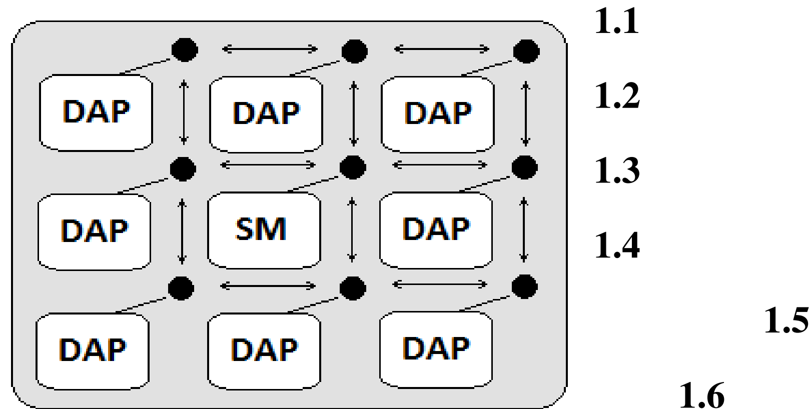
The dependency detection starts from the second instruction. In the example, the instruction number two reads the r4 register. As it is written by the previous instruction, a read after write (RAW) dependence is found. The DDH detects it (through the write bitmap) and allocates the instruction number two at the later column of instruction number one. The second bit of the fourth column of the write bitmap is set since this instruction has the register r2 as the target operand. The dependency analysis keeps these steps until instruction number five, where a loop is found.

The DDH supports speculation, so when the branch instruction is found, a speculation flag is set and the configuration continues the allocation of the following iterations. In other words, it is possible (if there is enough space on the array) to keep multiple iterations of a basic block on the same configuration. The instructions in black on Figure 1(c) represents the instructions allocated for the second iteration of the loop code of Figure 1(b).

This hardware is capable of performing register renaming – for false dependency treatment – as well. In instruction number ten, the register r1 could be read by the incoming instruction in the second column, but could not write in this same register at this column. This is detected by the DDH and the register is renamed to r5 (the next empty register of the input context). All subsequent instructions that contain a reference to r1 are modified accordingly.

3. CREAMS

The Custom Reconfigurable Arrays for Multiprocessor Systems (CReAMS) is an architecture based on the DAP and was presented in [3]. It is, actually, an extension of the later created to exploit TLP through the replication of the number of DAPs in a system. In this way, the reconfigurable system can now support (and take advantage of) multithreaded applications. The communication among the DAPs is done through a 2D-mesh Network on Chip (NoC) using a XY routing strategy. CReAMS also includes an on-chip unified L2 shared memory, illustrated as SM in the Figure 3.



1.1
1.2
1.3
1.4
1.5
1.6
1.7

Figure 3. CReAMS of 8 cores (DAPs) and a L2 shared memory (SM)

3.1. Homogeneous CReAMS

The homogeneous version of CReAMS is a configuration where all the DAPs are exactly the same. They all have the same size in area, the same memory size (L1 cache size, reconfiguration memory table size, number of input and output context...) and the same functional units. This configuration represents the traditional approach to multicore systems – where the DAPs would be generic cores – and it is simple to implement, as no special scheduling is necessary (a thread is simply allocated to the next free DAP). However, this is not the most efficient approach, whereas a low duty thread will execute on the same environment as a heavy duty one.

3.2. Heterogeneous CReAMS

A heterogeneous version of CReAMS is a configuration where each DAP has a different number of functional units, input and output context length and memory size. This allow for some cores to be bigger than others, or in other words, more efficient to execute threads that can explore higher levels of instruction parallelism. Similarly, smaller DAPs would be allocated to run jobs with low ILP.

Scheduling threads accordingly to their necessities leads to a greater energy efficiency. If a DAP is too big and is scheduled to execute a thread with low ILP, many functional units would be wasted for not being used. On the other hand, if a DAP is too small and is allocated for a thread with high ILP, the system would need many more cycles to execute the thread as no sufficient functional units would be available – wasting time and energy.

It is also expected for a heterogeneous system to have a smaller communication overhead. Considering the same chip area a CReAMS composed of only small DAPs could have much more cores than a CReAMS with small, medium and big sized DAP. Therefore, if the heterogeneous configuration performs better than the homogeneous, it will do so using fewer cores with less communication between them.

This configuration, however, leads to some problems that need to be addressed carefully. For instance, how to determine the best sizes of DAPs to be used and the best combination – should we have many small sized DAPs or more big sized ones – that leads to better performance if compared to the homogeneous configuration.

4. BENCHMARK AND SIMULATION ENVIRONMENT

This work will be using the Simics simulator[7] to generate the instruction trace from a series of applications. This instructions will then be split according to their threads and each of

these threads will be allocated to a DAP simulator. A series of tests were run to configure and validate the tools for the simulation environment.

Moreover, in order to measure the performance of CReAMS, 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 four benchmarks chosen were *jacobi* and *lu* from parallel suites [4][11][12] and *susan edges* and *susan smoothing* from the MiBench suite [5]. In [6] a study was made to characterize the potential of these applications on obtaining performance improvement when TLP or ILP exploration is applied. The mean basic block size gives us some clues about the limits of ILP that the selected application provide. In addition, the percentage of the entire application code that is executed in parallel, when multithreaded application environment is considered, is an important metric to obtain the actual TLP available in the applications. This metric is also called load balancing. Table 1 shows the results of this study. *jacobi* is an application with almost perfect load balancing, which means it is great for TLP exploitation. However, its small basic blocks should leave few space for ILP improvements. *susan_e*, on the other hand, has big basic blocks and little load balancing, meaning great ILP, but poor TLP and *susan_s* should have good results coming from both TLP and ILP as it has a good load balancing and big basic blocks. *lu* has an exotic behavior, as it has a medium size mean basic block and good load balancing on few threads, but poor load balancing on many.

Wrapping this information:

- *jacobi*: wide TLP, but small ILP.
- *susan_e*: small TLP, but wide ILP.
- *susan_s*: wide TLP and ILP.
- *lu*: wide TLP for up to eight threads, than small TLP. Medium ILP.

Table 1. Load balancing and mean basic block size of selected applications

Benchmark	Mean BB Size(#instr)	Load Balancing (%) in threads			
		4	8	16	64
jacobi	6,94	97,02	97,02	92,07	93,12
susan_e	16,60	39,80	24,90	4,80	0,90
susan_s	12,10	88,20	77,13	83,16	74,52
lu	8,32	82,20	56,77	29,35	7,03

5. HOMOGENEOUS SIMULATIONS

Our first simulations were based on homogeneous configurations of CReAMS. Our biggest motivations here were to try to push the systems to its limits on both ILP and TLP exploitation. In order to reach that limit, we have simulated CReAMS on different reconfiguration data path sizes, from 1 to 128 cores. These simulations are not taking in account the overhead caused by communication between the cores, this will be added on future simulations of this work. The three different configurations used are described in table 2.

Table 2: The three different configurations used.

Configuration (Number)	Lines	Columns	ULA	Memory access	Multiplication	Cache (KB)	Input context
Small (1)	9	9	3	1	2	32	8
Big (2)	15	15	4	4	2	128	32
Huge (3)	200	200	10	10	20	128	32

The small and big configurations can be used to compare the performance of CReAMS in real world situations, as their size represents approximately the area of 2 and 4 SparcV8 processors, respectively. The huge configuration, however, has the area of, about, 200 SparcV8 processors, thus it is not legible for comparison and was used only to stretch CReAMS to its limits of ILP exploitation. Figure 4 shows the results for the simulations of *susan smoothing, jacobi, susan edges* and *lu.* To better observe the small speedups on few cores of *jacobi* and *susan smoothing* (which have linear grow due to their high load balance), their charts are shown in $\log_{10}$ scale. We have also included two constant lines which illustrate the monocore speedup over the base processor. These lines represents the "ILP-only" gains and show us the points where the added TLP exploitation overcomes the "ILP-only". The motivation of these lines is to demonstrate that even if the ILP exploitation is high, it is still valuable to explore the TLP.

In Figure 4(a) and (b), we can see that *jacobi* and *susan soothing* are greatly enhanced by the number of cores. Configuration 2 single core has slower performance even when compared with a two core standalone SparcV8 (which has half the area). We can also see that even the huge (3) single core configuration does not have a much better speedup than a dual core processor. However, every many core configuration that use CReAMS have better performance than the standalone SparcV8, showing that both these applications can still provide gains by exploiting ILP. On the other hand, if area parity is added, we have to consider that every configuration 1 has twice the size of a SparcV8 processor and every configuration 2 has 4 times more area. Thus, on this two applications where TLP is highly appreciated, the SparcV8 out performs the CReAMS configurations in about 1.5x for configuration 1 and 2.5x for configuration 2.

Figure 4(c) shows that *susan edges'* speedup is not so affected by TLP, but is greatly improved by ILP. The configuration 3 with just one core is only out performed by configuration 2 when the latter has eight cores and configuration 1 has better performance just at sixteen cores. The huge configuration (3) out performs the standalone SparcV8 processor at all simulated scenarios, even with 128 cores. Though 3 is not a feasible configuration, it demonstrates the potential of ILP exploitation on high data flow applications – which have big basic blocks – using CReAMS. When comparing configuration 2 with configuration 1, we have to keep in mind that the first has about as twice the size as the latter. Therefore, we need to compare the speedups of the big configuration with half the number of cores of the small one. The small configuration has better performance than the big one starting from two to eight cores. When 1 has sixteen cores and 2 has eight cores, the roles are switched and the big configuration out performs the small one. This happens because at few number of cores, the TLP still has a good influence over the performance gains, so as 1 has more cores, it is faster. However, at sixteen cores the speedup curve slope starts to decrease and ILP becomes the major factor for the increase of performance. When compared with the standalone SparcV8, configuration 1 starts performs better at 8 cores when it is 1.16x faster than the 16 cores

SparcV8. Configuration 2 performs better at 4 cores, when it is 1.09x faster than the 16 cores SparcV8.

lu has an interesting behavior, as up to eight cores the TLP increases the overall performance of the configurations. However, from this point on, the number of cores starts to decrease the speedup to the point that on 64 cores the performance is almost the same (actually smaller in some configurations) as it is with only one core. The further increase in numbers of cores is so harmful to the system that we could not even test an environment with 128 cores, because the simulator ran out of resources. The results are as expected from Table 1, where we can see the load balancing of the applications. Up to eight cores, *lu* has good to medium TLP exploitation, but after that it widely decreases because the load balancing between the threads is greatly reduced. This performance loss is compensated, nonetheless, by the ILP and that is why configurations 1, 2 and 3 still have better performance than the standalone SparcV8 processor.

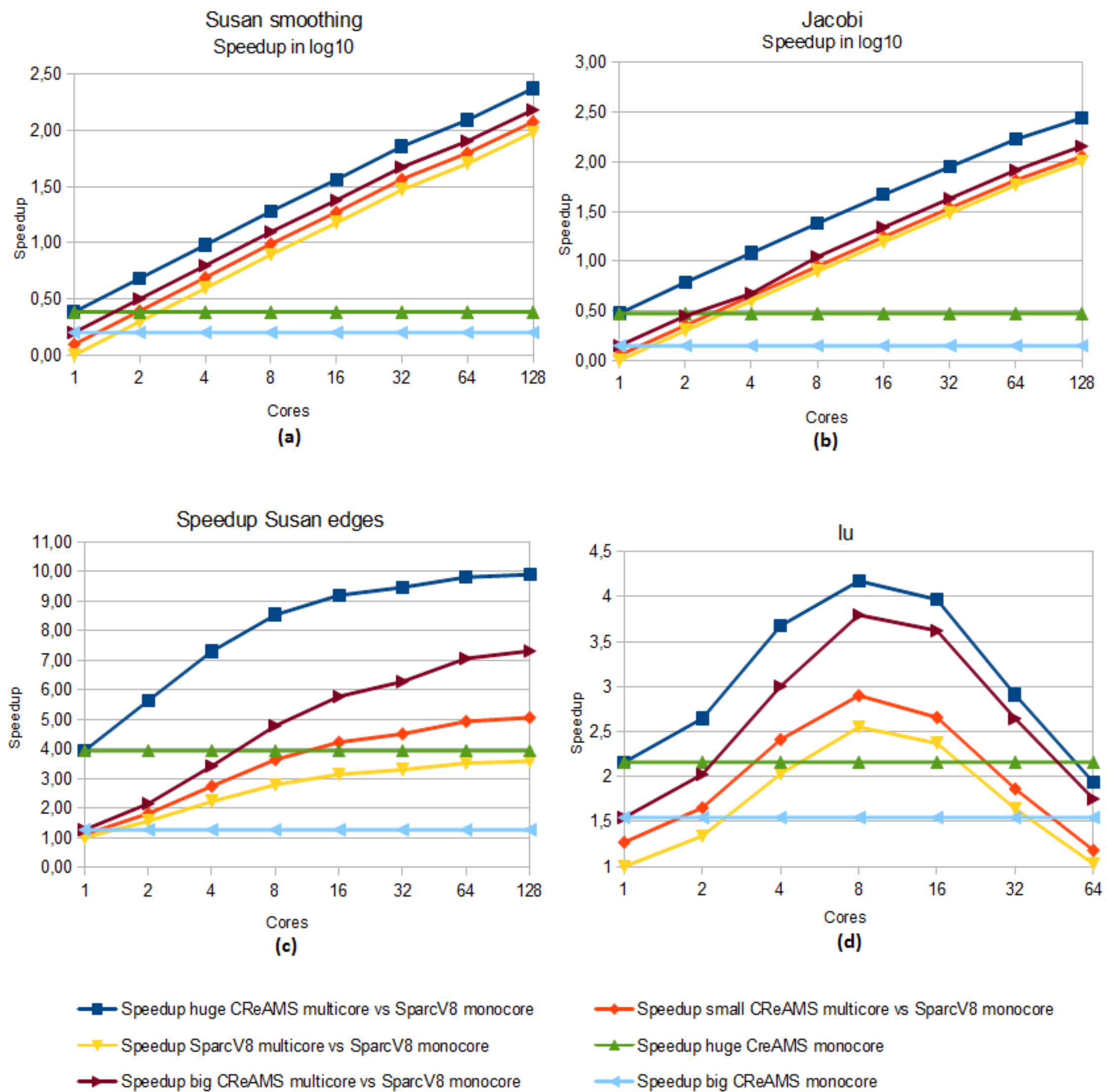


Figure 4: Speedup for the configurations simulated for each benchmark. (a) Susan smoothing. (b) Jacobi. (c) Susan edges. (d) lu.

6. FUTURE WORK

The next steps of this work will be the simulation of a variety of heterogeneous configurations of CReAMS. We will be comparing these results with homogeneous configurations – previously simulated and shown in section 5 – that have the same area.

After that, we will include on the simulation environment the communication overhead created by the exchange of information between the many cores of the system. A heterogeneous system tend to have less cores than a homogeneous of same size, so it is expected that the communication overhead will be more harmful to performance on homogeneous systems than on heterogeneous ones. We will be evaluating this assumption.

The ultimate objective of this work will be to find a heterogeneous configuration of CReAMS that has better performance than a homogeneous one of same area. This might be a big challenge, as the combinations of different sized DAP's are infinite. Moreover, some of the applications may take several hours to simulate, so time also becomes a constraint for this work. In order to reduce the simulation stage, we will try to find a small set of benchmarks which can represent the mean behavior of the whole set.

We believe that the factors presented on this report – lower communication overhead, efficient exploitation of ILP – will be the keys to achieve the foresaw results.

7. CHRONOGRAM

A chronogram of the activitivities that will be developed on this work is shown on table 3.

Table 3. Chronogram of activities

	Jul	Ago	Sep	Oct	Nov	Dec
Heterogeneous simulations	X					
Add communication overhead + simulations	X	X				
Write paper for conference		X	X			
Find better heterogeneous configuration			X	X	X	
Conclude work and write report					X	X

8. CONCLUSION

On section 5 we have showed that the homogeneous CReAMS is able to perform better than a standalone SparcV8 processor on most of the tested scenarios. It is observable that all applications are enhanced by the ILP exploitation given by the dynamic array processors. Moreover, CReAMS is also more energy efficient than the standalone SparcV8 processor when a power budget is considered[8].

Furthermore, on section 3.2 we have discussed why it is expected for a well scheduled heterogeneous system to perform better than a homogeneous one. The better area and power efficiency and smaller communication overhead are examples that support this assumption. On the next steps of this work, we will try to find configurations of heterogeneous CReAMS that confirms this performance improvement.

9. REFERENCES

[1] Beck, Antonio Carlos Schneider, Lang Lisbôa, Carlos Arthur,Carro, Luigi (Eds.), "Adaptable Embedded Systems", pages 41 – 43. Springer.

[2] Beck, A.C.S., Rutzig, M.B. Gaydadjiev,G., and Carro,L.. Transparent reconfigurable acceleration for heterogeneous embedded applications. In Proceedings of the conference on Design, automation and test in Europe (DATE'08). ACM, New York, NY, USA, 1208-1213.

[3] Rutzig, M. B., Beck, A. C. S., & Carro, L. (2013). A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Simultaneous ILP and TLP Exploitation. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013. Grenoble, France, 1559 – 1564.

[4] Woo S. C., Ohara M., Torrie E., Singh J.P., Gupta A.. The SPLASH-2 programs: Characterization and methodological considerations. INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (1995).

[5] M R Guthaus, J S Ringenberg, D Ernst, T M Austin, T Mudge, R B Brown. MiBench: A free, commercially representative embedded benchmark suite (2001). in Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC).

[6] Rutzig, M.B.. A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Efficient ILP and TLP Exploitation. 2012. pages 69 – 70. Doctor Thesis of Computer Science. Universidade Federal do Rio Grande do Sul (UFRGS).

[7] P S Magnusson, M Christensson, J Eskilson, D Forsgren, G Hllberg, J Hgberg, F Larsson, A Moestedt, B Werner. Simics: A full system simulation platform (2002).

[8] Rutzig, M.B.. A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Efficient ILP and TLP Exploitation. 2012. page 78. Doctor Thesis of Computer Science. Universidade Federal do Rio Grande do Sul (UFRGS).

[9] Watkins, M.A.; Albonesi, D.H., "Enabling Parallelization via a Reconfigurable Chip Multiprocessor", Workshop on Parallel Execution of Sequencial Programs on Multi-core Architectures, 2010. 37th International Symposium on Computer Architecture, June 2010.

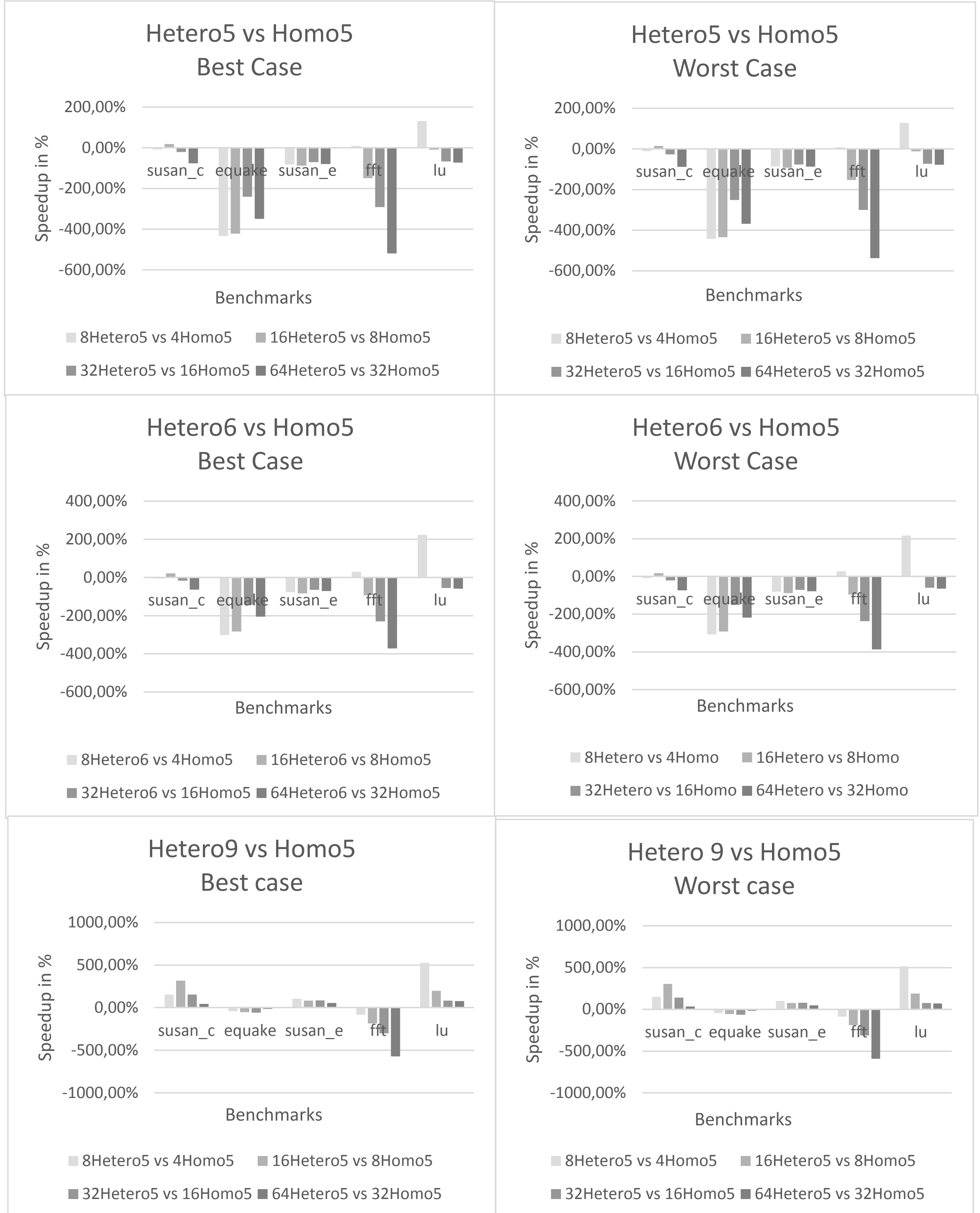
[10] Koenig, R.; Bauer, L.; Stripf, T.; Shafique, M.; Ahmed, W.; Becker, J.; Henkel, J.; "KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-SetMulti-grained-Array Architecture," Design, Automation &Test in Europe Conference, pp.819-824, 2010.

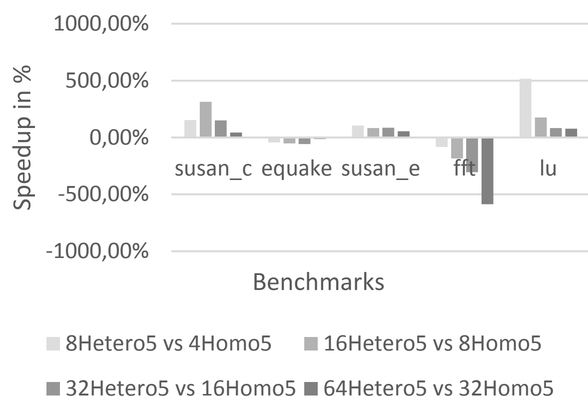
[11] Bienia C., Kumar S., Singh J.P., Li K.. The PARSEC benchmark suite: Characterization and architectural implications (2008). In Princeton University.

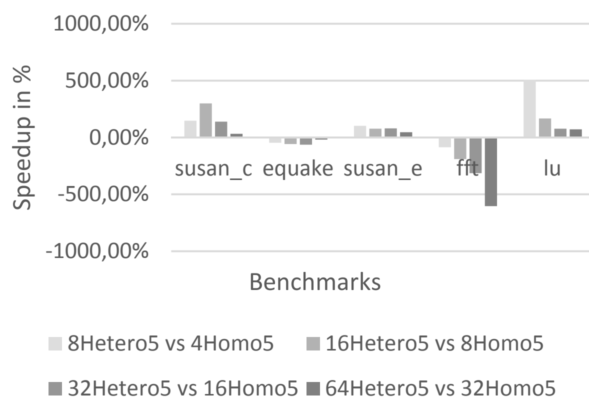
[12] A J Dorta, C Rodriguez, F de Sande, A Gonzalez-Escribano. The OpenMP source code repository (2005). In Proceedings of the 13th Euromicro conference on Parallel, Distributed and Network-Based Processing (PDP).

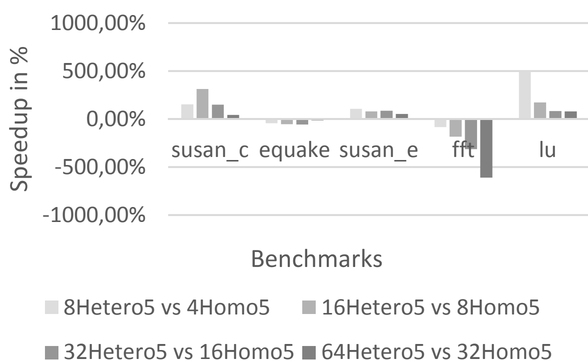
APPENDIX B

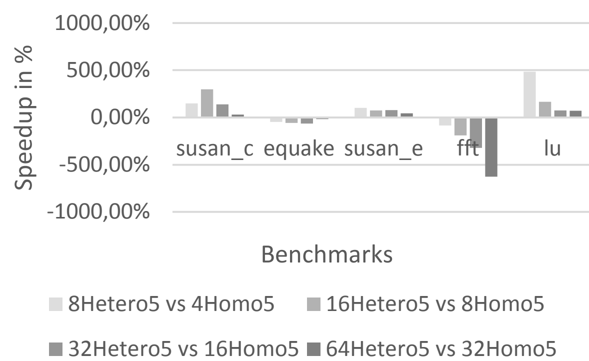
Results of heterogeneous configurations with smaller area than the homogeneous



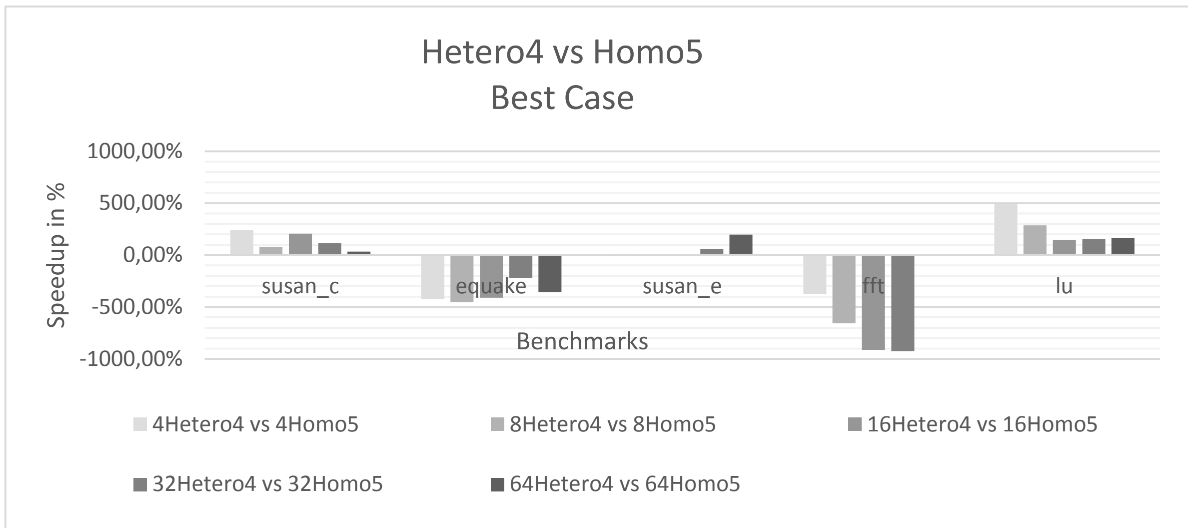

Hetero10 vs Homo5
Best case
1000,00%
500,00%
0,00%
-500,00%
-1000,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
8Hetero5 vs 4Homo5
16Hetero5 vs 8Homo5
32Hetero5 vs 16Homo5
64Hetero5 vs 32Homo5

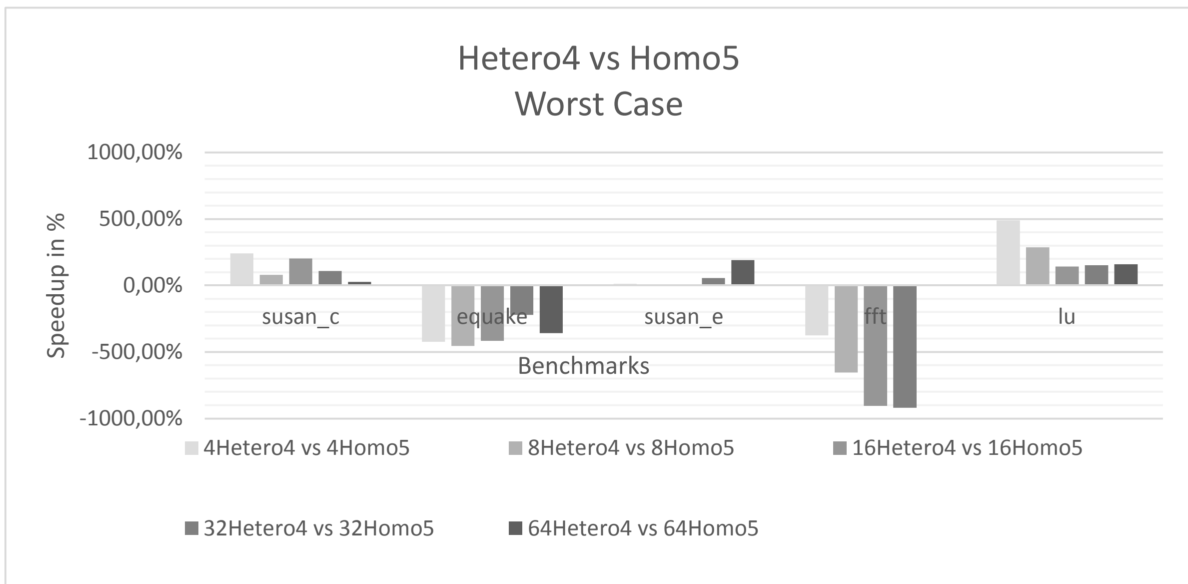

Hetero10 vs Homo5
Worst case
1000,00%
500,00%
0,00%
-500,00%
-1000,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
8Hetero5 vs 4Homo5
16Hetero5 vs 8Homo5
32Hetero5 vs 16Homo5
64Hetero5 vs 32Homo5

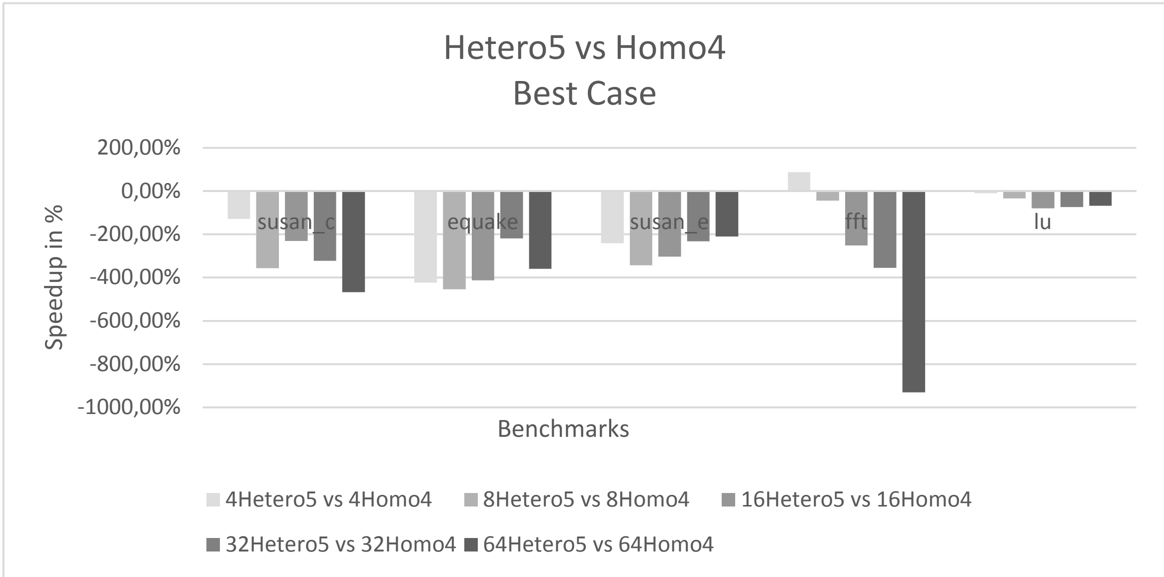

Hetero11 vs Homo5
Best case
1000,00%
500,00%
0,00%
-500,00%
-1000,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
8Hetero5 vs 4Homo5
16Hetero5 vs 8Homo5
32Hetero5 vs 16Homo5
64Hetero5 vs 32Homo5

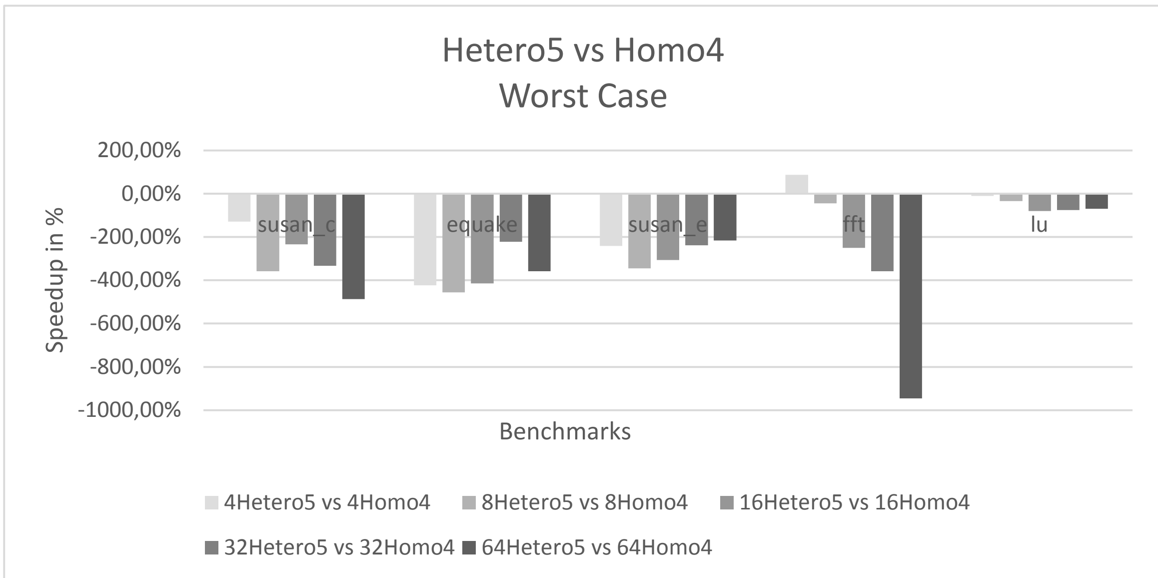

Hetero11 vs Homo5
Worst case
1000,00%
500,00%
0,00%
-500,00%
-1000,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
8Hetero5 vs 4Homo5
16Hetero5 vs 8Homo5
32Hetero5 vs 16Homo5
64Hetero5 vs 32Homo5

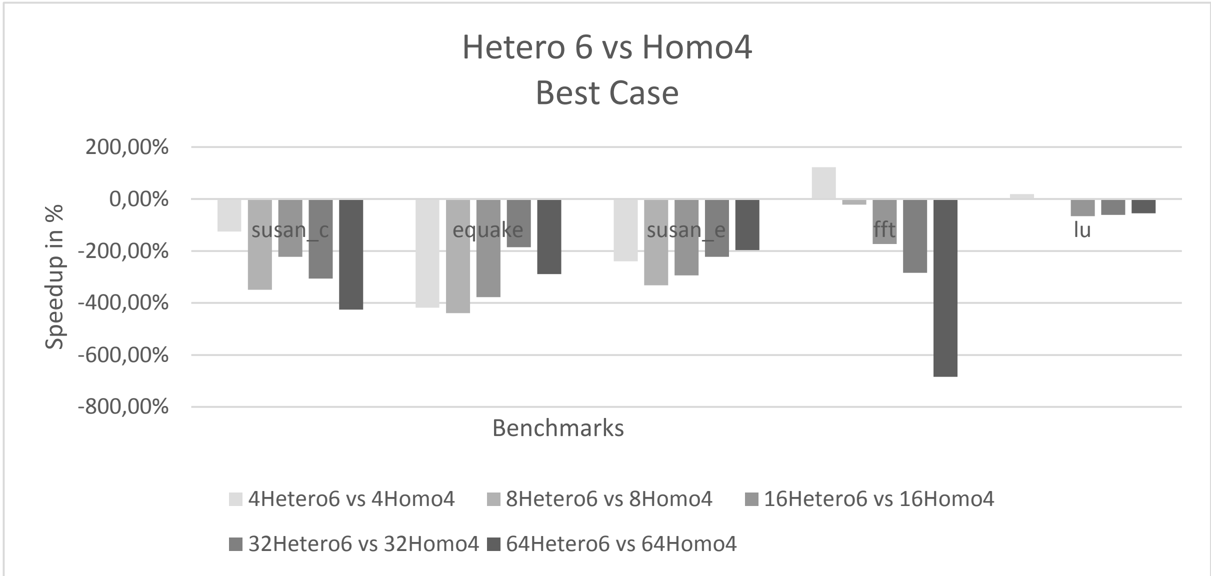
Results of heterogeneous configurations with same area than the homogeneous

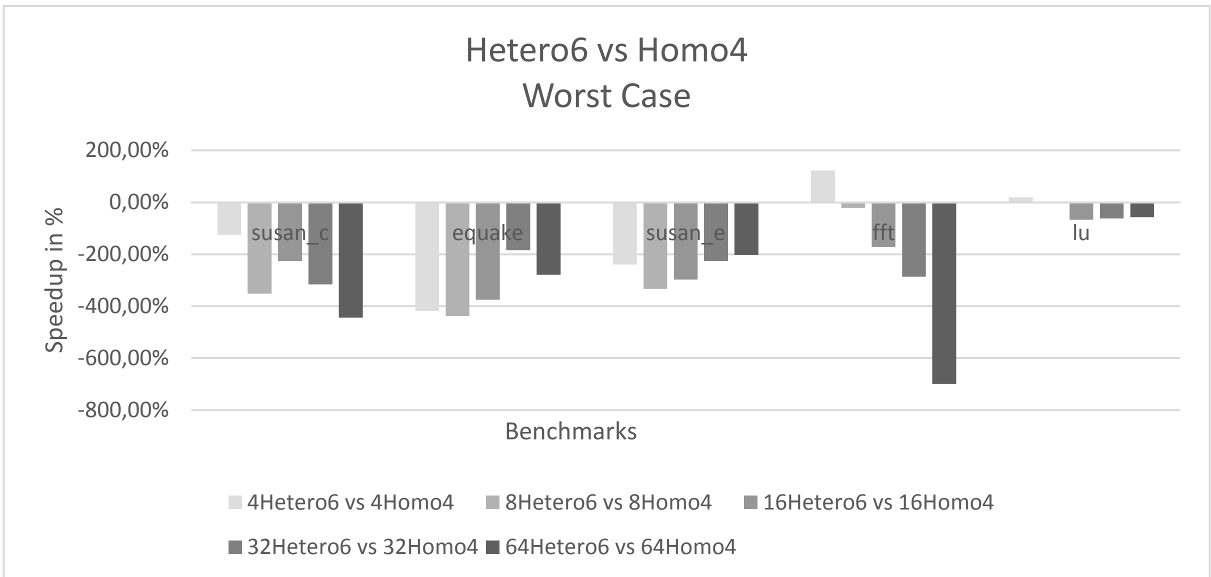


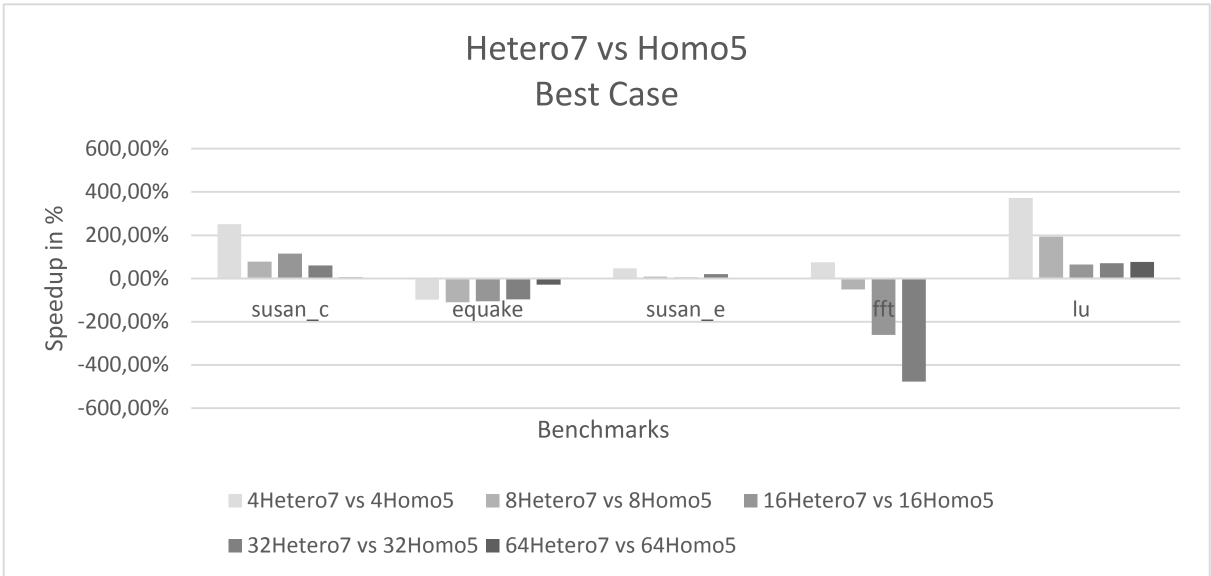


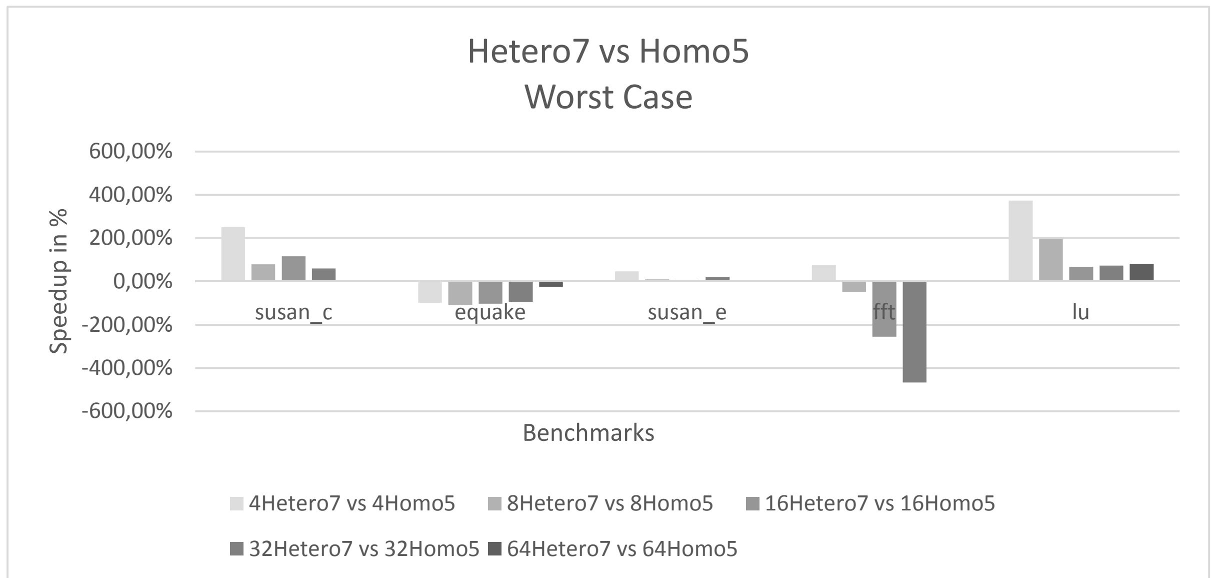

Hetero5 vs Homo4
Best Case
200,00%
0,00%
-200,00%
-400,00%
-600,00%
-800,00%
-1000,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero5 vs 4Homo4
8Hetero5 vs 8Homo4
16Hetero5 vs 16Homo4
32Hetero5 vs 32Homo4
64Hetero5 vs 64Homo4

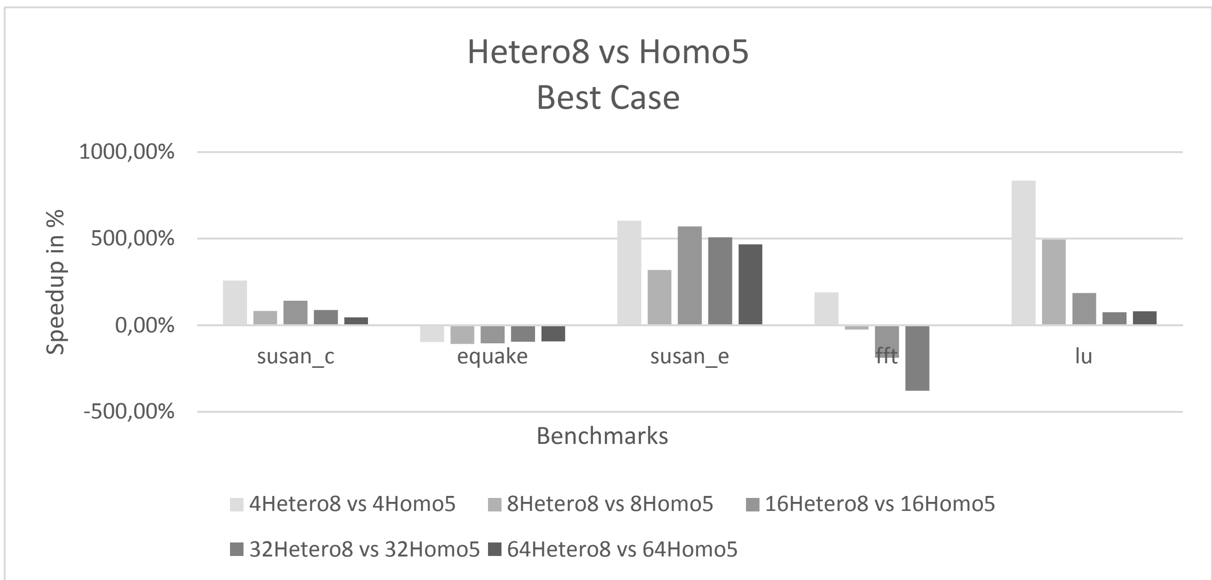

Hetero5 vs Homo4
Worst Case
200,00%
0,00%
-200,00%
-400,00%
-600,00%
-800,00%
-1000,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero5 vs 4Homo4
8Hetero5 vs 8Homo4
16Hetero5 vs 16Homo4
32Hetero5 vs 32Homo4
64Hetero5 vs 64Homo4

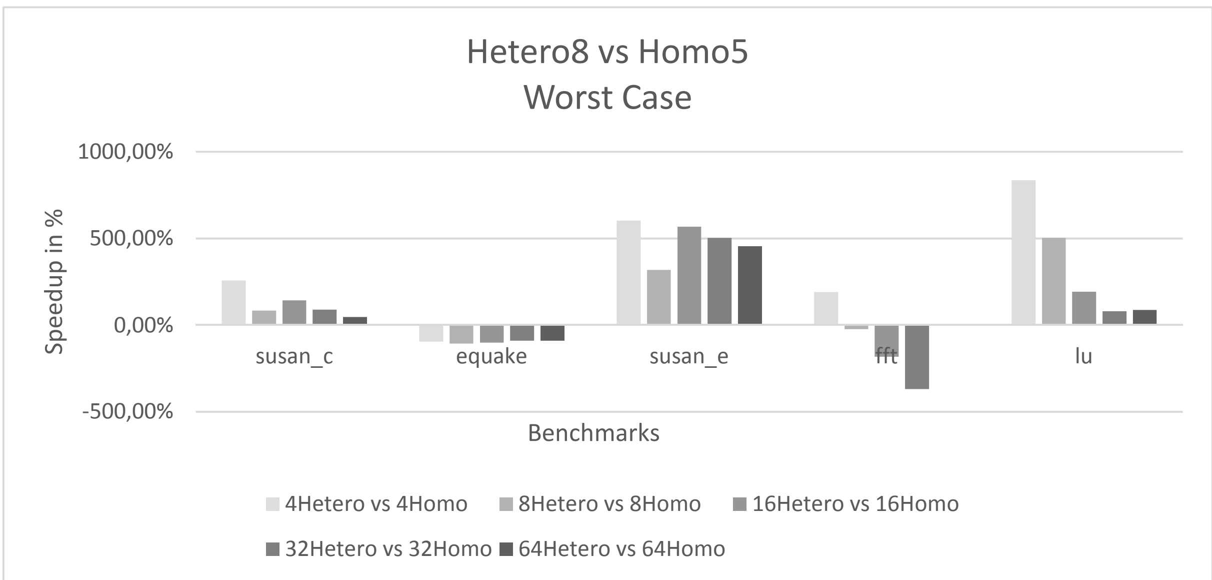

Hetero 6 vs Homo4
Best Case
200,00%
0,00%
-200,00%
-400,00%
-600,00%
-800,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero6 vs 4Homo4
8Hetero6 vs 8Homo4
16Hetero6 vs 16Homo4
32Hetero6 vs 32Homo4
64Hetero6 vs 64Homo4

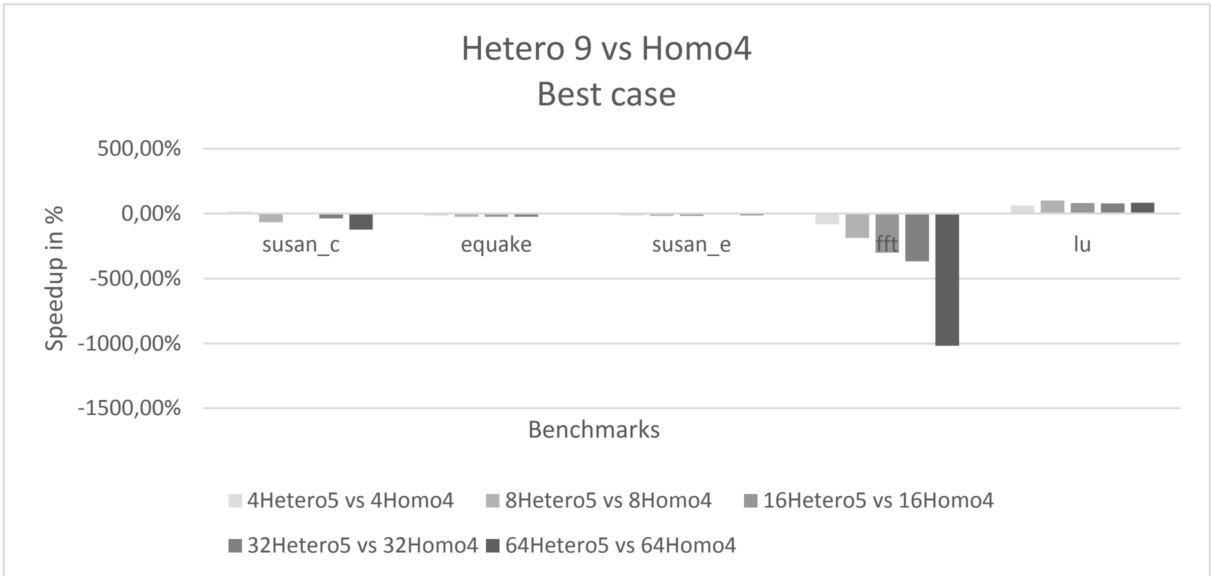

Hetero6 vs Homo4
Worst Case
200,00%
0,00%
-200,00%
-400,00%
-600,00%
-800,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero6 vs 4Homo4
8Hetero6 vs 8Homo4
16Hetero6 vs 16Homo4
32Hetero6 vs 32Homo4
64Hetero6 vs 64Homo4

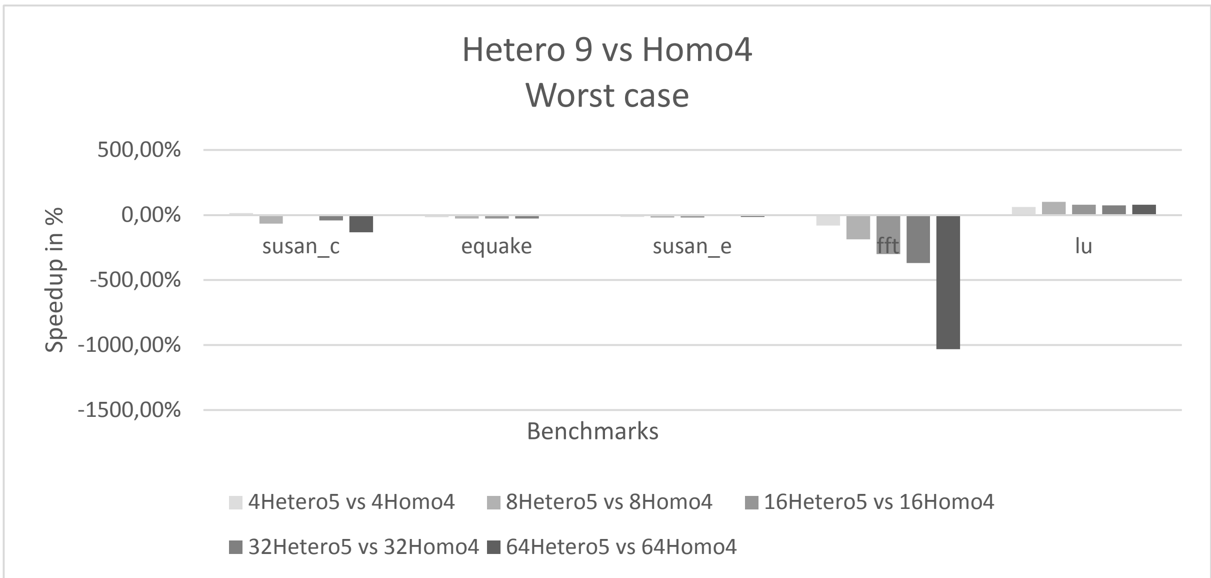

Hetero7 vs Homo5
Best Case
600,00%
400,00%
200,00%
0,00%
-200,00%
-400,00%
-600,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero7 vs 4Homo5
8Hetero7 vs 8Homo5
16Hetero7 vs 16Homo5
32Hetero7 vs 32Homo5
64Hetero7 vs 64Homo5

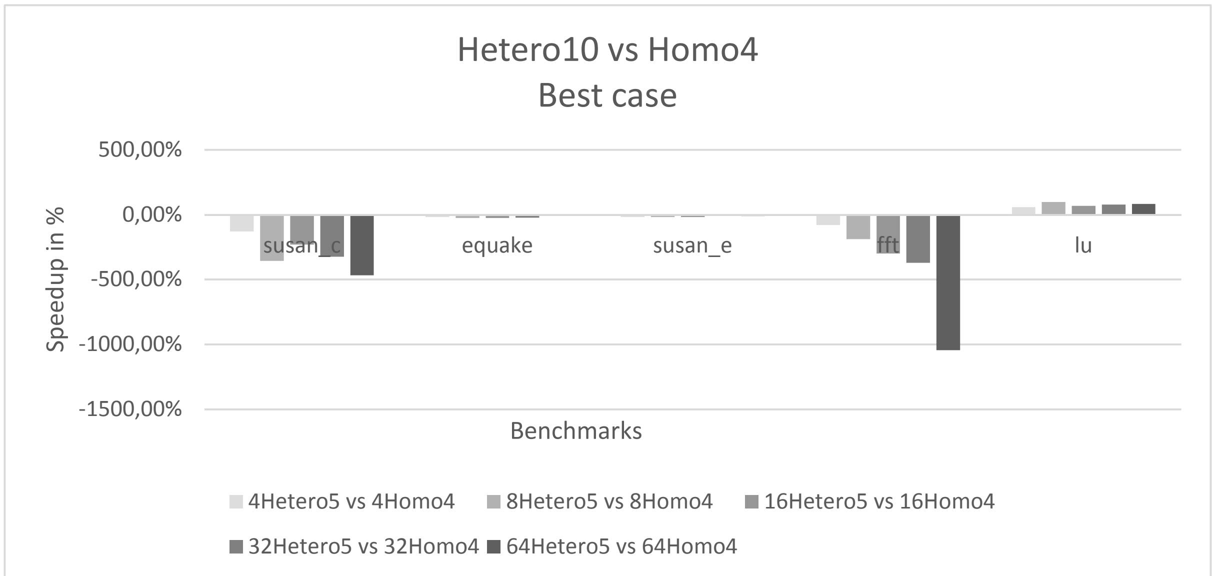

Hetero7 vs Homo5
Worst Case
600,00%
400,00%
200,00%
0,00%
-200,00%
-400,00%
-600,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero7 vs 4Homo5
8Hetero7 vs 8Homo5
16Hetero7 vs 16Homo5
32Hetero7 vs 32Homo5
64Hetero7 vs 64Homo5

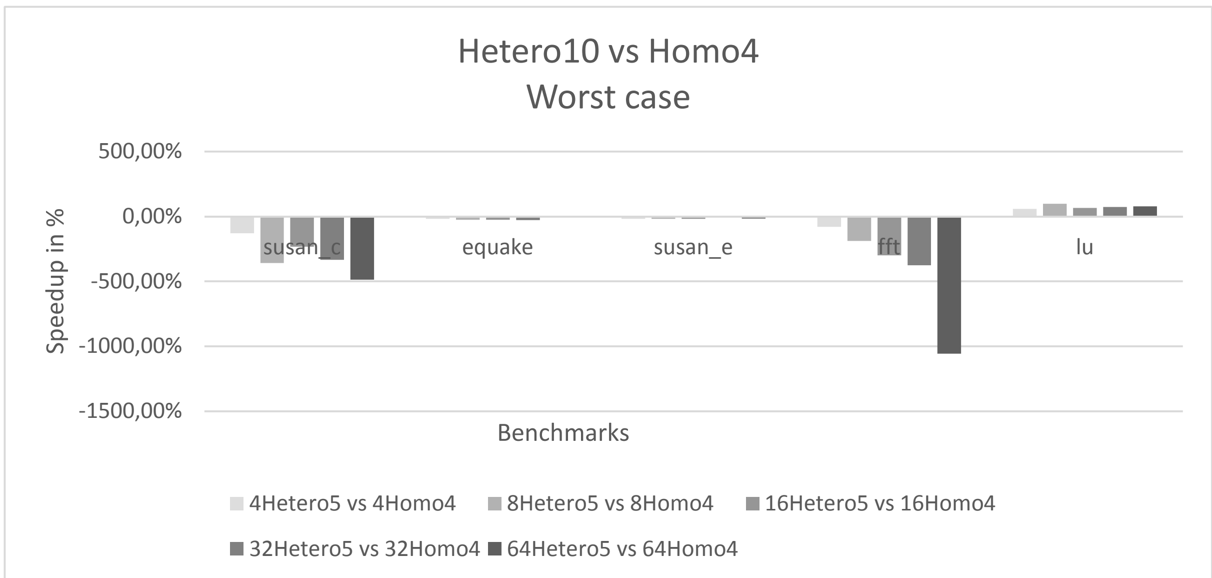

Hetero8 vs Homo5
Best Case
1000,00%
500,00%
0,00%
-500,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero8 vs 4Homo5
8Hetero8 vs 8Homo5
16Hetero8 vs 16Homo5
32Hetero8 vs 32Homo5
64Hetero8 vs 64Homo5

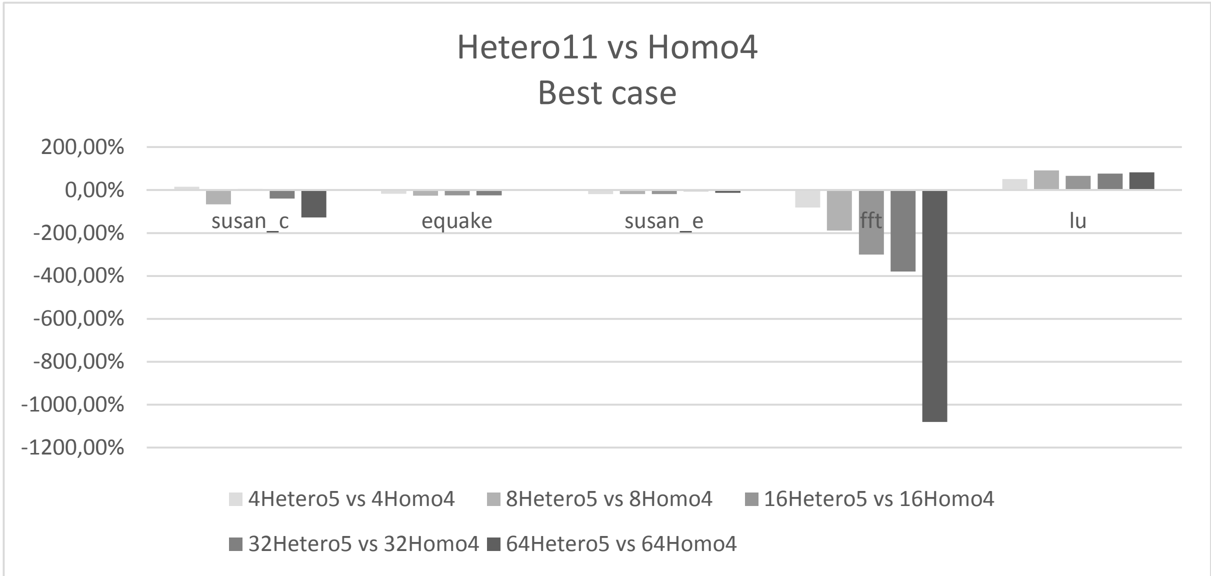

Hetero8 vs Homo5
Worst Case
1000,00%
500,00%
0,00%
-500,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero vs 4Homo
8Hetero vs 8Homo
16Hetero vs 16Homo
32Hetero vs 32Homo
64Hetero vs 64Homo

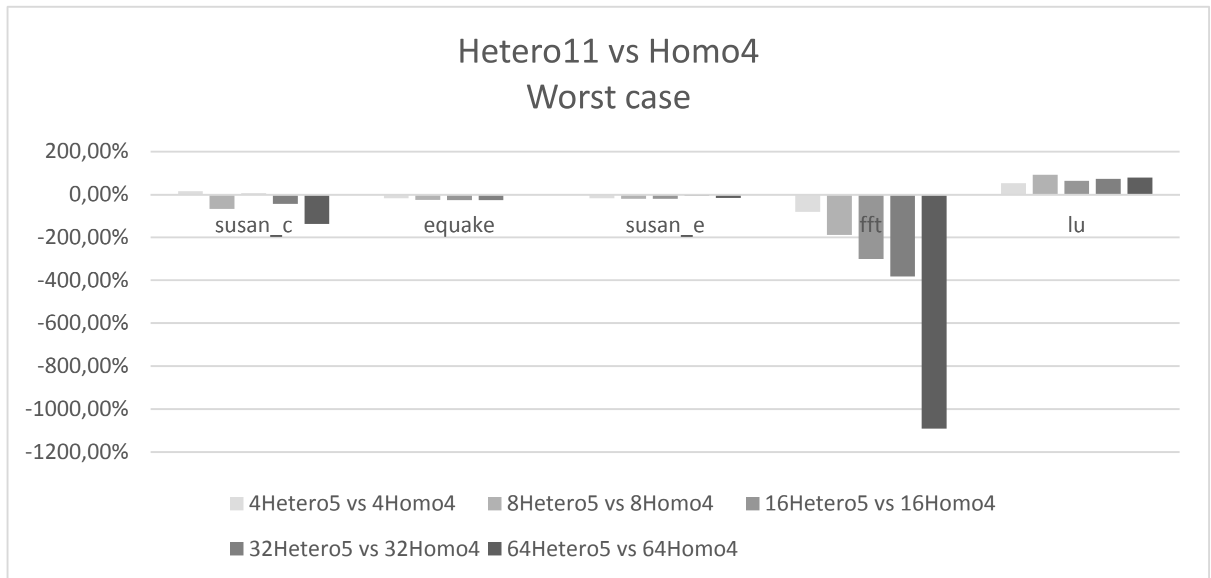

Hetero 9 vs Homo4
Best case
500,00%
0,00%
-500,00%
-1000,00%
-1500,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero5 vs 4Homo4
8Hetero5 vs 8Homo4
16Hetero5 vs 16Homo4
32Hetero5 vs 32Homo4
64Hetero5 vs 64Homo4


Hetero 9 vs Homo4
Worst case
500,00%
0,00%
-500,00%
-1000,00%
-1500,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero5 vs 4Homo4
8Hetero5 vs 8Homo4
16Hetero5 vs 16Homo4
32Hetero5 vs 32Homo4
64Hetero5 vs 64Homo4

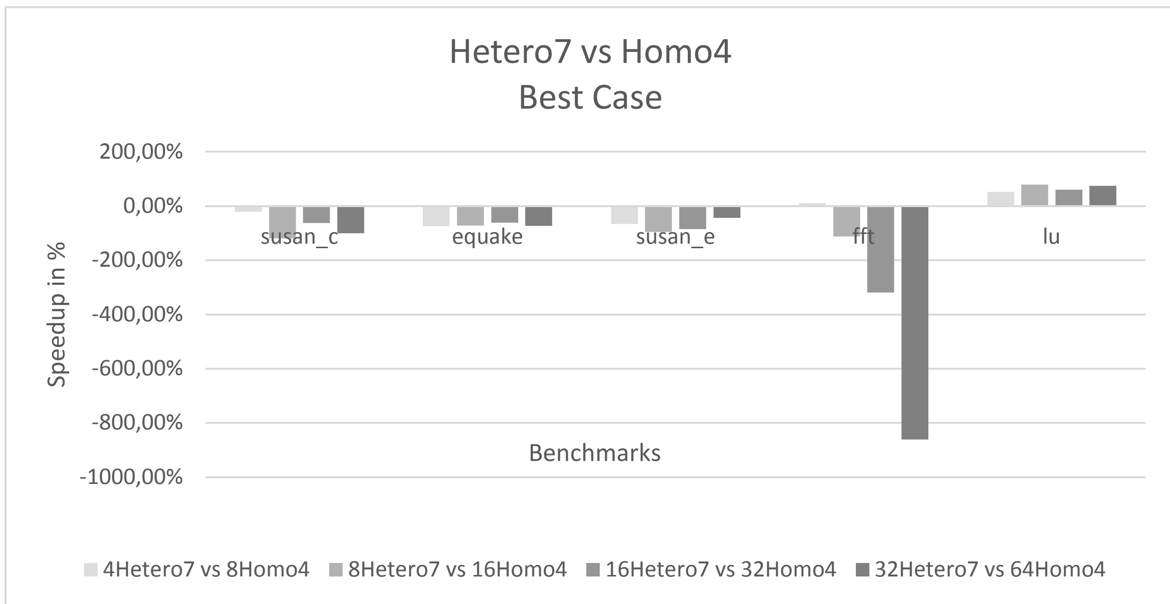

Hetero10 vs Homo4
Best case
500,00%
0,00%
-500,00%
-1000,00%
-1500,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero5 vs 4Homo4
8Hetero5 vs 8Homo4
16Hetero5 vs 16Homo4
32Hetero5 vs 32Homo4
64Hetero5 vs 64Homo4

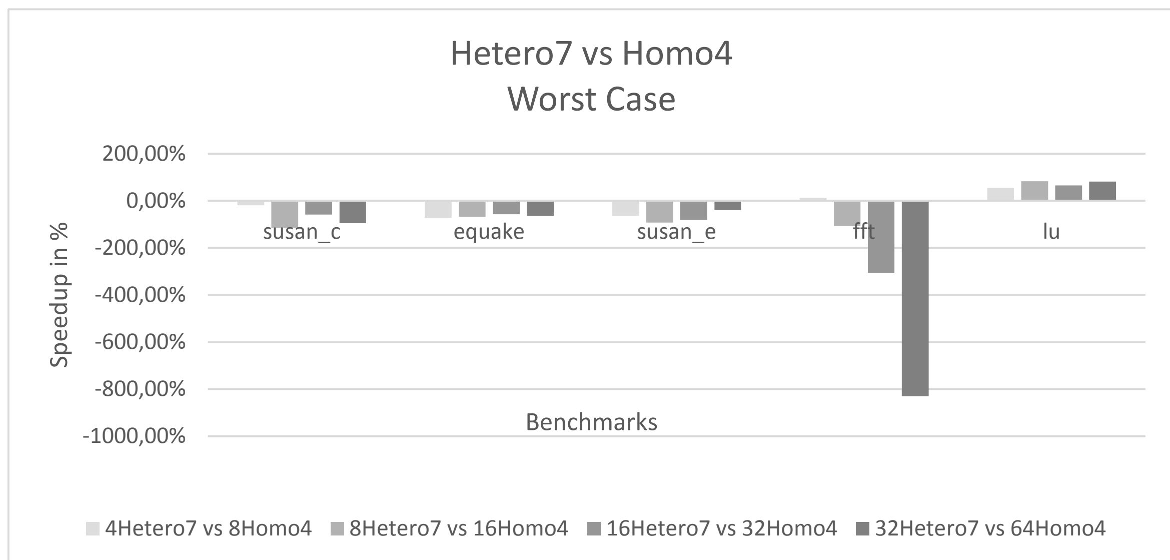

Hetero10 vs Homo4
Worst case
500,00%
0,00%
-500,00%
-1000,00%
-1500,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero5 vs 4Homo4
8Hetero5 vs 8Homo4
16Hetero5 vs 16Homo4
32Hetero5 vs 32Homo4
64Hetero5 vs 64Homo4

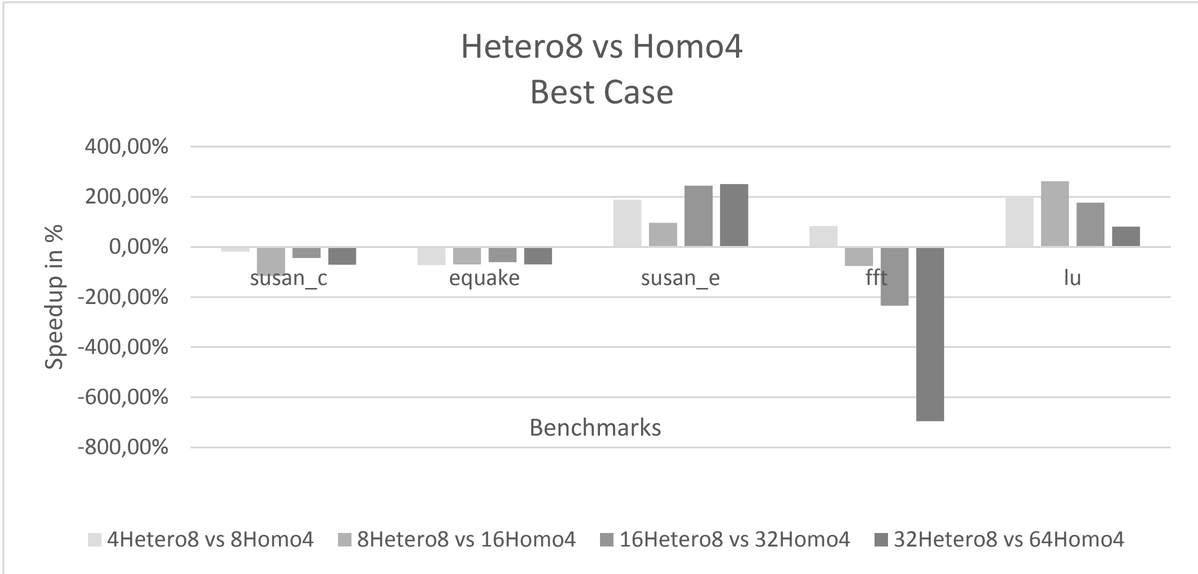

Hetero11 vs Homo4
Best case
200,00%
0,00%
-200,00%
-400,00%
-600,00%
-800,00%
-1000,00%
-1200,00%
susan_c
equake
susan_e
fft
lu
4Hetero5 vs 4Homo4
8Hetero5 vs 8Homo4
16Hetero5 vs 16Homo4
32Hetero5 vs 32Homo4
64Hetero5 vs 64Homo4

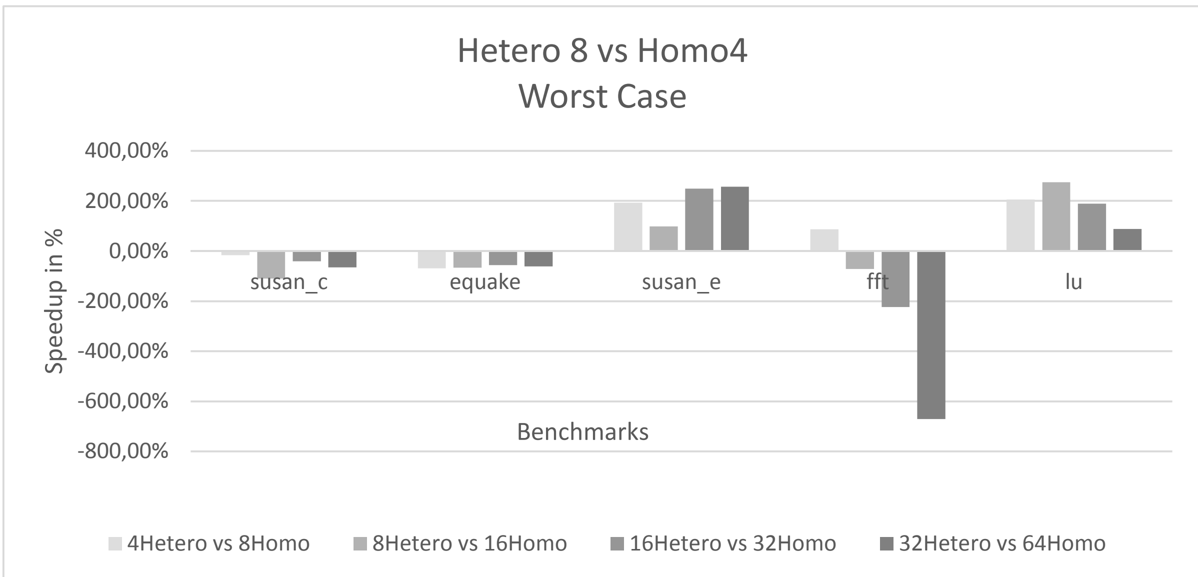

Hetero11 vs Homo4
Worst case
200,00%
0,00%
-200,00%
-400,00%
-600,00%
-800,00%
-1000,00%
-1200,00%
susan_c
equake
susan_e
fft
lu
4Hetero5 vs 4Homo4
8Hetero5 vs 8Homo4
16Hetero5 vs 16Homo4
32Hetero5 vs 32Homo4
64Hetero5 vs 64Homo4

Results of heterogeneous configurations with bigger area than the homogeneous

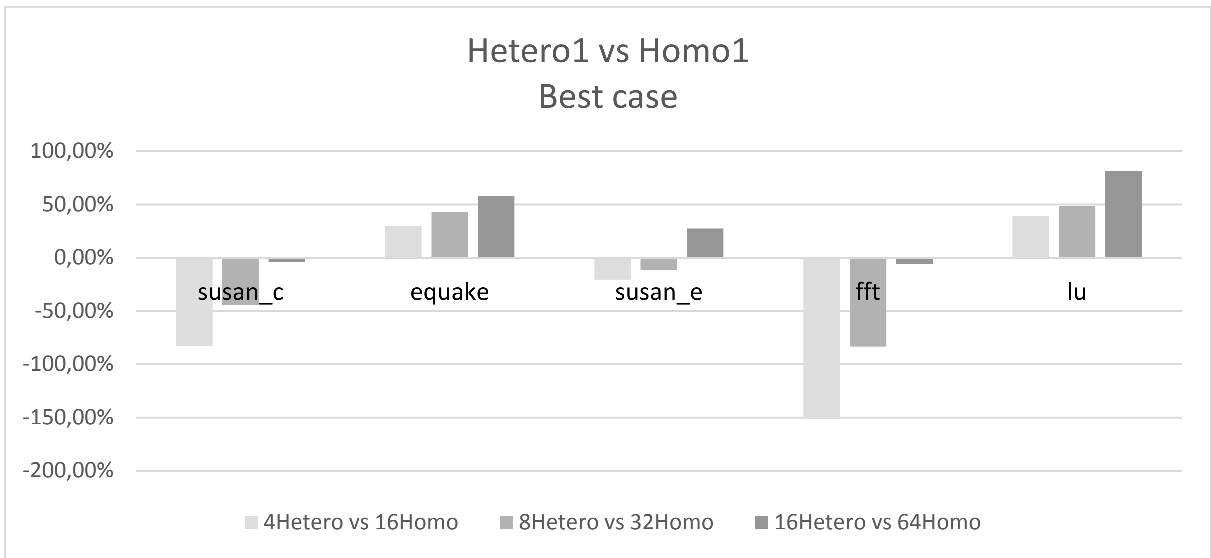


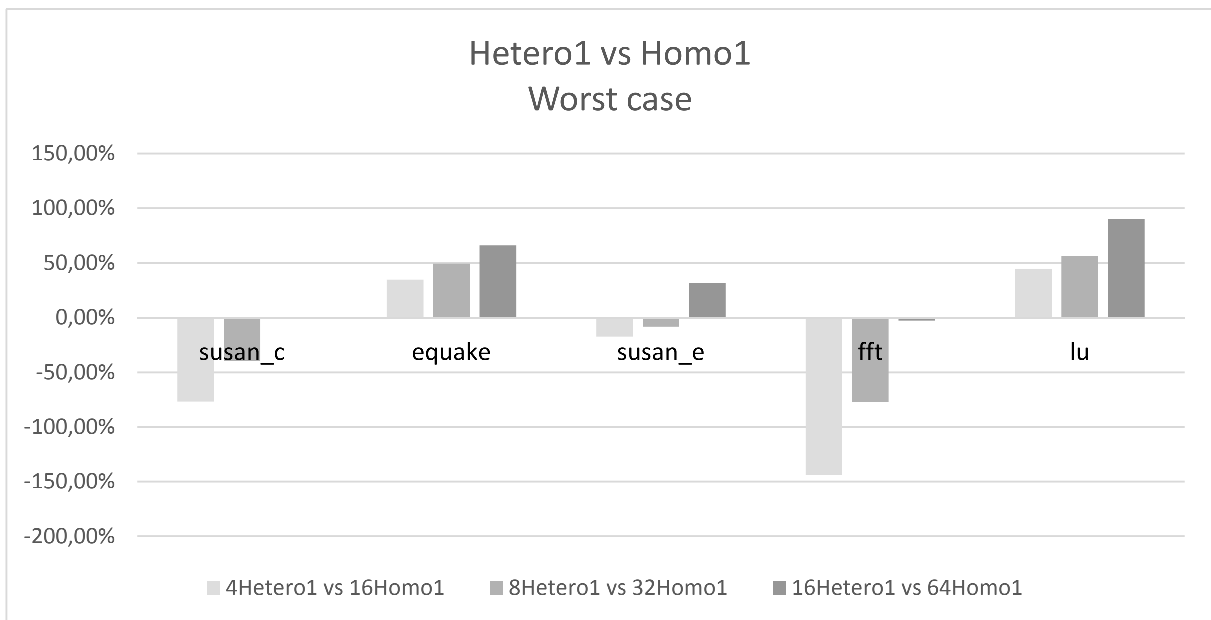


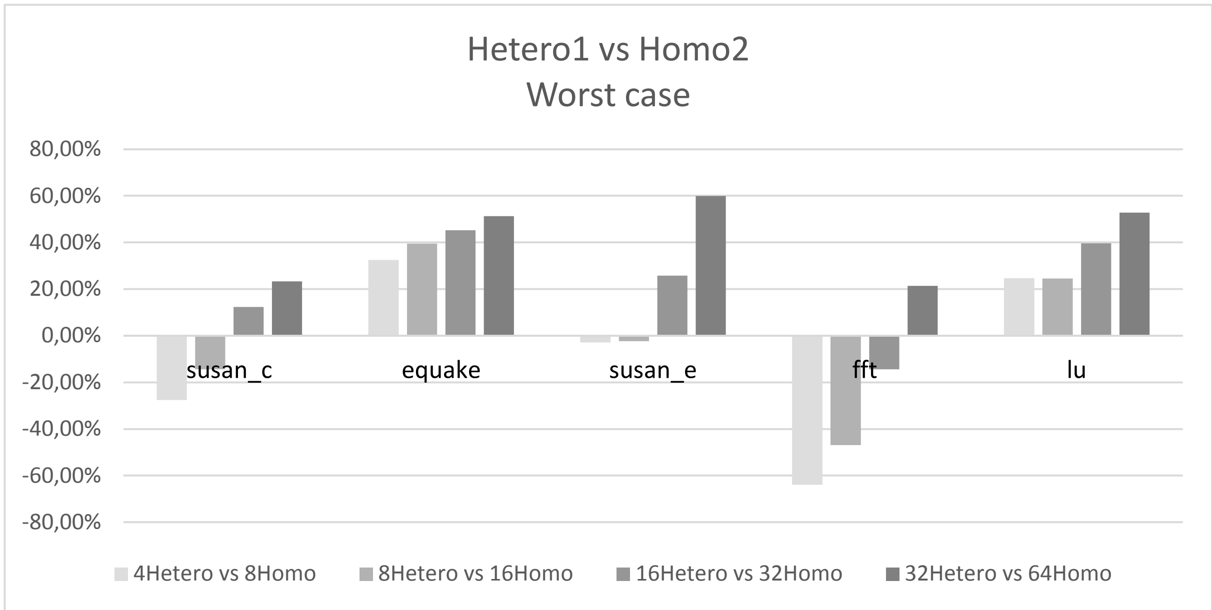

Hetero8 vs Homo4
Best Case
400,00%
200,00%
0,00%
-200,00%
-400,00%
-600,00%
-800,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero8 vs 8Homo4
8Hetero8 vs 16Homo4
16Hetero8 vs 32Homo4
32Hetero8 vs 64Homo4

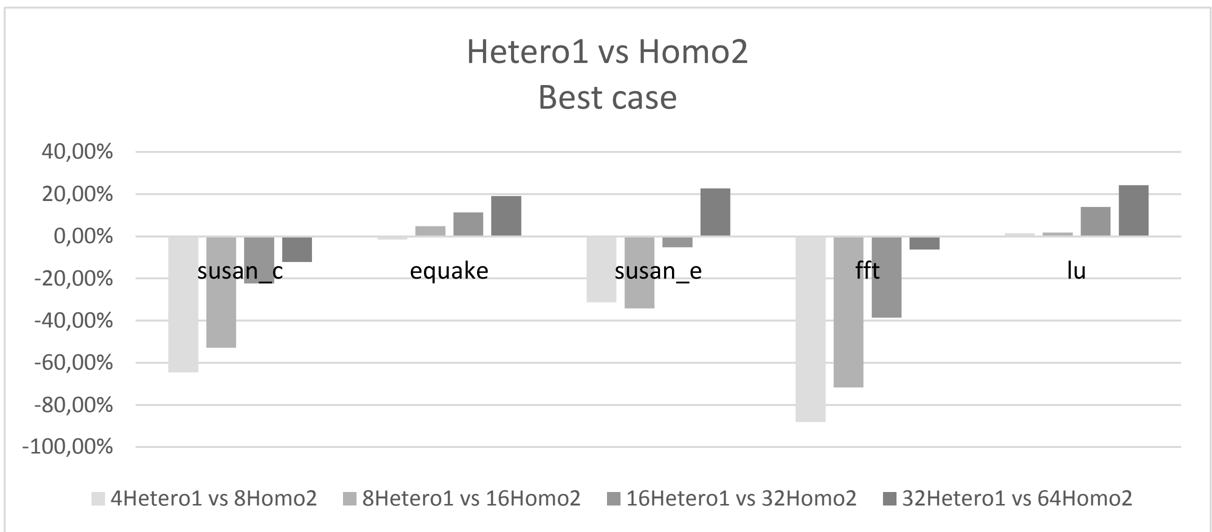

Hetero 8 vs Homo4
Worst Case
400,00%
200,00%
0,00%
-200,00%
-400,00%
-600,00%
-800,00%
Speedup in %
susan_c
equake
susan_e
fft
lu
Benchmarks
4Hetero vs 8Homo
8Hetero vs 16Homo
16Hetero vs 32Homo
32Hetero vs 64Homo

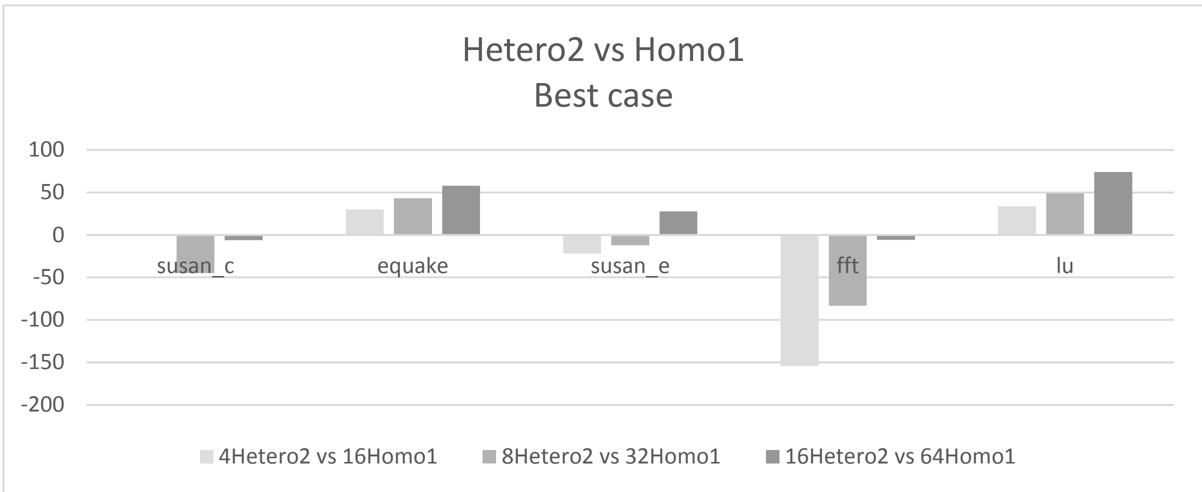
Results of heterogeneous configurations without scheduler

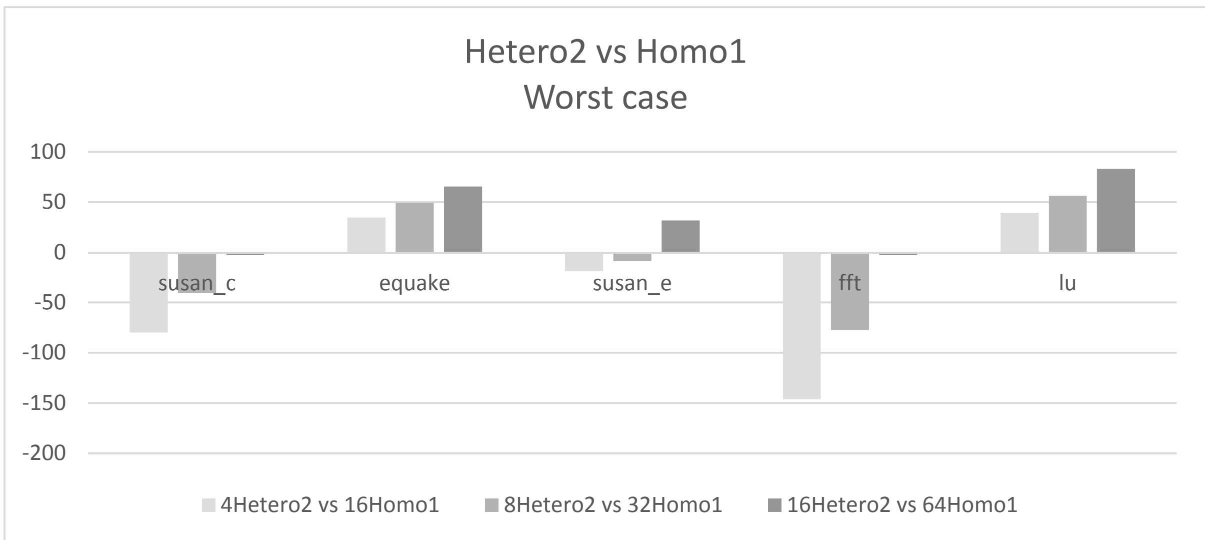


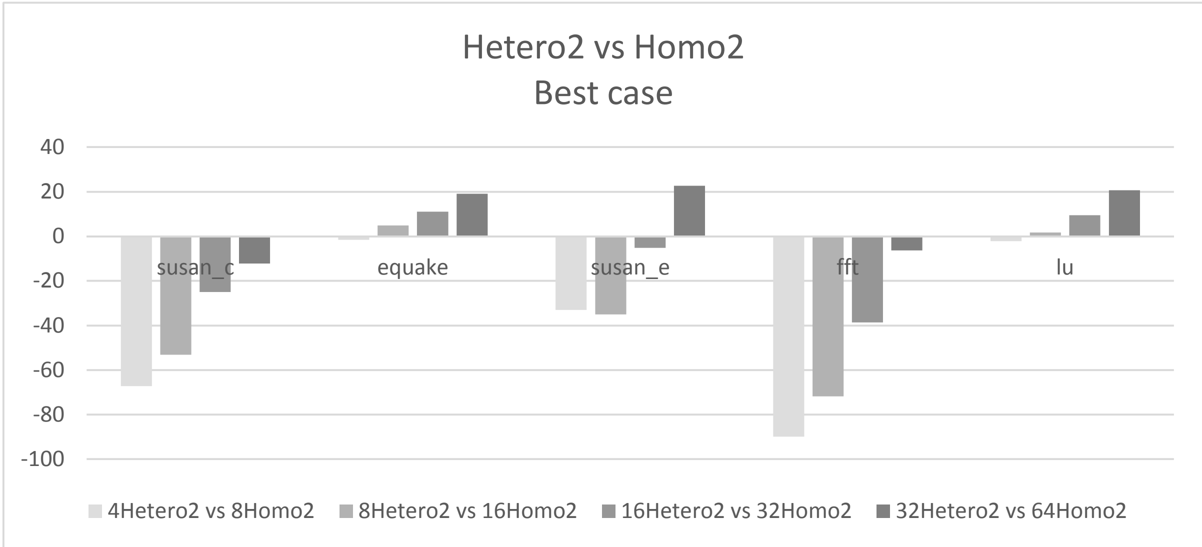


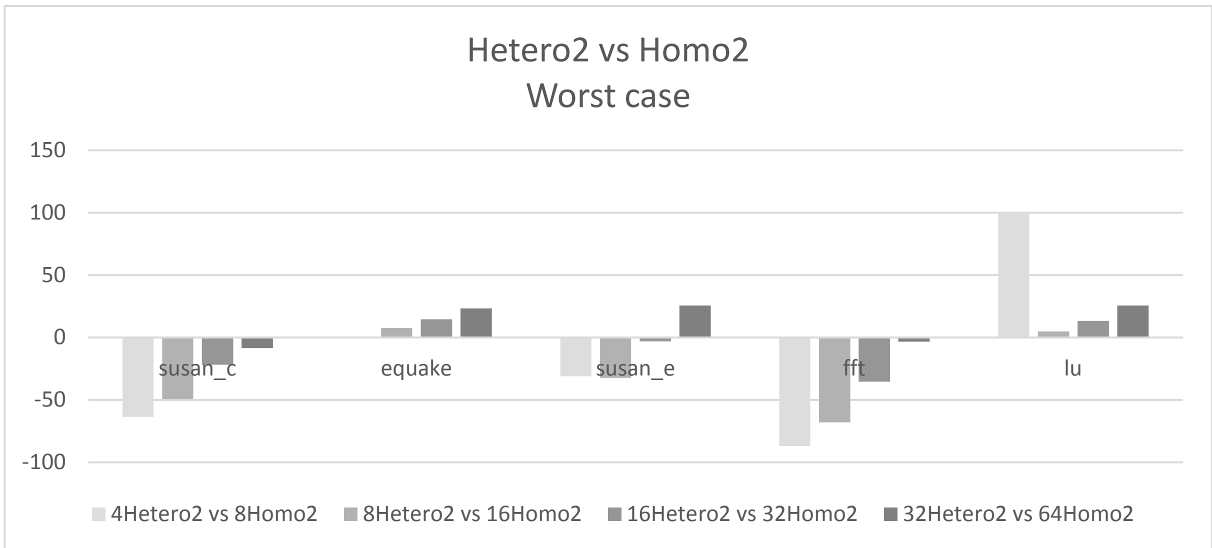

Hetero1 vs Homo2
Worst case
80,00%
60,00%
40,00%
20,00%
0,00%
-20,00%
-40,00%
-60,00%
-80,00%
susan_c
equake
susan_e
fft
lu
4Hetero vs 8Homo
8Hetero vs 16Homo
16Hetero vs 32Homo
32Hetero vs 64Homo

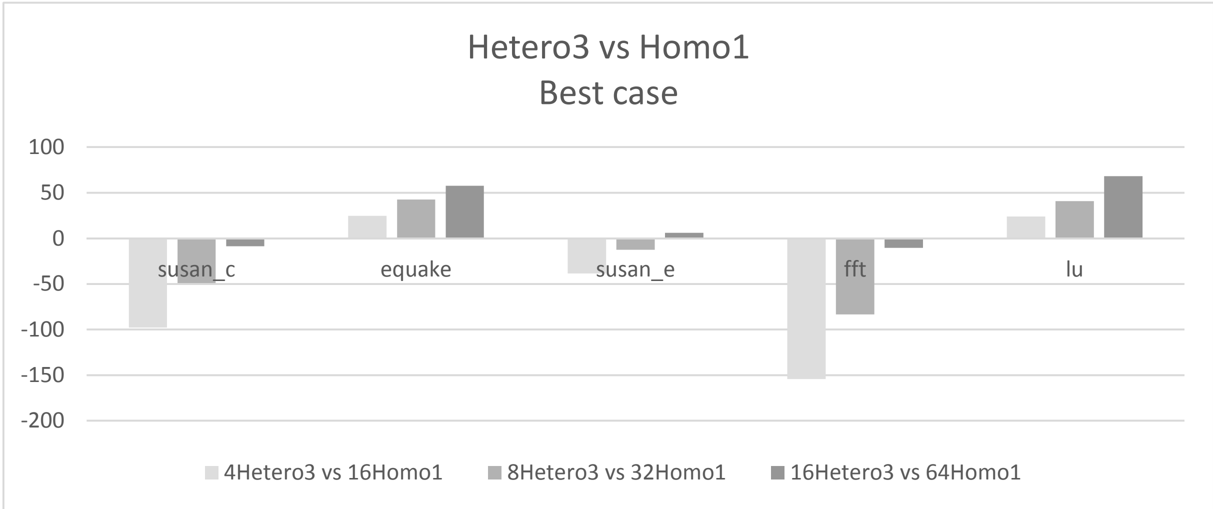

Hetero1 vs Homo2
Best case
40,00%
20,00%
0,00%
-20,00%
-40,00%
-60,00%
-80,00%
-100,00%
susan_c
equake
susan_e
fft
lu
4Hetero1 vs 8Homo2
8Hetero1 vs 16Homo2
16Hetero1 vs 32Homo2
32Hetero1 vs 64Homo2


Hetero2 vs Homo1
Best case
100
50
0
-50
-100
-150
-200
susan_c
equake
susan_e
fft
lu
4Hetero2 vs 16Homo1
8Hetero2 vs 32Homo1
16Hetero2 vs 64Homo1

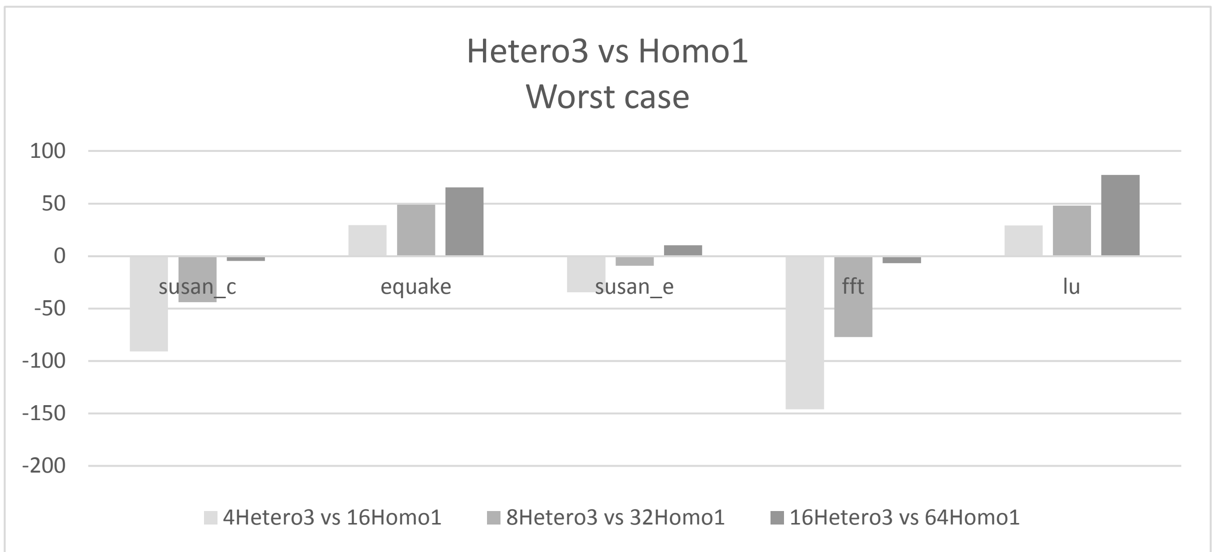

Hetero2 vs Homo1
Worst case
100
50
0
-50
-100
-150
-200
susan_c
equake
susan_e
fft
lu
4Hetero2 vs 16Homo1
8Hetero2 vs 32Homo1
16Hetero2 vs 64Homo1


Hetero2 vs Homo2
Best case
40
20
0
-20
-40
-60
-80
-100
susan_c
equake
susan_e
fft
lu
4Hetero2 vs 8Homo2
8Hetero2 vs 16Homo2
16Hetero2 vs 32Homo2
32Hetero2 vs 64Homo2

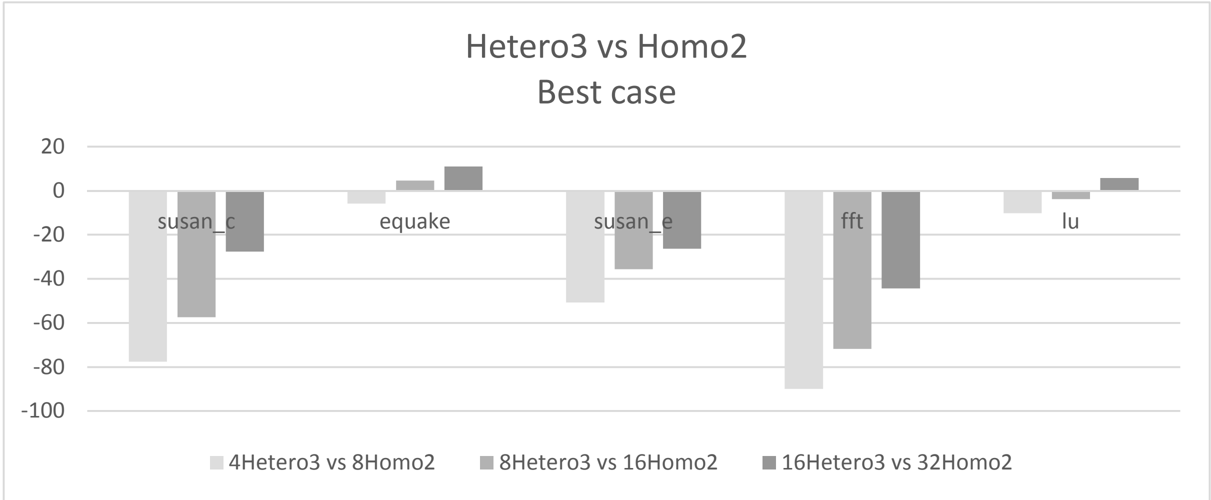

Hetero2 vs Homo2
Worst case
150
100
50
0
-50
-100
susan_c
equake
susan_e
fft
lu
4Hetero2 vs 8Homo2
8Hetero2 vs 16Homo2
16Hetero2 vs 32Homo2
32Hetero2 vs 64Homo2

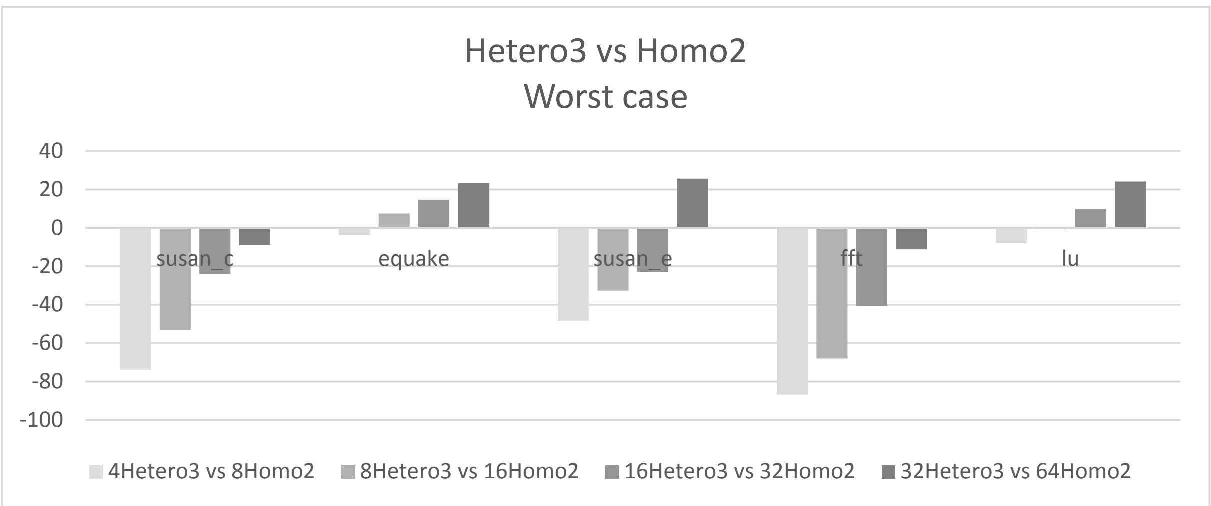


Hetero3 vs Homo1
Best case
100
50
0
-50
-100
-150
-200
susan_c
equake
susan_e
fft
lu
4Hetero3 vs 16Homo1
8Hetero3 vs 32Homo1
16Hetero3 vs 64Homo1



Hetero3 vs Homo1
Worst case
100
50
0
-50
-100
-150
-200
susan_c
equake
susan_e
fft
lu
4Hetero3 vs 16Homo1
8Hetero3 vs 32Homo1
16Hetero3 vs 64Homo1


Hetero3 vs Homo2
Best case
20
0
-20
-40
-60
-80
-100
susan_c
equake
susan_e
fft
lu
4Hetero3 vs 8Homo2
8Hetero3 vs 16Homo2
16Hetero3 vs 32Homo2


Hetero3 vs Homo2
Worst case
40
20
0
-20
-40
-60
-80
-100
susan_c
equake
susan_e
fft
lu
4Hetero3 vs 8Homo2
8Hetero3 vs 16Homo2
16Hetero3 vs 32Homo2
32Hetero3 vs 64Homo2