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

JECKSON DELLAGOSTIN SOUZA

A Reconfigurable Heterogeneous Multicore System with Homogeneous ISA

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Antonio Carlos Schneider Beck

Porto Alegre
2016

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Dellagostin Souza, Jeckson

A Reconfigurable Heterogeneous Multicore System with Homogeneous ISA / Jeckson Dellagostin Souza. – 2016.

126 f.

Orientador: Antonio Carlos Schneider Beck Filho

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2016.

1.Heterogeneity. 2.Multicore 3.Scheduling 4.Reconfigurable Architectures 5.Embedded Systems I. Beck, Antonio Carlos Schneider. II. A Reconfigurable Heterogeneous Multicore System with Homogeneous ISA.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento

Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb

Coordenador do PPGC: Prof. Luigi Carro

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço à minha família, por apoiar e acreditar em todas as minhas decisões. Tanto ao incentivo de meu pai Valério, como no apoio e abrigo de minha mãe Genelci, ambos nunca me faltaram e nunca duvidaram da minha escolha.

Agradeço também aos meus amigos, tanto os velhos companheiros de graduação quanto aos novos que conheci durante este ano de pós. Não só de trabalho e pesquisa um pós graduando vive, e vocês me ajudaram a esquecer o trabalho nos momentos em que ele deveria ser esquecido. Em especial, agradeço a Alessandra Leonhardt por todo o incentivo que ela me deu em seguir minhas ambições e ingressar no doutorado. Um agradecimento especial também para o Marcelo Brandalero, que não só dividiu o fardo de um ano de pós comigo, mas também me ajudou diversas vezes com ideias para este trabalho durante as suas inúmeras caronas.

Aos colegas do laboratório de sistemas embarcados, agradeço por toda a ajuda e apoio que me foi dado. Não houve um momento deste ano em que eu não pude contar com a ajuda e, ainda mais importante, a amizade de vocês. Agradeço também ao meu orientador, que foi fundamental para que eu cumprisse todos os meus *deadlines*. Sem a experiência e os conselhos dele, eu nunca estaria onde estou hoje.

Agradeço ao CNPQ por ter financiado minha bolsa de mestrado e pela fomentação à pesquisa no país.

Por fim, agradeço a todas as pessoas que passaram pela minha vida neste ano de pós-graduação. Todos contribuíram, de alguma forma, com minha formação intelectual e pessoal.

Obrigado a todos.

Um sistema Multinúcleo, Heterogêneo e Reconfigurável de ISA Homogênea.

RESUMO

Dada a grande diversidade de aplicações embarcadas presentes nos atuais dispositivos portáveis, ambos os paralelismos em nível de threads e de instruções devem ser explorados para obter ganhos de desempenho e energia. Enquanto MPSoCs (sistemas em chip de múltiplos núcleos) são amplamente usados para esse propósito, estes falham quando consideramos produtividade de software, já que eles são compostos de chips com diferentes arquiteturas que precisam ser programados separadamente. Por outro lado, processadores multi núcleos de propósito geral implementam a mesma arquitetura, mas são compostos de núcleos homogêneos de processadores superescalares que consomem muita potência. Nesta dissertação, propõe-se um novo sistema, que tira proveito de circuitos reconfiguráveis para criar diferentes organizações que implementam a mesma arquitetura, capazes de apresentar alto desempenho com baixo custo energético. Para garantir a compatibilidade binária, usa-se um mecanismo de tradução binária que transforma o código a ser executado no circuito reconfigurável durante a execução. Usando aplicações representativas, mostra-se que uma versão do sistema heterogêneo pode ganhar da sua versão homogênea em média de 59% em desempenho e 10% em energia, com melhoras em EDP (*Energy-Delay Product* – Produto da energia pelo tempo de execução) em quase todos os cenários.

Além disso, este trabalho também propõe e avalia seis escalonadores para este sistema heterogêneo: dois algoritmos estáticos, os quais alocam as threads no primeiro núcleo livre, onde elas permanecerão durante toda a execução; um escalonador direcionado por contagem de instruções, o qual realoca as threads durante pontos de sincronização de acordo com a sua contagem de instruções; um escalonador de *Feedback*, que usa dados de dentro da unidade reconfigurável para realocar threads; o *PC-Feedback*, que adiciona um mecanismo de reuso de dados ao último escalonador; e um escalonador Oráculo, que é capaz de decidir a melhor alocação de threads possível. Mostra-se que o algoritmo estático pode ter alto desempenho em aplicações com alto paralelismo, contudo para um desempenho mais uniforme em todas as aplicações os algoritmos de *Feedback* e *PC-Feedback* são mais indicados.

Palavras-chave: heterogeneidade; multi núcleos; alocação; arquiteturas reconfiguráveis; sistemas embarcados

ABSTRACT

Given the large diversity of embedded applications one can find in current portable devices, for energy and performance reasons one must exploit both Thread- and Instruction Level Parallelism. While MPSoCs (Multiprocessor system-on-chip) are largely used for this purpose, they fail when one considers software productivity, since it comprises different ISAs (Instruction Set Architecture) that must be programmed separately. On the other hand, general purpose multicores implement the same ISA, but are composed of a homogeneous set of very power consuming superscalar processors. In this dissertation, we show how one can effectively use a reconfigurable unit to provide a number of different possible heterogeneous configurations while still sustaining the same ISA, capable of reaching high performance with low energy cost. To ensure ISA compatibility, we use a binary translation mechanism that transforms code to be executed on the fabric at run-time. Using representative benchmarks, we show that one version of the heterogeneous system can outperform its homogenous counterpart in average by 59% in performance and 10% in energy, with EDP (Energy-Delay Product) improvements in almost every scenario.

Furthermore, this work also proposes and evaluates six schedulers for the heterogeneous system: two static algorithms, which allocate the threads on the first free core, where they will run during the entire execution; an Instruction Count (IC) Driven scheduler, which reallocates threads during synchronization points accordingly to their instruction count; a Feedback scheduler, which uses data from inside the reconfigurable unit to reallocate threads; the PC-Feedback scheduler, that adds a reuse mechanism to the last one; and an Oracle scheduler, which is capable of deciding the best thread allocation possible. We show that the static algorithm can reach high performance in applications with high parallelism, however for uniform performance in all applications, the Feedback and PC-Feedback algorithms are better designated.

Keywords: heterogeneity; multicore; scheduling; reconfigurable architectures; embedded systems

LIST OF FIGURES

Figure 1.1 Measured CPU and SoC power savings on a big.LITTLE MPSoC (Cortex-A15 and Cortex-A7) relative to a Cortex-A15 MPSoC. 16
Figure 2.1: Basic steps followed by a reconfigurable system. 19
Figure 2.2 Different types of reconfigurable unit coupling to a general propose processor. RFU: Reconfigurable Functional Unit 22
Figure 3.1 The big.LITTLE organization. The A15 (big) and A7 (LITTLE) cores are used accordingly to the necessities of the application, while the Mali GPU is dedicated for graphic processing. 26
Figure 3.2 big.LITTLE operating modes. (a) Cluster migration: either all the A15 cores running or all A7. (b) CPU migration: for each couple of A15 and A7 cores, one is executing and the other is off. (c) Global task scheduling: All the cores can be used simultaneously and independently. 27
Figure 4.1 HARTMP and CReAMS NoC connection. The cores communicate through the nodes of the network. 29
Figure 4.2: Instruction flow during execution of a non-stored basic block. Instructions are executed inside the GPP pipeline, while the binary translator checks for data dependencies and builds a configuration for the reconfigurable array. 30
Figure 4.3: Instruction flow during execution of a stored basic block. The GPP pipeline is stalled, the configuration is loaded into the reconfigurable unit and the basic block is executed there. 31
Figure 4.4 DAP: A GPP tightly coupled to a reconfigurable logic and a binary translator. 32
Figure 4.5 Interconnection mechanism. The input muxes indicate the source operand for each functional unit, while the output muxes decides the source of the output for each functional unit. 33
Figure 4.6 (a) Example assembly code running on a DAP system. (b) Filling of the write bitmap and functional units on the reconfigurable datapath for the given assembly code using speculation. 36
Figure 4.7 (a) Heterogeneous threads running on big cores. (b) Many functional units are wasted when running threads with small basic blocks. 37
Figure 4.8: (a) Heterogeneous threads running on small cores. (b) Threads of bigger basic blocks have to be split. 38
Figure 4.9: Heterogeneous threads running on heterogeneous cores. Load balance can be reached. 38
Figure 4.10 Reconfigurable unit of a DAP. Blocks consist of Input/Output Context, Functional Units and Reconfiguration Memory. 39
Figure 4.11: Increasing the Input/Output Context of the reconfigurable unit. More registers are available for feeding the functional units and for register renaming. 40
Figure 4.12: Area increase due to Input/Output Context increase. 41
Figure 4.13: Changing the number of functional units. More ILP can be exploited both from bigger basic blocks and from higher levels of speculation. 41
Figure 4.14: Area increase due to functional units. 42

Figure 4.15: Changing the reconfiguration memory size. More configurations can be stored, resulting in less configuration misses due to replacement policy. 43
Figure 4.16: Area increase due to reconfiguration memory. 43
Figure 4.17: Dynamic power increase due to reconfiguration memory. Memory context impacts greatly on energy consumption. 44
Figure 4.18: A good configuration must consider a balance between the different parameters of the reconfigurable unit to avoid system bottlenecks. 44
Figure 5.1: Steps for the Oracle allocation algorithm. 46
Figure 5.2: Pseudo code of the static scheduler. 48
Figure 5.3: Pseudo code of the inverse static scheduler 49
Figure 5.4: Pseudo code for the IC-Driven Scheduler. 50
Figure 5.5: Pseudo code for the Feedback scheduler. 51
Figure 5.6: Behavior of the Feedback and the PC-Feedback schedulers compared while running a code trace. The red stripes represent always the same barrier (same PC), while the blue stripes represent a second barrier. 53
Figure 5.7: Pseudo code for PC-Feedback scheduler. 54
Figure 6.1: Virtutech's Simics generates a trace file with the instructions of all the threads. This main trace is split to create individual trace files for each thread, which are used to feed the inputs of the DAP simulator. One instance of a DAP simulator is created for each trace file (thread). 57
Figure 6.2: Operation of the backwards script for performance. At each barrier, the core that took most number of cycles is accounted, as the others need to wait for the slowest core to reach the synchronization point. 58
Figure 6.3: Application trace generated by Simics. It contains all the instructions of all the executed threads. 59
Figure 6.4: Main trace generated by Simics is split in many other files, one for each thread 60
Figure 6.5: Behavior of the scheduling script when using the IC-Driven Algorithm. After the synchronization point, the Thread 1 is written on the input file of Core 2 and Thread 2 on the input of Core 1. 61
Figure 6.6: Oracle methodology example for a 4-threaded application. Each core size has to run all the threads, and then the Oracle algorithm uses their results to find the best combination of allocation for performance 62
Figure 6.7: Example of a DAP for the Ho1 configuration. 65
Figure 6.8: Illustration of the DAP Ratio between the He1 configuration and the three homogeneous configurations. 67
Figure 7.1: Performance comparison between He1 and Ho1. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS. Unbalanced

applications have advantages in a heterogeneous environment, while the balanced ones take advantage of the extra cores. 71
Figure 7.2: Performance comparison between He1 and Ho2. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS. The heterogeneous environment is better for applications with some inbalance in the threads. However, in a perfect balanced application, the small cores of He1 configuration holds the performance of the larger, thus being the homogeneous version (composed of medium cores only) better. 73
Figure 7.3: Performance comparison between He1 and Ho3. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS. The He1 configuration can efficiently exploit the ILP from threads while exploiting more TLP as well. 74
Figure 7.4: Energy comparison between He1 and Ho1. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS. The homogeneous version has double the number of GPPs and L1 caches, generating an overhead in energy consumption, even when the homogeneous version has better performance. 76
Figure 7.5: Energy comparison between He1 and Ho2. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS. The larger cores of the heterogeneous critically compromises the energy consumption of the system. 77
Figure 7.6: Energy comparison between He1 and Ho3. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS. The larger caches in the Ho3 configuration critically compromises the energy consumption of the system. 78
Figure 7.7: EDP comparison between He1 and Ho1. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS. Due to small energy consumption changes, the EDP is a reflection of the performance gains. 79
Figure 7.8: EDP comparison between He1 and Ho2. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS. This scenario shows that although the energy consumption is much higher in the heterogeneous version, the performance gains are enough to compensate it. 80
Figure 7.9: EDP comparison between He1 and Ho3. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS. 81
Figure 7.10: Performance comparison between He2 and Ho1. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS. 82
Figure 7.11: Performance comparison between He2 and Ho2. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS. 83
Figure 7.12: Performance comparison between He2 and Ho3. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS. 84
Figure 7.13: Energy comparison between He2 and Ho1. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS. 85

Figure 7.14: Energy comparison between He2 and Ho2. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS. 86
Figure 7.15: Energy comparison between He2 and Ho3. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS. 87
Figure 7.16: EDP comparison between He2 and Ho1. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS... 88
Figure 7.17: EDP comparison between He2 and Ho2. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS... 89
Figure 7.18: EDP comparison between He2 and Ho3. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS... 90
Figure 7.19: Performance comparison between the Oracle and the Static schedulers. The speedup is normalized to the Oracle's performance. The static scheduler can reach near Oracle performance on very unbalanced applications (due to most of the work being executed on a large core) and very balanced applications (as all cores execute the same load, no scheduling is necessary). 91
Figure 7.20: Performance comparison between the Oracle, the Static and the Inverse Static schedulers. The speedup is normalized to the Oracle's performance. The inverse statics shows the impact of allocating the thread 0 into a small core.. 93
Figure 7.21: Performance comparison between the Oracle, the Static and the IC-Driven schedulers. The speedup is normalized to the Oracle's performance. The IC-Driven is a first real scheduling attempt and showed the importance of using a good metric for allocation... 94
Figure 7.22: Performance comparison between the Oracle, the Static the IC-Driven and the Feedback schedulers. The speedup is normalized to the Oracle's performance. The Feedback reaches near Oracle performance as the static do and has better overall performance in other applications............. 96
Figure 7.23: Performance comparison between the Oracle, Static, Feedback, and PC-Feedback schedulers. The speedup is normalized to the Oracle's performance. The Feedback-PC shows better performance in some applications, however the improvement over the Feedback is only marginal.... 97
Figure 9.1: Examples of shared reconfigurable units. (a) Every core has its own reconfigurable datapath. (b) Multiple reconfigurable fabrics are shared between many cores. (c) One reconfigurable logic is shared between many cores. 102
Figure 9.2: SMT in a shared reconfigurable logic. Each core assigns its threads to execute together on the datapath 103
Figure 9.3: Cores dynamically request private spaces in a shared reconfigurable datapath. 104

LIST OF TABLES

Table I: The configuration of the reconfigurable unit on the CReAMS systems................................. 64
Table II: The configuration of the reconfigurable unit on the HARTMP systems............................... 66
Table III: DAP Ratio between HARTMP and CReAMS configurations... 67
Table IV: List of benchmarks used in this work with their expected ILP and TLP exploitation. The more '+' signals a benchmark has, the more it can exploit the given parallelism, while the more '-' the benchmark has, the less it can exploit. 69
Table V: Execution time (in milliseconds) of each application in each configuration. The arrows indicate the area parities between configurations. For instance, arrow 1 shows that 4He1 (a 4-core He1 configuration) has approximately the same area as 2Ho3, 4Ho2 and 8Ho1. 109
Table VI: Energy consumption (in Joules) of each application in each configuration. The arrows indicate the area parities between configurations. For instance, arrow 1 shows that 4He1 (a 4-core He1 configuration) has approximately the same area as 2Ho3, 4Ho2 and 8Ho1. 109
Table VII: Energy-Delay Product (Joules x Seconds) of each application in each configuration. The arrows indicate the area parities between configurations. For instance, arrow 1 shows that 4He1 (a 4-core He1 configuration) has approximately the same area as 2Ho3, 4Ho2 and 8Ho1....................... 110
Table VIII: Power consumption of each of the components of the reconfigurable array 111
Table IX: Power consumption for each access in the reconfiguration memory of the homogeneous configurations.. 112
Table X: Power consumption for each access in the reconfiguration memory of the different DAPs in the heterogeneous He1 configurations. 112
Table XI: Power consumption for each access in the reconfiguration memory of the different DAPs in the heterogeneous He2 configurations. 113
Table XII: Area ratio for components inside the He1 DAPs. 114
Table XIII: Area ratio for components inside the He2 DAPs. 114
Table XIV: Area ratio for components inside the Ho DAPs.. 114

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic and Logic Unit
API	Application Programming Interface
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuits
ASIP	Application-Specific Instruction Set Processor
CPU	Central Processing Unit
CReAMS	Custom Reconfigurable Arrays for Multiprocessor Systems
DAP	Dynamic Adaptive Processor
DDH	Dynamic Detection Hardware
DSP	Digital Signal Processor
DV	Dependence Verification
DVFS	Dynamic Voltage and Frequency Scaling
EDP	Energy-Delay Product
FIFO	First-In-First-Out
GPP	General Propose Processor
GPU	Graphics Processing Units
HARTMP	Heterogeneous Arrays for Reconfigurable and Transparent Multicore Processing
I/O	Input/Output
IC	Instruction Count
ID	Instruction Decode
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture

L2	Level 2
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
OS	Operating System
PC	Program Counter
RA	Resource Allocation
RAW	Read After Write
RISC	Reduced Instruction Set Computing
RU	Reconfigurable Unit
SM	Shared Memory
SoC	System-on-Chip
SRAM	Static Random Access Memory
TLP	Thread Level Parallelism
UT	Update Tables
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLIW	Very Large Instruction Word

TABLE OF CONTENTS

RESUMO **3**
ABSTRACT **4**
LIST OF FIGURES **5**
LIST OF TABLES **9**
LIST OF ABBREVIATIONS AND ACRONYMS **10**
1 INTRODUCTION **14**
1.1 Contributions **16**
1.2 Work Structure **17**
2 RECONFIGURABLE ARCHITECTURES **18**
2.1 Basic Principles **18**
2.2 Classification **20**
2.2.1 Code Analysis 20
2.2.2 Coupling 21
2.2.3 Granularity 21
3 RELATED WORK **23**
3.1 Multithreaded Reconfigurable Systems **23**
3.2 Heterogeneous Systems **25**
3.3 Scheduling Algorithms for Heterogeneous Systems **28**
4 HARTMP **29**
4.1 An Overview **29**
4.2 DAP - Dynamic Adaptive Processors **31**
4.2.1 Block 1 – Reconfigurable datapath 31
4.2.2 Block 2 – Processor pipeline 33
4.2.3 Block 3 – Storage components 34
4.2.4 Block 4 – Dynamic detection hardware 34
4.3 Heterogeneity on HARTMP **36**
4.4 Leveraging Heterogeneity with the DAP **39**
4.4.1 Changing the Input/Output Context 39
4.4.2 Changing the amount of functional units 41
4.4.3 Changing the Reconfiguration Memory 42
4.4.4 Finding the best reconfigurable unit configuration 44
5 SCHEDULING ALGORITHMS **45**
5.1 The Oracle Scheduler **46**
5.2 Static Scheduler **47**
5.3 Inverse Static Scheduler **48**
5.4 IC-Driven Scheduler **49**
5.5 Feedback Scheduler **50**
5.6 PC-Feedback Scheduler **51**
6 METHODOLOGY **55**
6.1 Simulators **55**
6.1.1 Simics Simulator 55
6.1.2 DAP Simulator 56
6.2 Scheduling **58**
6.2.1 Scheduling scripts 58
6.2.2 The Oracle Methodology 61
6.3 Third party tools **62**
6.3.1 Synopsys Design Compiler 62

6.3.2 CACTI-P 63
6.4 HARTMP and CReAMS configurations 64
6.4.1 CReAMS Configurations 64
6.4.2 HARTMP Configurations 65
6.4.3 Area parity between HARTMP and CReAMS configurations 66
6.5 Benchmarks 68
7 RESULTS 70
7.1 HARTMP vs CReAMS 70
7.1.1 He1 Configuration 71
7.1.2 He2 Configurations 82
7.2 HARTMP Schedulers 90
7.2.1 The Static Scheduler 91
7.2.2 The Inverse Static Scheduler 93
7.2.3 The IC-Driven Scheduler 94
7.2.4 The Feedback Scheduler 95
7.2.5 The PC-Feedback Scheduler 96
8 CONCLUSIONS 98
8.1 HARTMP usage 98
8.2 Schedulers 99
9 FUTURE WORK 101
9.1 A Big Little HARTMP 101
9.2 Sharing the reconfigurable unit 101
9.3 Exploiting new Schedulers 105
REFERENCES 106
APPENDIX A 109
APPENDIX B 111
APPENDIX C 114
APPENDIX D 115
Introdução 115
HARTMP 116
DAP 116
Escalonadores 120
Metodologia 122
Resultados 123
Conclusão 126

1 INTRODUCTION

Current embedded systems have become popular with the dissemination of smartphones, wearables and many other smart gadgets. Such systems execute many kinds of applications: from web browsers to video decoders, which cover a wide range of functionalities and result in a heterogeneous environment. To deal with this matter, multicore systems became very common on embedded environments. Having multiple cores allows the system to exploit the thread level parallelism (TLP) of applications. Furthermore, most multicore systems are homogeneous in architecture and organization and are usually comprised of superscalar processors (HENESSY; DAVID A. PATTERSON, 2011), which allows instruction level parallelism (ILP) to be exploited as well.

However, homogeneous multicore systems are inefficient when computing applications with heterogeneous behavior. These systems have cores capable of exploiting the same levels of instruction parallelism each, which might result in resource wasting on the wide range of workloads of an application. To accelerate execution of specific applications, Application Specific Integrated Circuits (ASICs) are commonly used in embedded systems, as well as Digital Signal Processors (DSPs) and Graphics Processing Units (GPUs). The integration of such specialized units has lead the embedded systems to the System on Chip (SoC) era, creating a highly heterogeneous and energy efficient processing environment(WOLF; JERRAYA; MARTIN, 2008).

On the other hand, the SoC environment is not only heterogeneous in organization, but also in architecture: each processing unit has its own Instruction Set Architecture (ISA). The ISA defines the set of instructions a processor is capable of decoding and executing. Having a different instruction set for each processing element means that each core in the SoC has a distinct interface between software and hardware. In other words, there is no compatibility between the software binary code for each core. This imposes a great drawback for such systems, as they become highly dependent on specialized tools or greatly increase the developers’ efforts to create software capable of efficiently use the entire system. Thus, the software for each new generation of SoCs that introduces new ISAs must be rewritten, greatly increasing the time-to-market of an application.

To reduce the complexity of developing embedded software, but still keep the advantages of having an efficient environment, heterogeneous multicore processors of homogeneous ISA

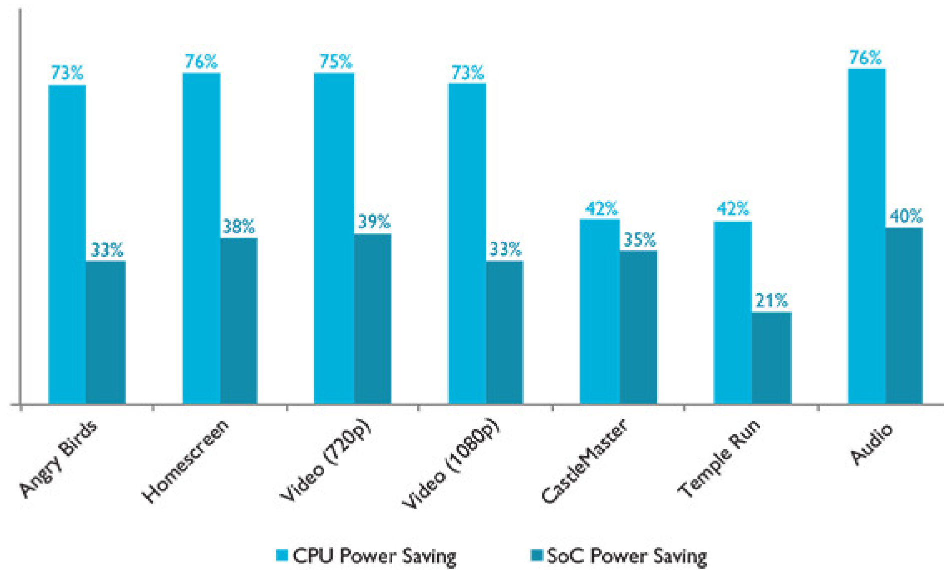
were proposed (KUMAR et al., 2003). An example already available in the industry is the big.LITTLE technology from ARM ("big.LITTLE Technology", 2016). This system is composed of two distinct superscalar processors of same ISA, capable of exploiting different levels of instruction parallelism and with different power requirements each. An online scheduler is responsible for switching between cores according to the applications demands, greatly reducing energy consumption when applications enter a low demand stage, as show in Figure 1.1. The main advantage of this approach is the capability of keeping software compatibility: as both processors use the same ISA, no software intervention is necessary (no recompilation or redistribution) to execute in the new system.

Nevertheless, superscalar processors are not designed to be energy efficient. These processors rely on exhaustive dependencies check on an instruction window to determine which operations can be executed in parallel. At every dispatch cycle, the processor has to evaluate the dependencies, even if the same basic block is executed repeatedly. According to (OLUKOTUN; HAMMOND, 2005), the complexity of the additional logic for the dependency check hardware is approximately proportional to the square of the number of instructions that can be executed in parallel in a superscalar processor. This additional logic creates huge area and power overheads for a marginal increase in performance, which suggests a scalability problem.

Therefore, an alternative to superscalar processors are reconfigurable organizations. These systems can adapt themselves to the application at hand, reconfiguring their datapaths to improve execution. Some of these organizations can be transparent to the programmer, keeping code compatibility, as in (RUTZIG; BECK; CARRO, 2013). Implementation strategies are adopted to optimize data dependency and maximize the ILP exploitation, providing huge performance improvements and energy saving over classic processors, as shown in (BECK et al., 2008). By replicating reconfigurable cores, it is also possible to build multicore systems, exploiting both TLP and ILP (the latter more efficiently).

Moreover, reconfigurable organizations are extremely regular: they have a set of basic cells (usually implemented in combinational logic) that is repeated many times. Thus, the number of processing elements in these organizations can be easily changed for each core, creating a highly heterogeneous environment. The same cannot be accomplished in superscalar processors like those that ARM uses: as to reach heterogeneity, they change the number of issues or the execution mode (in-order or out-of-order). These changes represent a huge adaptation in the circuit of a superscalar processor.

Figure 1.1 Measured CPU and SoC power savings on a big.LITTLE MPSoC (Cortex-A15 and Cortex-A7) relative to a Cortex-A15 MPSoC.



Source: (“big.LITTLE Technology”, 2016)

Although creating heterogeneous cores capable of exploiting different levels of instruction parallelism is easier on reconfigurable systems than on the superscalar, allocating the threads to use them efficiently is still a challenge. Differently from traditional homogeneous systems – in which the schedulers were not designed to expect different ILP in cores, but only voltage and frequency scaling – in heterogeneous processors the scheduler must allocate the threads accordingly to their necessities of resource usage. Furthermore, the heterogeneity in reconfigurable systems lies on the reconfigurable logic of the core, thus the scheduler must be aware of the potential of acceleration of each thread instead of its general ILP. This means that traditional scheduling policies must be adapted to address the allocation problem on these systems.

1.1 Contributions

This work exploits the many aspects, advantages and challenges of reconfigurable heterogeneous systems. Firstly, we show how it is possible to extend an originally homogeneous reconfigurable system to be heterogeneous. For this, we use the previously proposed Custom Reconfigurable Arrays for Multiprocessor Systems (CReAMS) (RUTZIG;

BECK; CARRO, 2013), which is a homogeneous multicore system based on a reconfigurable array of processing units in each core. We exploit the regularity of the reconfigurable array to create cores of distinct sizes and to create a new heterogeneous system called HARTMP (Heterogeneous Arrays for Reconfigurable and Transparent Multicore Processing).

Then, we show the advantages of using HARTMP over CReAMS: for several applications, the heterogeneous environment can have better performance (in execution time) and/or better energy consumption. Furthermore, we also evaluate the tradeoff between performance and energy consumption through the EDP (Energy-Delay Product) metric. The EDP is capable of showing when an eventual performance loss is compensated by higher energy savings, and vice versa. We demonstrate that in many cases, the HARTMP system is better in the EDP metric than its homogeneous counterpart is.

Furthermore, as mentioned earlier, the scheduler is vital for the efficient execution in a heterogeneous environment. We propose five scheduling algorithms and compare them with a predictive scheduler (the Oracle) to assert their performance. We show that simple scheduling algorithms might be used for some applications without a considerable performance loss, while algorithms that are more complex have a general better performance.

1.2 Work Structure

This dissertation is organized as follows: In chapter 2, we present for the reader the basis of reconfigurable architectures and their classification. Then in chapter 3, we discus some of the main previous works in the literature. Chapter 4 introduces HARTMP organization and explains how it works. In chapter 5, we present the six scheduling algorithms proposed for operating in HARTMP, while in chapter 6 we show the methodology for achieving the results, which are discussed in chapter 7. Chapter 8 draws the conclusions for this work and chapter 9 lists some of the possible future researches that can be done over the HARTMP organization.

2 RECONFIGURABLE ARCHITECTURES

The architectures that can adapt themselves to provide a hardware expertise for a specific application are known as reconfigurable architectures. Because of this specialization, these systems are expected to provide performance and energy saving improvements. On the other hand, reconfigurable architectures are still expected to be flexible: they must accelerate a large range of applications. Consequently, they have smaller gains if compared to dedicated circuits, like Application-Specific Instruction Set Processors (ASIPs) and Application-Specific Integrated Circuits (ASICs) (BECK; LANG LISBÔA; CARRO, 2013). On the next sections we will discuss the basic principles of reconfigurable architectures, how they are usually implemented and the many possible classifications for the different kinds of reconfigurability on systems.

2.1 Basic Principles

Reconfigurable systems are usually composed of a reconfigurable logic, a controller – to control the reconfiguration and the communications – and a context memory – to store the configurations for the reconfigurable fabric. These blocks are usually coupled to a General Propose Processor (GPP), which executes the software regions that cannot be accelerated, while the reconfigurable fabric processes the other regions (called kernels). Consequently, the more and larger are the kernels on the application, the better is the efficiency of the reconfigurable system. On the other hand, one must keep in mind the area constraints for the circuitry, as the extra blocks required for reconfigurability might greatly increase the size of the whole system design.

Reconfigurable systems usually implement the following six steps, also illustrated in the Figure 2.1:

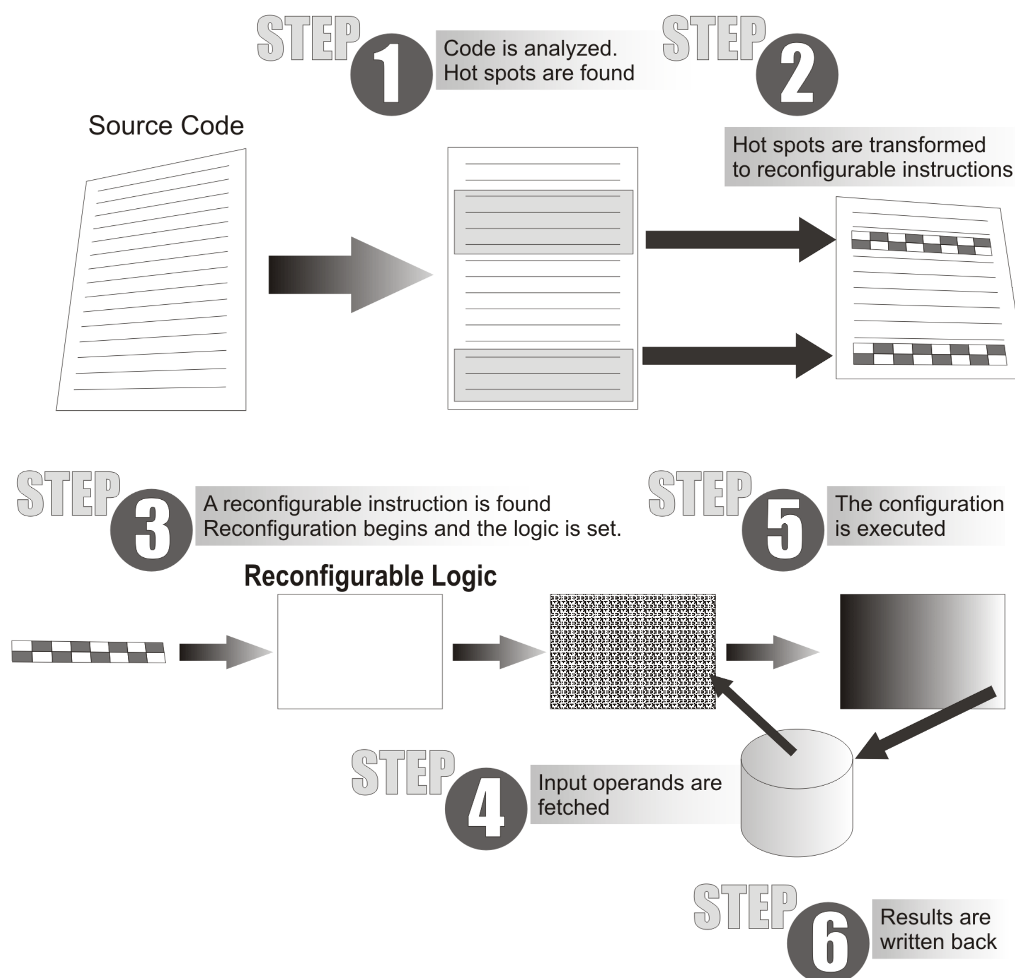
1) *Code Analysis*: The first step consists of analyzing the code to identify kernels (regions of code that are mostly executed) that can be transformed for execution in the reconfigurable logic. The objective is to find the best tradeoff between performance gains and the available resources on the reconfigurable unit. This analysis is usually made dynamically on a previously executed trace of the

application, as it is hard (and sometimes even impossible) to determine the kernels by only analyzing the software code. This step can be done by automated tools or even manually by the system designer.

2) *Code transformation:* After the identification of the kernels, the code instructions of these regions (originally from the ISA of the GPP) must the replaced by reconfigurable instructions. The new instructions are handled by the control unit of the reconfigurable system, guiding the reconfiguration process.

3) *Reconfiguration:* When a new reconfiguration instruction is ready to be executed, the programmable units on the reconfigurable logic are reorganized in order to perform the given instruction. Firstly, a set of configuration bits, called configuration context, are loaded from a special memory: the reconfiguration memory. The time required for the memory to load and the configuration of the

Figure 2.1: Basic steps followed by a reconfigurable system.



Source: (BECK; LANG LISBÔA; CARRO, 2013)

reconfigurable logic is called reconfiguration time. Both the reconfiguration time and the context memory compose the total reconfiguration overhead (in performance and area).

4) *Input Context Loading:* To execute the given reconfiguration instruction, a set of inputs is necessary. These data can be loaded from a register file, a shared memory or even by a message passing protocol.

5) *Execution:* Once the units on the reconfigurable logic are configured and the input operands are ready, the actual execution of the instruction begins. This execution is more efficient than an execution on a GPP processor, because now there is a specialized circuitry ready to process the operation.

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 until the end of the application execution.

2.2 Classification

A reconfigurable system can have many classifications, as shown by Beck; Lisbôa; Carro (2013). Here we will discuss the ones that are most important for this work.

2.2.1 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 advantage of using the trace method is the availability of the dynamic information of the execution. 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. By having a trace of the execution, the designer can use tools that dynamically analyze it and indicate which are the kernels of the application that are mostly executed.

After identification of the kernels, the instructions must be replaced with reconfigurable instructions. It is possible to replace them manually in the assembly code or in the source code. In the case of the latter, code annotation can also be employed. For instance, macros can be used in the source code to indicate a region that will be replaced by reconfiguration instructions,

leaving the task of generating the modified code to the assembler. Another possibility is to fully automate the process, using a tool capable of identifying the kernels and handling the issues of communication, instruction translation, reconfiguration overhead, execution and write back of the results. Although the fully automated solution is more transparent and leaves minimum effort to the software developer, it is highly dependent on the reconfigurable system in use.

2.2.2 Coupling

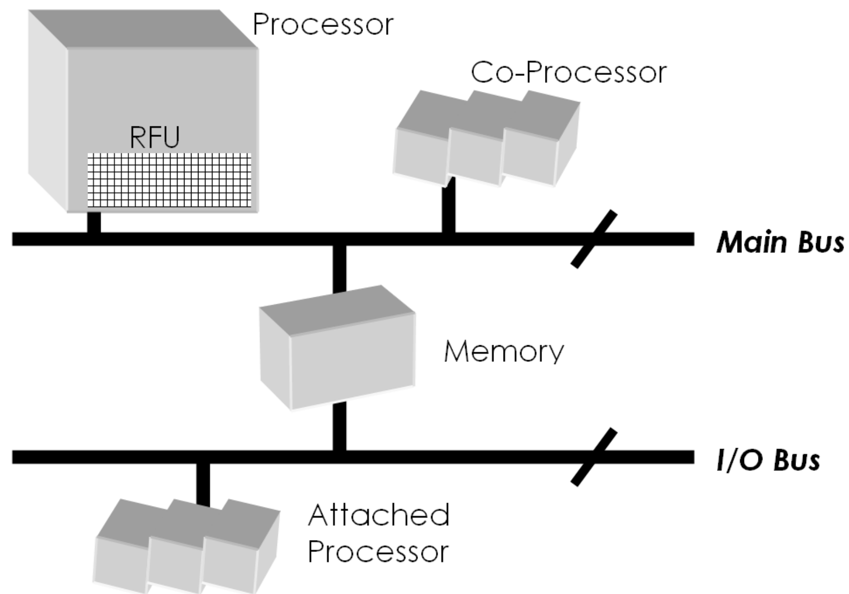
How the reconfigurable unit is coupled to the GPP defines the way in which data transmission and synchronization between the both is done. The position where the reconfigurable unit is placed directly affects the performance of the whole system, as communication costs and data transfer time are important factors to be considered. There are three main ways the Reconfigurable Unit (RU) can be connected to the main processor, which are illustrated in the Figure 2.2 and described as follows:

1. **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.
2. **Coprocessor:** The RU is placed next to the GPP. The communication is usually done using a protocol similar to those used by coprocessors.
3. **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.

2.2.3 Granularity

The granularity defines the level of data manipulation of the reconfigurable unit. For fine-grained logic, the smallest blocks that can be configured are usually gates. Like LUTs of FPGAs, they are more 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.

Figure 2.2 Different types of reconfigurable unit coupling to a general propose processor. RFU: Reconfigurable Functional Unit



Fine-grained systems provide high flexibility and in theory can implement any digital circuit; however, they require larger configuration contexts as more bits are required to configure the gates. Coarse-grained systems, on the other hand, have smaller context, as the number of processing elements needed to be configured is much lower.

3 RELATED WORK

In this chapter, we will show and discuss some of the previous works developed on reconfigurability, heterogeneity and thread scheduling. There are many works on single- and multi- processing reconfigurable systems, however, these works do not exploit heterogeneity in their organizations, which is the gap this work is mostly interested to fill. The next sections will present several works of multithreaded reconfigurable systems, discussing their advantages and disadvantages and comparing them with our solution: HARTMP. Then, we show some works on heterogeneous systems and the challenges they face. Finally, works on thread scheduling, which is essential for heterogeneous systems, are discussed.

3.1 Multithreaded Reconfigurable Systems

There are many proposed reconfigurable architectures in the literature. One can find different environments that apply some kind of adaptability to improve the performance of applications (BECK; CARRO, 2010; BECK; LANG LISBÔA; CARRO, 2013; COMPTON; HAUCK, 2002). Considering only reconfigurable architectures built upon multicore systems, Watkins (WATKINS; ALBONESI, 2010) 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 (LEE et al., 2010). It extends the Warp Processing (LYSECKY; STITT; VAHID, 2006) system to support multiple threads executions. In this case, one processor is entirely dedicated to execute the operating system tasks needed to synchronize threads and to schedule their kernels in the accelerators. KAHRISMA (BINGFENG MEI et al., 2005) 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.

CReAMS (RUTZIG; BECK; CARRO, 2013) (Custom Reconfigurable Arrays for Multiprocessor Systems) couples a homogeneous reconfigurable matrix of functional units with a main processor to form each core that comprises the system. CReAMS maintains binary compatibility, like the superscalar multiprocessors do, but using a special translation circuit. This hardware translator transforms sequences of instructions in configurations for the reconfigurable logic, which can be stored and reused in the future, avoiding repetitive code analysis – which is one of the main drawbacks of superscalar processors.

As just discussed, reconfigurable multicore architectures can be both 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). HARTMP is homogeneous in architecture, allowing binary compatibility, and heterogeneous in organization, which gives flexibility for thread scheduling. Therefore, the advantages of using HARTMP for heterogeneous cores over the architectures reviewed above include:

- Unlike KAHRISMA and Thread Warping, HARTMP is homogeneous in architecture. It employs a binary translation system implemented in hardware that eases the software development process since a well-known toolchain (i.e. gcc) is used for any of its versions. Neither source code modifications nor additional libraries are necessary as the binary translator will handle the process of generating configurations.
- KAHRISMA and ReMAPP rely in special and particular tool chains to extract Thread-Level Parallelism and to prepare the platform for execution. HARTMP 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 using the same APIs used in typical systems.
- In contrast to ReMAPP and Thread Warping, HARTMP employs a coarse-grained reconfigurable fabric instead of a fine-grained one. Fine-grained architectures may 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 are capable of accelerating the entire application as they have reduced reconfiguration time.
- Even though CReAMS can cover most of the drawbacks discussed, it is a homogeneous organization: all cores have the same resources and can exploit the same levels of

instruction parallelism, which leads to inefficiency when it comes to execute applications with low or unbalanced TLP.

Heterogeneity on HARTMP is completely transparent to the programmer or the operating system. The instructions are evaluated and dispatched to the reconfigurable logic by the binary translator implemented on chip, so no additional modifications in code are necessary.

3.2 Heterogeneous Systems

An early work on heterogeneous systems (KUMAR et al., 2003) 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. However, it showed the great potential of heterogeneous systems for energy savings by using simpler cores when application demand is small instead of always using a complex and power hungry core.

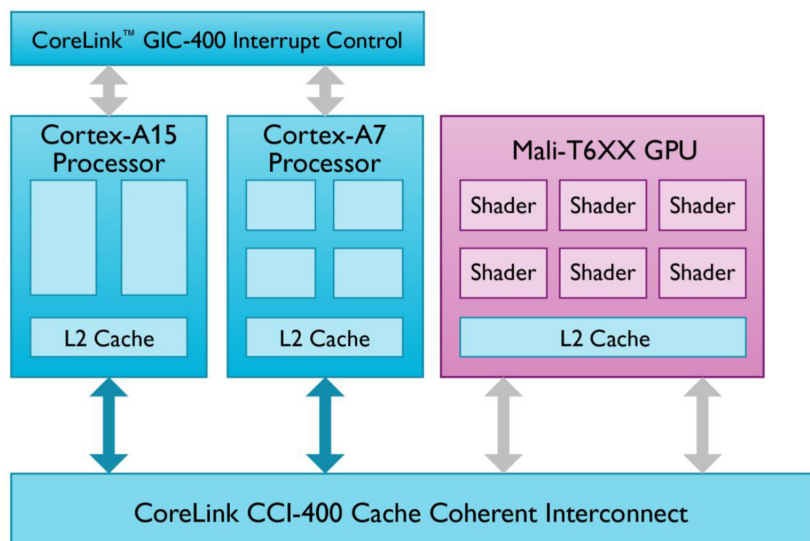
Heterogeneity can be exploited, also, at runtime, as proposed in (SRINIVASAN et al., 2013). In this work, the authors create an out-of-order multicore superscalar processor in which cores are able to polymorph themselves into in-order cores whenever the performance/Watt efficiency requirements change. Using this approach, many blocks 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 creates an overhead to turn the blocks on and off and, while the core is in in-order processor mode, leaves many resources turned off, wasting chip area.

One of the most common class of heterogeneous systems are the MPSoCs (Multiprocessor System-on-Chip) (WOLF; JERRAYA; MARTIN, 2008). SoCs (Systems-on-Chip) are chips with processors that include several accelerators in order to execute specific applications more efficiently. MPSoCs are SoCs with multiple cores and they have a great potential for energy savings, especially as they are commonly used for signal processing applications, which require a main processor as a master controller and other DSP accelerators to execute chunks of data over the same vector instructions. One of the first MPSoCs was the Lucent Daytona

(ACKLAND et al., 1999), designed for wireless base stations, it was composed of 4 SPARCV8 processors coupled to other DSP accelerators. These systems, however, have a great drawback when software productivity is considered: each type of accelerator and processor on the chip have a different ISA. This means that each accelerator has a different interface for data communication and execution, which brings two possible consequences: either the software developer will need to handle manually the usage of each accelerator, or specialized tools and compilers must be distributed by the vendor, which greatly increases the time-to-market of new systems. Furthermore, each MPSoC that introduces new accelerators with new ISAs must have its software updated, thus, code compatibility is not maintained.

A current approach to maintain software compatibility and still exploit the advantages of efficient application execution are the processors of heterogeneous organization and homogeneous architectures. By having the same ISA between the many processor cores, the system keeps the binary compatibility and can directly execute already distributed software, without the need of recompiling or rewriting the application. The big.LITTLE technology ("big.LITTLE Technology", 2016) from ARM is an example of such system that is already in use in many embedded processors and has proven to be very efficient. It combines the high performance Cortex-A15 and the energy efficient Cortex-A7 superscalar cores (as shown in Figure 3.1) to create a heterogeneous environment where the application can be executed accordingly to its needs.

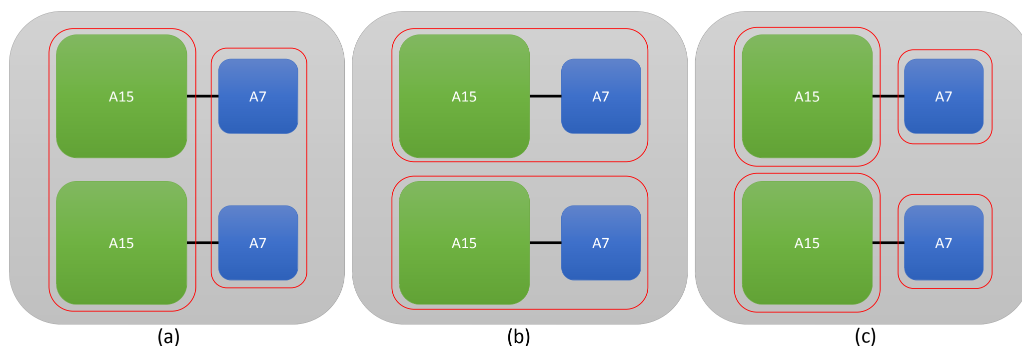
Figure 3.1 The big.LITTLE organization. The A15 (big) and A7 (LITTLE) cores are used accordingly to the necessities of the application, while the Mali GPU is dedicated for graphic processing.



There are three modes in which the big.LITTLE cores can operate (JEFF, 2013) and they are represented in Figure 3.2. In Figure 3.2(a), the big cores (A15) and the LITTLE cores (A7) are grouped in separated clusters and only one cluster can be operational at a time. This is the cluster migration mode. In Figure 3.2(b), called CPU migration, each cluster is formed of a couple of A15 and A7 cores. In this mode, one cluster might have the A15 core running, while in other the A7 might be active. A thread in a cluster can migrate between the big and the LITTLE cores, independently of other clusters, but only one core may be in use at a time. Finally, in Figure 3.2(c), the global task scheduling mode, the threads can be allocated in all the cores independently, creating a very flexible and heterogeneous environment. The cluster and CPU migration modes are controlled directly by the processor, using existing information on Dynamic Voltage and Frequency Scaling (DVFS) to determine the best cluster or CPU to run an application with. The global task scheduling, though, is much more complex and requires modifications on the operating system scheduler, which must be aware of the present hardware utilization and must use predictive algorithms to decide on which core a thread will optimally run.

The drawbacks of using the big.LITTLE technology can also be targeted in order to build new and more efficient processors. For instance, as previously discussed, superscalar processors (such as the ones used in big.LITTLE) were not designed for energy efficiency. It is possible to apply reconfigurable logic and build heterogeneous systems capable of efficiently exploiting many levels of ILP in heterogeneous applications. This is not viable in the

Figure 3.2 big.LITTLE operating modes. (a) Cluster migration: either all the A15 cores running or all A7. (b) CPU migration: for each couple of A15 and A7 cores, one is executing and the other is off. (c) Global task scheduling: All the cores can be used simultaneously and independently.



Source: the autor

big.LITTLE technology due to its already complex organization inherited from the superscalar model. Moreover, the technology is restricted to few possible designs (currently, only two superscalar models), while the reconfigurable fabric can be freely customized.

3.3 Scheduling Algorithms for Heterogeneous Systems

When considering unaware schedulers (when no details of the cores are necessary), Becchi; Crowley (2006) and Kumar et al. (2003) works are two of the most well-known in the literature. Both rely on online performance checks to determine the best thread allocation on a dual core (“fast” and “slow” cores) system. Becchi uses an IPC(Instruction Per Cycle)-driven algorithm to sample threads’ instructions per cycle (IPC) on the two types of cores. Threads with high IPC ratio have priority to be assigned to the fast core, as they can achieve higher speedups on it. Kumar’s work creates an oracle heuristic to optimize energy efficiency and energy-delay product in superscalar cores. Then, it compares the optimal results with a similar approach to Becchi’s. Differently from Becchi’s, Kumar’s work makes many performance samples of groups of threads in different core types. This allows for finding globally optimal allocations instead of making local decisions on thread swapping.

On the other hand, many other works for aware heterogeneous scheduling have also been proposed (BALAKRISHNAN et al., 2005; MOGUL et al., 2008; TEODORESCU; TORRELLAS, 2008). Shelepov; Alcaide proposes HASS (2009), a heterogeneous aware algorithm that creates architectural signatures of applications. The scheduler must know in advance the cache sizes and frequencies from the distinct cores. By profiling the application before execution, HASS identifies its memory-boundedness and use it to predict the best core to allocate the threads, considering expected cache misses and clock frequency. However, none of the above proposals have been applied in heterogeneous systems with reconfigurable logic.

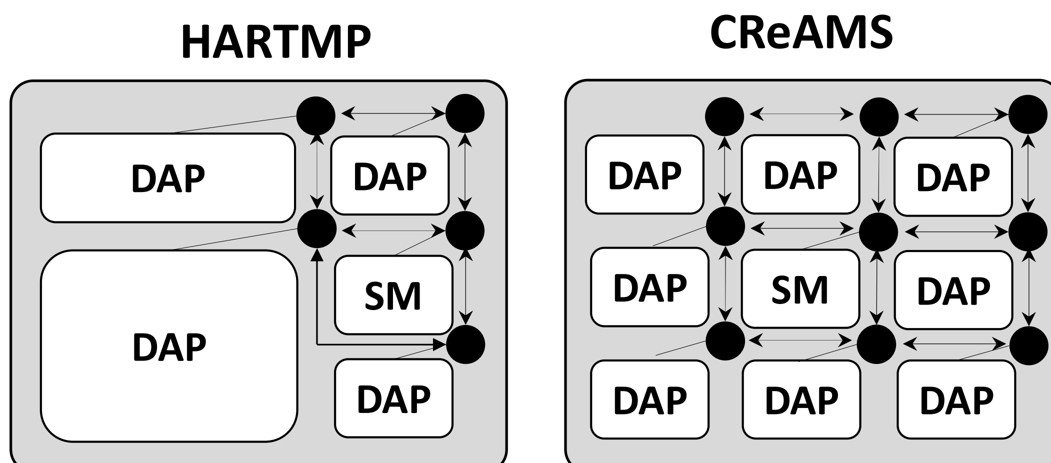
4 HARTMP

HARTMP (Heterogeneous Arrays for Reconfigurable and Transparent Multicore Processing) is a transparent and dynamic multicore organization based on reconfigurable arrays of functional units (the Dynamic Adaptive Processor – DAP) just like CReAMS (RUTZIG; BECK; CARRO, 2013) is. However, unlike CReAMS, on HARTMP each core of the system has a distinct sized array, thus being heterogeneous in organization. The communication among the DAPs is done through a 2D-mesh Network on Chip (NoC) using a XY routing strategy, as illustrated in the Figure 4.1. HARTMP also includes an on-chip unified L2 shared memory, illustrated as SM in the Figure 4.1. On the following sections, we will introduce an overview on the inner workings of the organization, followed by details on the reconfigurable logic, general propose processor and memories on HARTMP, as well as how one can reach heterogeneity in this system.

4.1 An Overview

HARTMP is an organization composed of both a general propose processor and a reconfigurable logic that use the same ISA. This is accomplished through a dynamic binary

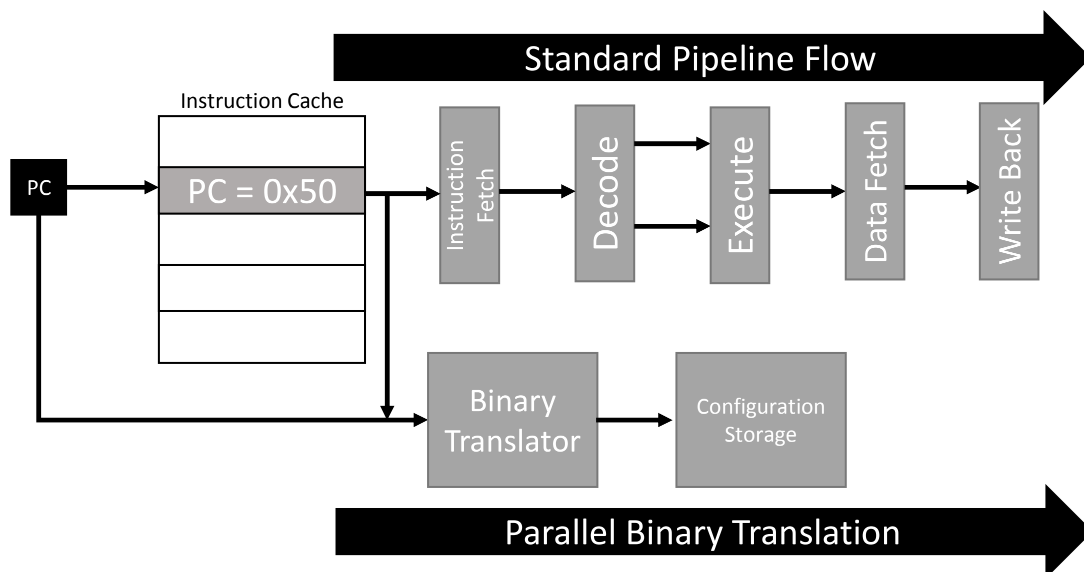
Figure 4.1 HARTMP and CReAMS NoC connection. The cores communicate through the nodes of the network.



Source: the author

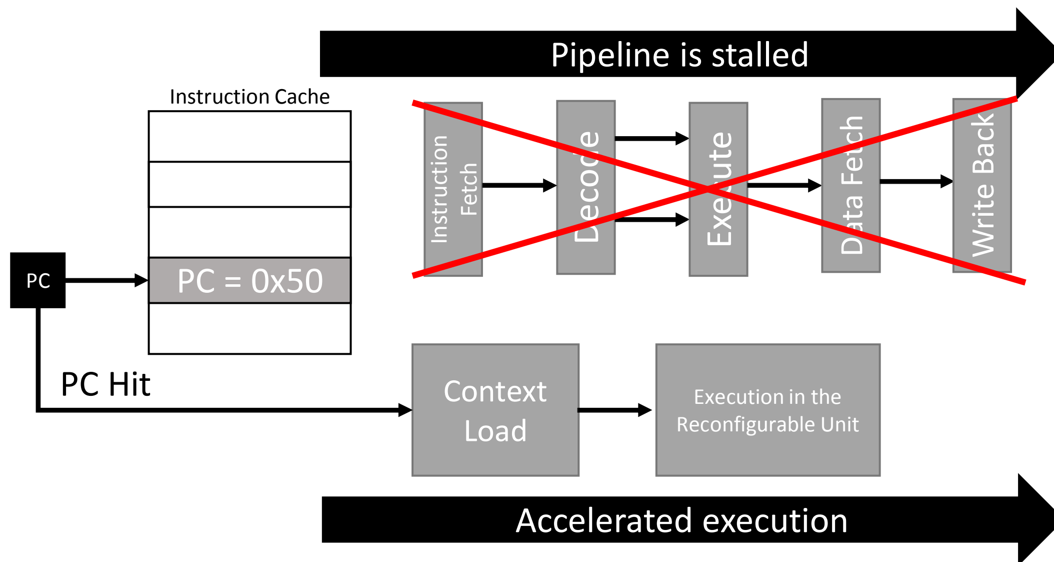
translator coupled directly in the hardware, which checks if a basic block running in the GPP can be executed in the reconfigurable logic. The first time a basic block is fetched, its instructions are executed by the GPP pipeline. However, at the same time, the instructions are also sent to the binary translator, which transforms the GPP instructions for execution on the reconfigurable unit. When an entire basic block is translated, it is saved as a configuration in a special memory for future use and indexed by the program counter (PC) of the basic block first instruction. This is illustrated in the Figure 4.2. The next time that basic block is executed, i.e., the next time the PC points to the first instruction of the basic block, the GPP is stalled, the configuration previously generated is loaded in the reconfigurable unit and the entire basic block executes there, as shown in the Figure 4.3. As we will see in the next section, the advantages of running the basic block in the reconfiguration unit include high ILP exploitation and storage of instruction dependency, avoiding repetitive checks.

Figure 4.2: Instruction flow during execution of a non-stored basic block. Instructions are executed inside the GPP pipeline, while the binary translator checks for data dependencies and builds a configuration for the reconfigurable array.



Source: the author

Figure 4.3: Instruction flow during execution of a stored basic block. The GPP pipeline is stalled, the configuration is loaded into the reconfigurable unit and the basic block is executed there.



Source: the author

4.2 DAP - Dynamic Adaptive Processors

The DAP is a reconfigurable architecture tightly coupled to the processor and was presented in (BECK et al., 2008). 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 4.4, and we explain them in the following subsections.

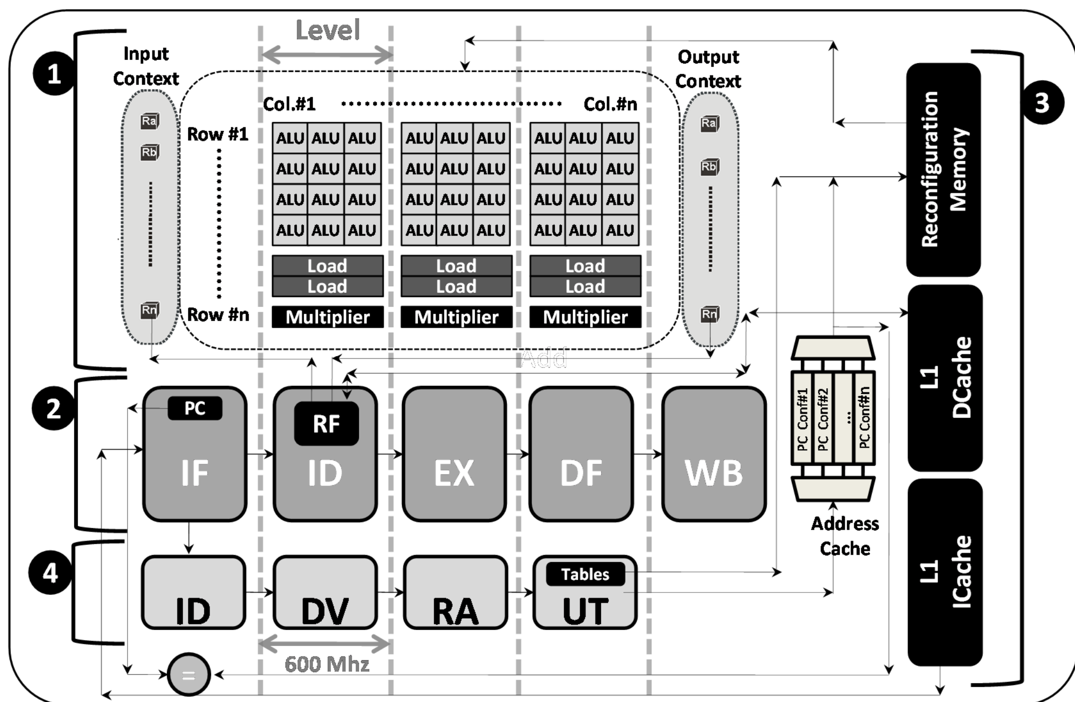
4.2.1 Block 1 – Reconfigurable datapath

Block 1 shows the structure for the reconfigurable datapath. 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 for communication and, consequently, saves power and reduces the reconfiguration time. As we can see in the Figure 4.4, the datapath is organized in a matrix structure separated by levels. Each level takes one processor cycle to execute and is composed of rows and columns. The number of rows is the maximum number of instructions that can be executed in parallel: independent instructions are allocated on the

same column. The number of columns dictates the maximum number of dependent instructions that can be executed in sequence in a level – the columns in one level are executed in sequence as a combinational block. For example, the configuration in Figure 4.4 performs up to four independent arithmetic and logic operations in parallel. 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 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 datapath 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 – the output context – 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 datapath level (a cycle of the processor) and if it will not be changed in the next levels, than this value can be written

Figure 4.4 DAP: A GPP tightly coupled to a reconfigurable logic and a binary translator.



Source: the author

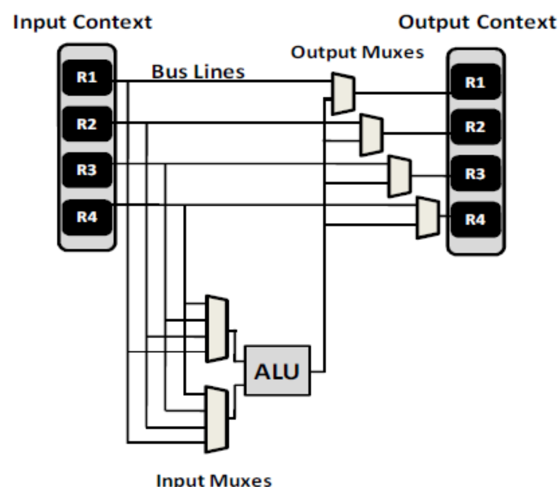
back on the next cycle. If the number of writes produced by the array is greater than the number of available write ports in the register file, than the excess instructions are forwarded to the next level. In the example shown in Figure 4.4, the maximum number of ports available is two.

Figure 4.5 shows a simplified overview of the reconfigurable datapath interconnection structure. 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 follow. 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 4.5, 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 memory and a set of such signals composes a configuration of the array.

4.2.2 Block 2 – Processor pipeline

Block 2 is the basic processor coupled to the array. In this work, the baseline processor is a SparcV8-based architecture running at 600MHz. Its five stage pipeline reflects a traditional single-issue 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). Thus, all the ILP exploitation comes from the reconfigurable datapath.

Figure 4.5 Interconnection mechanism. The input muxes indicate the source operand for each functional unit, while the output muxes decides the source of the output for each functional unit.



4.2.3 Block 3 – Storage components

There are two memory units specialized for the reconfiguration system: the address and reconfiguration memories. The first holds the memory address of the first instruction of every configuration built by the dynamic detection hardware (explained later). An address memory hit indicates that a configuration was found, therefore this memory 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 saved configuration. Each bit is a control bit for an output or input multiplexers or functional unit. They indicate which functional unit will be active and which register will be read as operators. Furthermore, the configuration also holds the operands and immediate addresses for the operations.

4.2.4 Block 4 – Dynamic detection hardware

The Dynamic Detection Hardware (DDH) is a binary translation mechanism that turns the instructions from SparcV8 ISA to reconfigurable array configurations. DDH is a four-stage pipelined circuit that runs in parallel to the GPP being out of the critical path of the system. Instruction Decode (ID) stage is responsible for decoding the operands in the base processor instruction to datapath code, while Dependence Verification (DV) checks if these operands have any dependency with the instructions already stored in the configuration being built. Resource Allocation (RA) stage uses the DV analysis to determine the optimal functional unit for the given operation inside the array. Finally, Update Tables (UT) stage saves the new allocation in the reconfiguration memory for future use. The translation process is performed as the GPP 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 the operating frequency unchanged (600MHz).

When an instruction is fetched from the instruction cache, the ID decodes it just as the decoder of the GPP would do. However, it does not fetches values from the registers, it just identifies the operands and operation from the instruction to forward them for the next pipeline stage.

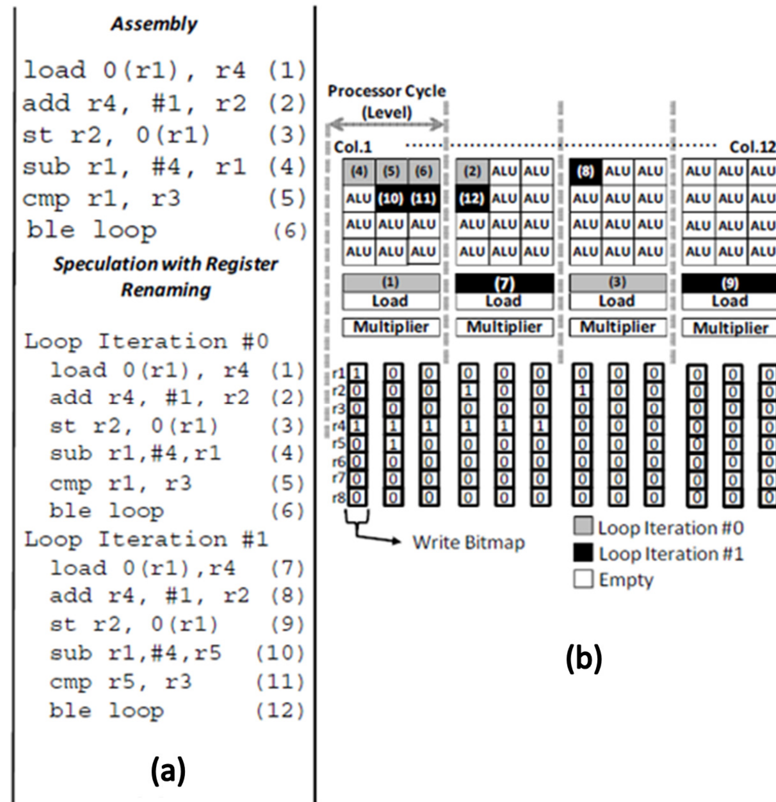
For each column of the reconfiguration datapath (Figure 4.4), there is a bitmap responsible for storing in the target operands of the already allocated instructions in the respective column. The set of bitmaps for all columns is named as Write Bitmap (Figure 4.6(b)). Thus, for each incoming instruction, its source operands will be compared to the target operands in this bitmap to decide in which column the instruction will be allocated, according to data dependencies. The DV is responsible for this process. Figure 4.6(a) has an assembly code as an example. In the Figure 4.6(b), the allocation of this code on the reconfigurable datapath 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, or in other words, to keep the dependency between instructions.

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 DV (write bitmap) and allocates the instruction number two at the later column of instruction number one. The allocation process is done by the RA component. 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. Every time an instruction translation finishes, the UT component updates the context of the owner basic block to include the new instruction.

The DDH supports speculation, so when a 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 4.6(a) represents the instructions allocated for the second iteration of the loop code of Figure 4.6(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. In (BECK; RUTZIG; CARRO, 2014) the authors present all the details on the algorithm of the reconfigurable unit used on the DAP.

Figure 4.6 (a) Example assembly code running on a DAP system. (b) Filling of the write bitmap and functional units on the reconfigurable datapath for the given assembly code using speculation.

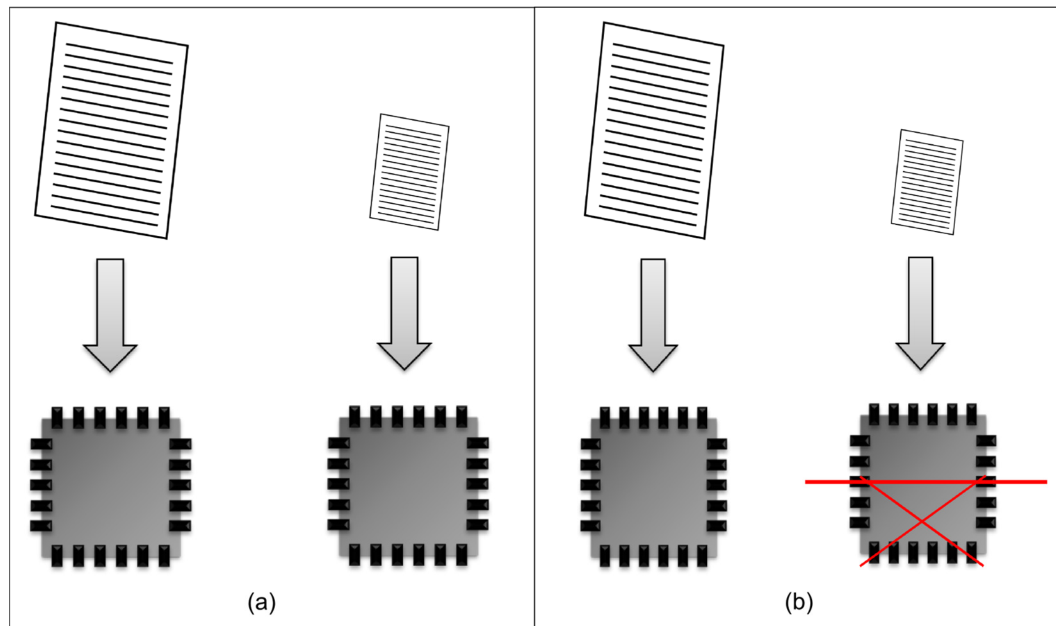


Source: (RUTZIG; BECK; CARRO, 2013)

4.3 Heterogeneity on HARTMP

On a HARTMP configuration, each DAP has a different number of functional units, input and output context length and memory size. Each of these components and the impact on changing their sizes will be discussed individually later. The variation of these components sizes allows 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. One should keep in mind that, differently from traditional heterogeneous processors of single-ISA, HARTMP leverages ILP by accelerating entire basic blocks, thus the level of instruction parallelism exploited is given by the resources in the reconfigurable unit. Superscalar processors, like ARM's big.LITTLE, exploit ILP by comparing dependencies on a window of instructions, being able to find parallelism in every region of code, at the cost of extremely complex and power hungry hardware. The ILP exploitation on superscalar processors is given by the size of the instruction window and the

Figure 4.7 (a) Heterogeneous threads running on big cores. (b) Many functional units are wasted when running threads with small basic blocks.

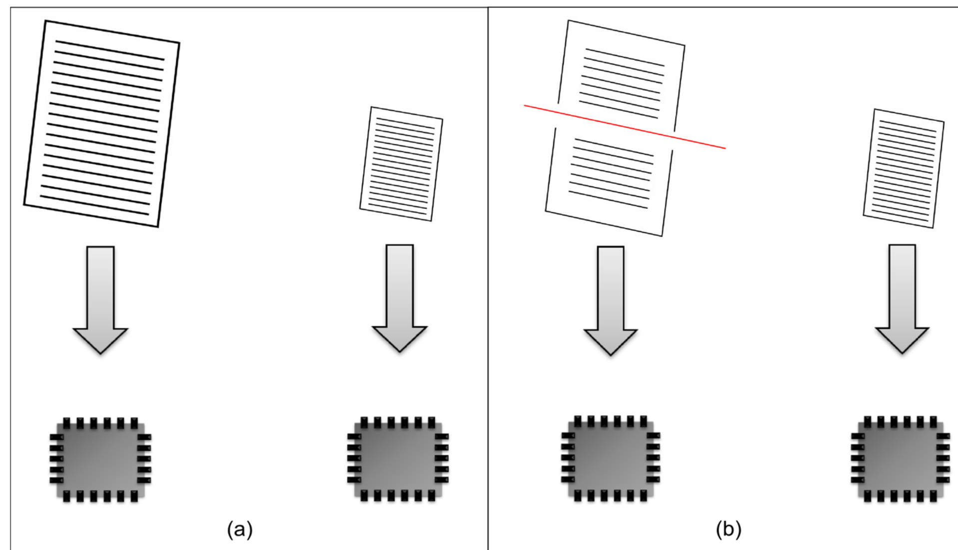


Source: the author

number of issues (instructions that can be sent to execution) available, while in HARTMP the ILP capacity is limited by the size of the array, reconfiguration memory and input and output length.

For instance, in Figure 4.7(a) we illustrate two heterogeneous threads. The one in the left is bigger, meaning has more potential for ILP exploitation in the array. The one in the right is smaller, so its kernels are composed of groups of fewer instructions. In this case, both threads are being scheduled on DAPs that have large arrays. The processor will be able to execute both threads; however, the DAP that is receiving the smaller thread will not use most of its functional units on the reconfigurable datapath – as illustrated on Figure 4.7(b) – simply because the configurations this thread generates do not require all the resources available. Similarly, in the Figure 4.8(a) we show an example where the same heterogeneous threads are executed in small DAPs. Again, the homogeneous processor will be able to run both threads, however – as illustrated in Figure 4.8(b) –, 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 it has 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.

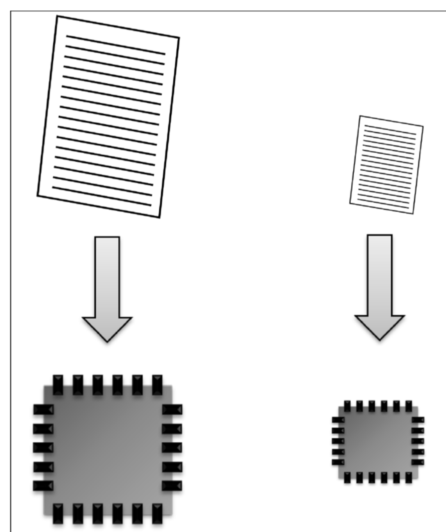
Figure 4.8: (a) Heterogeneous threads running on small cores. (b) Threads of bigger basic blocks have to be split.



Source: the author

We aim to achieve a configuration like the one illustrated in the Figure 4.9, where the larger threads are scheduled to bigger DAPs and smaller threads to smaller DAPs. The expected result is not only to have better energy efficiency, but also – in some cases – better performance. If a system which was only composed of small DAPs now possess a few large, threads with bigger basic blocks can be accelerated. On the other hand, a system which had only larger DAPs can now replace some of them by several smaller ones, increasing the potential for exploiting TLP.

Figure 4.9: Heterogeneous threads running on heterogeneous cores. Load balance can be reached.



Source: the author

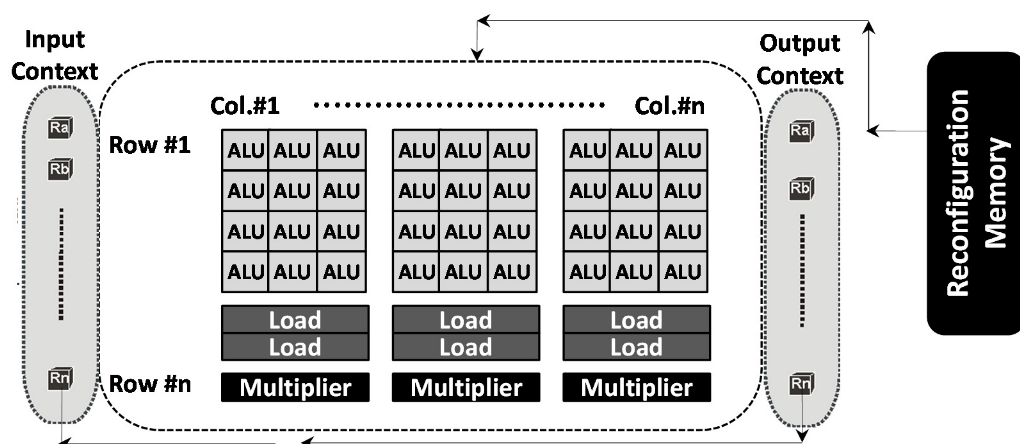
4.4 Leveraging Heterogeneity with the DAP

As mentioned earlier, heterogeneity on HARTMP is reached by changing the resources on the reconfigurable unit of the DAP. There are several ways one can change the size of the reconfigurable fabric: by increasing the size of the input/output context or reconfiguration memory or even the number of functional units available for execution. Figure 4.10 is an extract of the Figure 4.4 showing the reconfigurable unit and the blocks that can be changed. On this section we will present details on the impacts of changing each of these parameters.

4.4.1 Changing the Input/Output Context

The input context of the reconfigurable unit is a set of registers responsible for keeping temporary data loaded from the register file of the DAP. It is also responsible for feeding the initial data for the functional units inside the reconfigurable logic. The output context, on the other hand, is responsible for keeping the data that must be written back to the GPP register file. One can increase the size of the context so the array can receive more data from the GPP. If the input/output context is not large enough, the configurations on the array must be split to fit the input. Moreover, the register renaming (needed for handling false dependency) would be affected as well, as fewer register names would be available. Therefore, speculation, which heavily depends on register renaming for handling false dependencies, would also be affected.

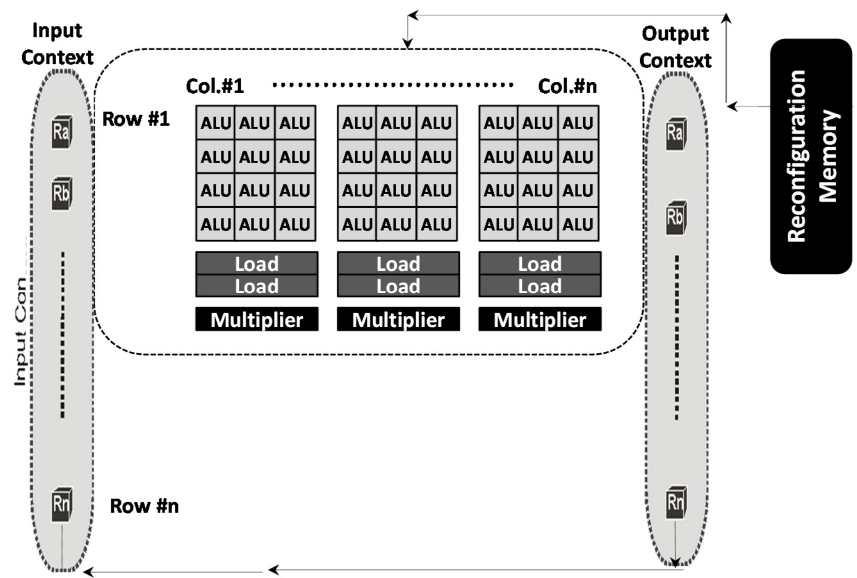
Figure 4.10 Reconfigurable unit of a DAP. Blocks consist of Input/Output Context, Functional Units and Reconfiguration Memory.



Source: the author

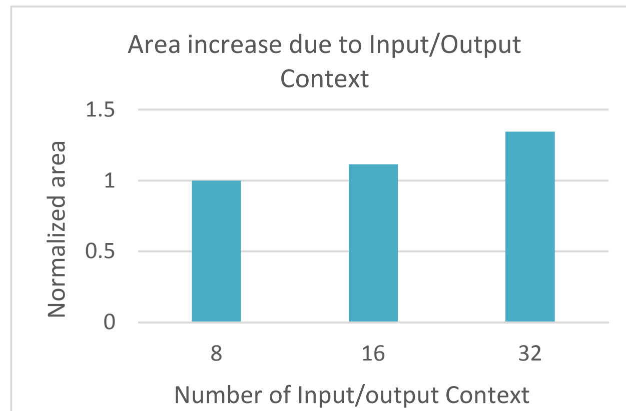
A disadvantage of having large input/output context is the load time. Most register files have only two read ports, meaning that if a reconfigurable unit has an input context of 16 registers, it would need 8 cycles to load all the required data inside the logic. Furthermore, as seen in the Figure 4.5, for each functional unit in the reconfigurable unit, there is an output multiplexer for each other register on the output context. Thus, increasing the output context greatly increases the number of multiplexers on the reconfigurable unit. Figure 4.12 shows the increase in area due to simply increasing the length of the input/output context. A context of 16 registers is 11% bigger than a context of 8 registers, while a context of 32 registers is 34% bigger.

Figure 4.11: Increasing the Input/Output Context of the reconfigurable unit. More registers are available for feeding the functional units and for register renaming.



Source: the author

Figure 4.12: Area increase due to Input/Output Context increase.

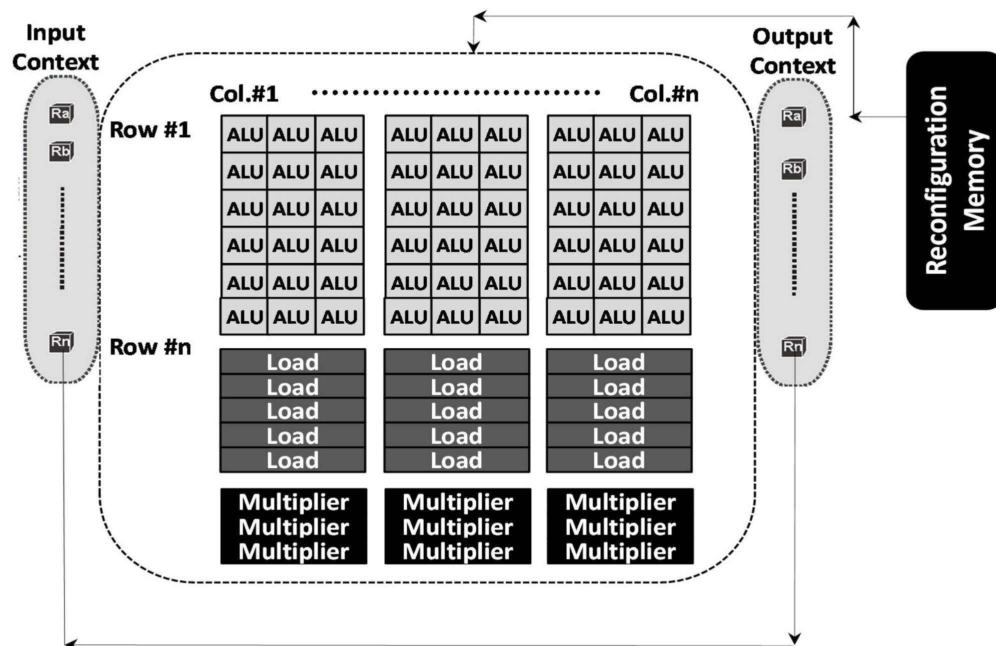


Source: the author

4.4.2 Changing the amount of functional units

The functional units on the reconfigurable unit are responsible for executing the many instructions of the configuration. The higher the number of functional units, the more parallelism can be exploited. Thus, the number of functional units directly impacts the levels of exploitation in instruction parallelism.

Figure 4.13: Changing the number of functional units. More ILP can be exploited both from bigger basic blocks and from higher levels of speculation.



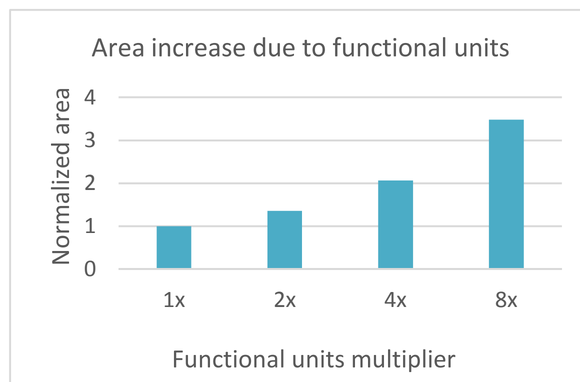
Source: the author

On the other hand, a reconfigurable unit with too many functional units might waste too many resources if the application running on it cannot fully use all the hardware provided. Idle functional units in a reconfigurable system introduce a huge overhead in area that could be used for other resources (like cache memories or even more DAPs) and increases the configuration size. Figure 4.14 shows the increase in area due to the number of functional units. When the number of functional units is doubled, the area increases in 35%, without considering the input and output contexts and the reconfiguration memory. However, when the number of functional is multiplied by 8 times, the area increases 3.5 times, primarily because the number of multiplexers also grows greatly. Furthermore, these units also increase the total static power of the system: the power a circuit consumes for being active, but not necessarily in use.

4.4.3 Changing the Reconfiguration Memory

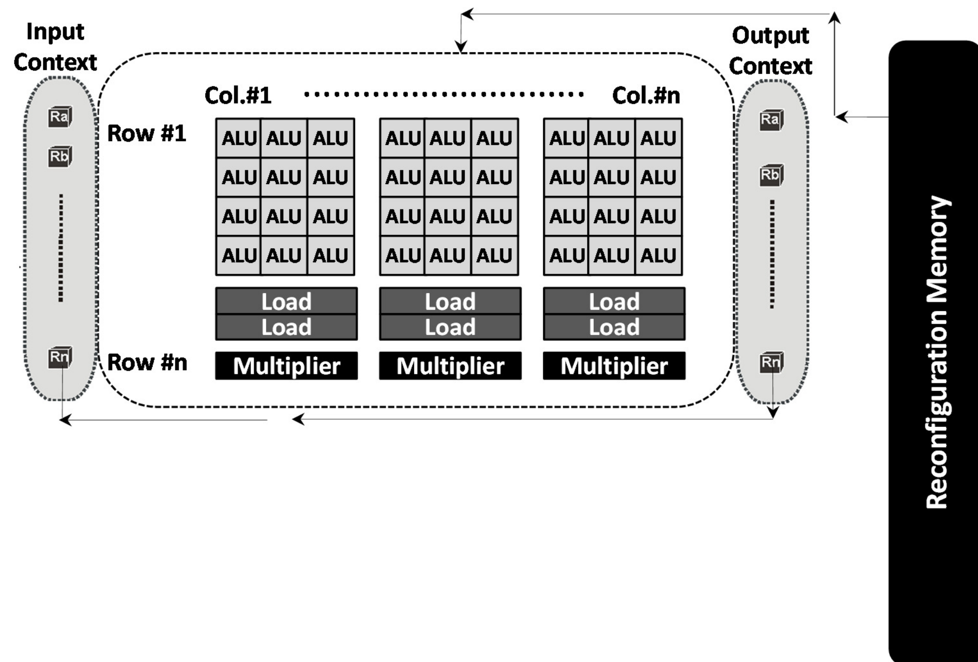
The reconfiguration memory is responsible for storing the bits of each configuration. These bits control the active functional units, the function they operate (arithmetic, logic, float point …) and the datapath of the configuration (multiplexers), as seen in the Figure 4.5. Thus, the size of this memory is dependent on the size of the reconfigurable unit as a whole: the amount of functional units and the length of the input/output context. Furthermore, the size of the reconfiguration memory also dictates the number of configurations that can be stored. When this memory is full, a policy of replacement must be adopted to free space for new configurations, which might lead to wrong replacement decisions. The bigger this memory is, the less the replacement policy will be applied.

Figure 4.14: Area increase due to functional units



Source: the author

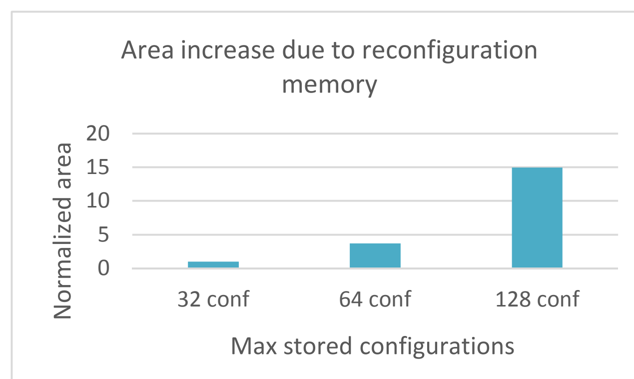
Figure 4.15: Changing the reconfiguration memory size. More configurations can be stored, resulting in less configuration misses due to replacement policy.



Source: the author

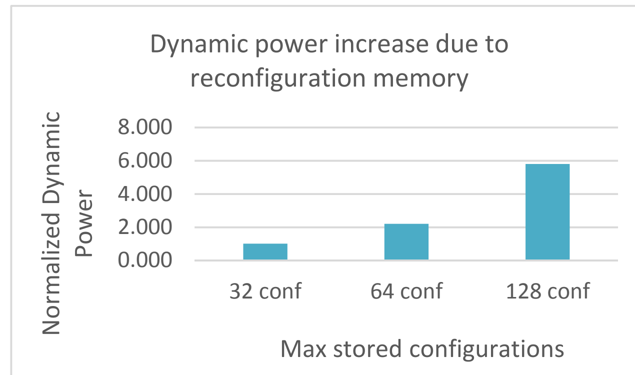
On the other hand, the reconfiguration memory is a cache-like memory that takes up a huge chunk of area on the chip. Figure 4.16 shows that a memory that can hold 128 configurations is almost 15x bigger than one that can hold 32 configurations. Furthermore, the larger the memory is more power it consumes at read and write operations. As can be seen in the Figure 4.17, the memory that can hold 128 configurations consumes approximately 6 times more dynamic power on read and write operations. Thus, increasing too much the reconfigurable memory might lead to huge area and power overheads.

Figure 4.16: Area increase due to reconfiguration memory.



Source: the author

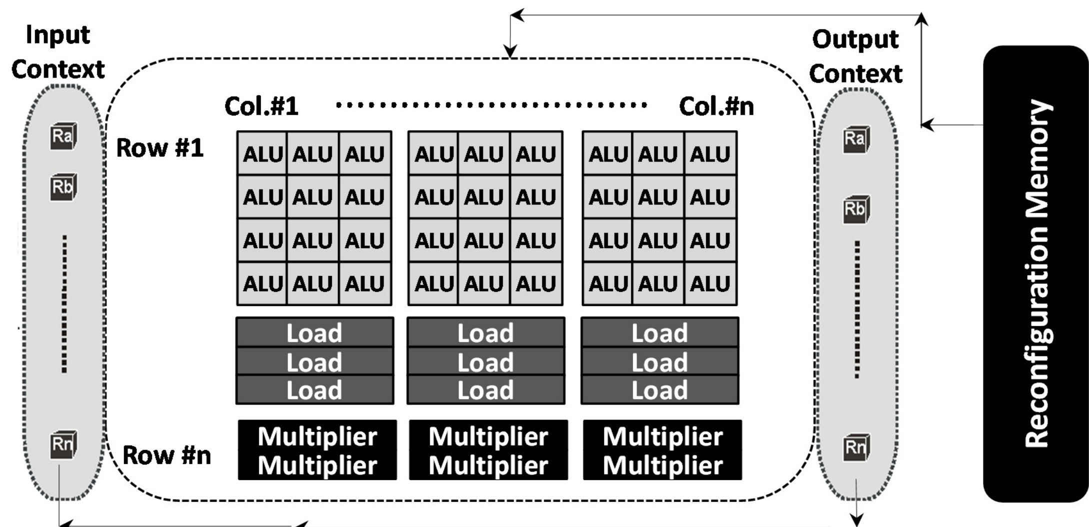
Figure 4.17: Dynamic power increase due to reconfiguration memory. Memory context impacts greatly on energy consumption.



4.4.4 Finding the best reconfigurable unit configuration

To achieve good results in the reconfigurable array, without wasting resources, one must find a balance between the different parameters to set. In our previous work (SOUZA, 2014), an extensive evaluation of several configurations for the reconfigurable unit was proposed. The two best configurations were selected to continue the study in heterogeneity, which have resulted in this work and will be detailed next, in section 6.4.

Figure 4.18: A good configuration must consider a balance between the different parameters of the reconfigurable unit to avoid system bottlenecks.



Source: the author

5 SCHEDULING ALGORITHMS

A critical point of heterogeneous systems is the scheduling algorithm in use. Scheduling is the act of allocating work in resources that will execute the job. In a multicore system, the scheduler is responsible for distributing jobs for DAPs keeping a good load balance between them. In a heterogeneous system, this becomes an even more complex job, as the DAPs can have different resources and performance. The main idea of heterogeneous systems is to execute applications efficiently, according to their needs. However, if a thread is badly allocated, the heterogeneous system can introduce serious drops in performance. For instance, if a heavy-duty thread is allocated to a simple DAP, the whole system might need to halt, waiting for that thread to finish.

In this work, we have implemented six scheduling algorithms to analyze the impacts of correct allocation in HARTMP. The first algorithm is the Oracle scheduler, which can always determine the right allocation for optimal performance. Then, we show the static and the inverse static schedulers, which simply allocate a thread to the next free DAP, never reallocating it again. Their only difference comes in the order of the allocation. The next one is the IC-Driven scheduler, which counts the number of instruction between barriers in each thread and reallocates using this information. The Feedback scheduler is similar to the IC-Driven, but uses the number of instructions that were executed inside the reconfigurable logic. Finally, the PC-Feedback, which is a variation of the Feedback scheduler, which keeps information about each barrier and stores this information indexing it by its PC. Next time the barrier is executed, the PC-Feedback uses this information to take decision over allocation. All the algorithms are best described in the following sections.

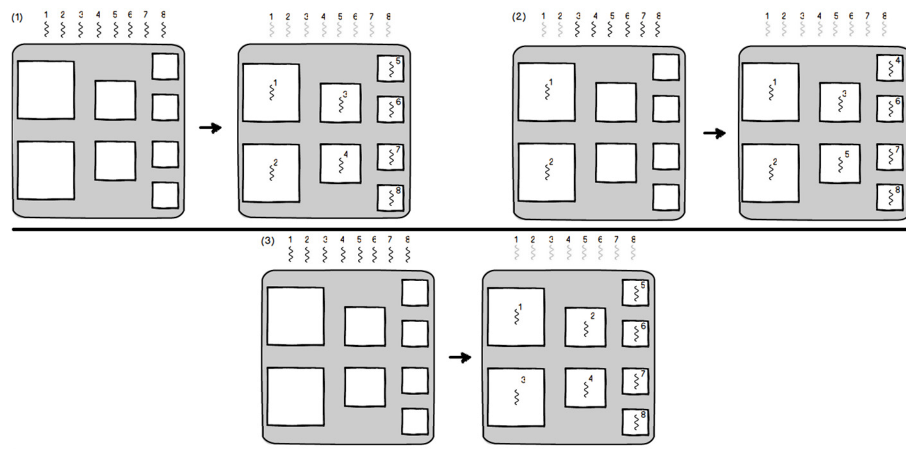
These schedulers were evaluated (as shown in chapter 7) using HARTMP – our reconfigurable heterogeneous organization – without considering costs for thread migration to show the best scenarios for each of the algorithms. Furthermore, four of the six algorithms shown in this work are novel contributions: the Oracle, inverse static, Feedback and PC-Feedback schedulers. The traditional static and IC-Driven schedulers were originally created in (RUTZIG, 2012).

5.1 The Oracle Scheduler

The Oracle is a predictive scheduler implemented in C language, which objective is to analyze which DAP – among all – best fits a thread. In this regard, the Oracle uses results generated by pre-executed simulations of all threads in all possible DAP sizes. This process works as follow: Firstly, one needs to define the number of DAPs of the heterogeneous system and their sizes. The algorithm assumes that the number of executed threads is equal to the number of DAPs. Thereafter, the Oracle evaluates each synchronization point (i.e. a point where all threads must meet, such as a join or a barrier) of the program, generating every possible combination of thread-DAP scheduling. The implementation is based on an exhaustive search through these combinations, ultimately selecting the one that outputs the fewest number of cycles. However, this kind of search does not scale to a solvable number of combinations as the number of DAPs grows. To resolve the allocation problem for this work's target number of DAPs (up to 16), we designed an algorithm capable of discarding repeated combinations. Figure 5.1 illustrates the Oracle in three steps.

Step 1 in Figure 5.1 shows how the threads are initially arranged in the DAPs. A disjunctive set of threads is associated with a DAP size. The allocation process is done in a sequential and pre-defined way. In the example, a set is allocated to large DAPs, followed by medium DAPs and finally by small DAPs. This configuration defines the first combination, which will be

Figure 5.1: Steps for the Oracle allocation algorithm.



Source: the author

analyzed by the Oracle, and it will be, mandatorily, the one with the fewest number of cycles at that moment.

Step 2 in Figure 5.1 shows how partial changes of sets occur in order to analyze a set of combinations. Two of the three disjunctive sets are reconfigured and allocated to distinct sizes of DAPs, generating a new combination. In the example, medium and small DAPs have their threads deallocated, while threads that are in large DAPs remain unchanged. Next, a new set of threads (which had not yet been analyzed) is generated and allocated to medium DAPs, while the remainder is allocated to small DAPs. This new combination is analyzed and then compared with the combination that has generated the fewest number of cycles at this point. This continues until all formed combinations of disjunctive sets of threads are generated and analyzed.

Step 3, in Figure 5.1, shows how changes in the remaining set of threads occur in order to analyze all possible combinations. The remaining disjunctive set is kept unchanged until the moment that all possible combinations of this set with the other two sets are analyzed. When this condition is met, all allocated threads in the system will be deallocated, and a new set is configured for the DAP size where past unaltered threads were allocated. In the example, after generating all possible combinations in medium and small DAPs, all threads were deallocated, including those present in large DAPs. Afterwards, a new combination is generated for large DAPs, and then step 2 repeats. This continues until all possible combinations in large DAPs are generated. The whole process is then repeated for the next synchronization point.

The criterion that defines the dynamics of disjunctive sets in each different DAP size relates with the number of combinations generated by each of them. In this case, the performance gain is associated with the generation of the minimum number of combinations to result in the fewest number of cycles. This is possible because the algorithm needs to generate all combinations only for two DAP sizes. The third size will be automatically resolved, as it will take the remaining threads.

5.2 Static Scheduler

In this solution, when a new thread is created, it is allocated to the next free DAP, blind to any consistency between what the thread needs and the DAP resources. After allocation, the thread executes on the same DAP until completion. The static scheduler adds no complexity to

the hardware or operating system nor extra overhead due to swapping threads between DAPs. Furthermore, most applications have a master thread, which composes the most sequential parts of the application. A property of the static scheduler is that the master thread is always allocated to a large DAP in the HARTMP, as it allocates the first spawned threads in the large DAPs, then in the medium and small, respectively.

Figure 5.2 shows the simple pseudo code of the static scheduler. It runs once, at the beginning of the application. It is important to highlight that in the CoreList, the DAPs are sorted from the largest to the smallest, i.e. in an N-DAP system, the CoreList[0] is always a Large DAP and CoreList[N-1] is always a Small DAP.

Figure 5.2: Pseudo code of the static scheduler.

```
For all indices I of CoreList
        Allocate (ThreadList[I], CoreList[I])
```

Source: the author

5.3 Inverse Static Scheduler

The static scheduler has good performance in many applications, as will be seen on the results chapter. This is due to the property of always allocating the thread 0 into the DAP 0. The thread 0 is always the master thread of the application and, usually, the most sequential and demanding thread. The DAP 0, as already mentioned, is always a large DAP. Thus, the static scheduler usually gets the allocation of the most demanding thread right.

To determine if this assumption is correct, we have created the inverse static scheduler. Instead of allocating the first threads into the first DAPs, we have reversed the scheduling, allocating the thread 0 into the DAP N-1, the thread 1 into the DAP N-2 and so on. This algorithm also runs once, at the beginning of the application. In the Figure 5.3 we show the pseudo code for the inverse static scheduler. In the chapter 7, we will show that our assumption was correct and there are great performance drops from the normal static to the inverse in many applications.

Figure 5.3: Pseudo code of the inverse static scheduler

```
For all indices of CoreList
        Allocate (ThreadList[I], CoreList[N-1-I])
```

Source: the author

5.4 IC-Driven Scheduler

This algorithm inserts a simple instruction counter for each thread. When a synchronization point is reached, the scheduler is activated. The threads with the highest instruction count between the last and current synchronization point are swapped to the larger DAPs, while threads with fewer instructions are allocated to medium and small DAPs. Figure 5.4 shows a simplified pseudo-code for the IC-Driven scheduler. The proposal of this algorithm is not to reach near Oracle performance results, but rather to evaluate the impact of having a simple real-time dynamic scheduler on the system. This implementation is the instinctive way to exploit ILP from DAPs with different resources.

One of the disadvantages of this algorithm, however, is the high likelihood of miss allocation when the threads change their behavior constantly. A thread might be running many instructions between one synchronization point, which would make it swap to a larger DAP on the next barrier. However, this same thread might change its behavior and run fewer instructions. This would create a wrong allocations, because the thread is using information from the past to try to guess the future. Another disadvantage of this algorithm is that HARTMP extracts parallelism from the instructions only inside the reconfigurable array, but the application might have many instructions executed in the GPP, if they are not supported by the array. Thus, the number of instructions executed by the DAP is not a suitable metric for measuring the load inside the array.

Figure 5.4: Pseudo code for the IC-Driven Scheduler.

```
OrderByInstructionCount (ThreadList, Descendant)
While notEmpty ThreadList
    Thread <= Remove (ThreadList, First)
    If notEmpty LargeCoreList then
            Allocate (Thread, Unqueue (LargeCoreQueue))
    If notEmpty MediumCoreList then
            Allocate (Thread, Unqueue (MediumCoreList))
    If notEmpty SmallCoreList then
            Allocate (Thread, Unqueue (SmallCoreList))
```

Source: the author

5.5 Feedback Scheduler

The HARTMP is a reconfigurable system that translates instructions from a basic block to the reconfigurable logic, aiming to accelerate this code region by exploiting the parallelism of the instructions and executing them in a combinational fashion. Inside the DAP, instructions can be executed in the classic GPP (SparcV8 in the case of HARTMP) or inside the reconfigurable unit. The SparcV8 processor is single-issue, which means that it can only execute one instruction in each stage of its pipeline. All the instruction parallelism in the DAP is exploited inside the reconfigurable array, where many functional units are available. However, not all the application instructions can be executed inside the array, so they have to execute on the GPP. Because of this, using the number of executed instructions by the DAP as a metric of ILP exploitation is not a good parameter for our system. Instructions executed by the reconfigurable logic represent the true load in instruction parallelism inside the DAP. Furthermore, it is on the reconfigurable unit that lies heterogeneity on a HARTMP system. Thus if a thread executes more instructions on the reconfigurable block than other, than this thread should migrate to a larger DAP.

To exploit such characteristic of the HARTMP system, we proposed the Feedback scheduler. By adding simple hardware counters inside the DAP, one can send the number of

instructions that are executed inside the reconfigurable block only as a feedback to the operating system. With this information, the scheduler can decide the next allocation of threads more precisely. The algorithm of the feedback scheduler is very similar to the IC-Driven, however, instead of using the number of instructions executed by the thread (GPP and reconfigurable unit), it uses only the instructions executed inside the reconfigurable unit. Figure 5.5 shows a pseudo code of the Feedback scheduler.

5.6 PC-Feedback Scheduler

Although the Feedback scheduler considers the particularities of the HARTMP system, it still is an algorithm like the IC-Driven: it directly tries to find the best allocation by using information from the past. As already discussed on the section 5.4, this might lead to miss allocations when the thread changes its behavior constantly. To try to solve this problem, we propose the PC-Feedback scheduler, which uses the PC to identify barriers and their behaviors, so the following times a barrier is executed, the scheduler already knows its approximate load. Most multithread applications have fixed synchronization points – like joins in loops – that repeat during the execution. Like the Feedback scheduler, the PC-Feedback also uses the feedback from the hardware counter to consider only the instructions executed inside the reconfigurable logic.

Figure 5.5: Pseudo code for the Feedback scheduler.

```
ThreadList <= GetArrayInstructionCount()
OrderByArrayInstructionCount (ThreadList, Descendant)
While notEmpty ThreadList
    Thread <= Remove (ThreadList, First)
    If notEmpty LargeCoreList then
        Allocate (Thread, Unqueue (LargeCoreQueue))
    If notEmpty MediumCoreList then
        Allocate (Thread, Unqueue (MediumCoreList))
    If notEmpty SmallCoreList then
        Allocate (Thread, Unqueue (SmallCoreList))
```

Source: the author

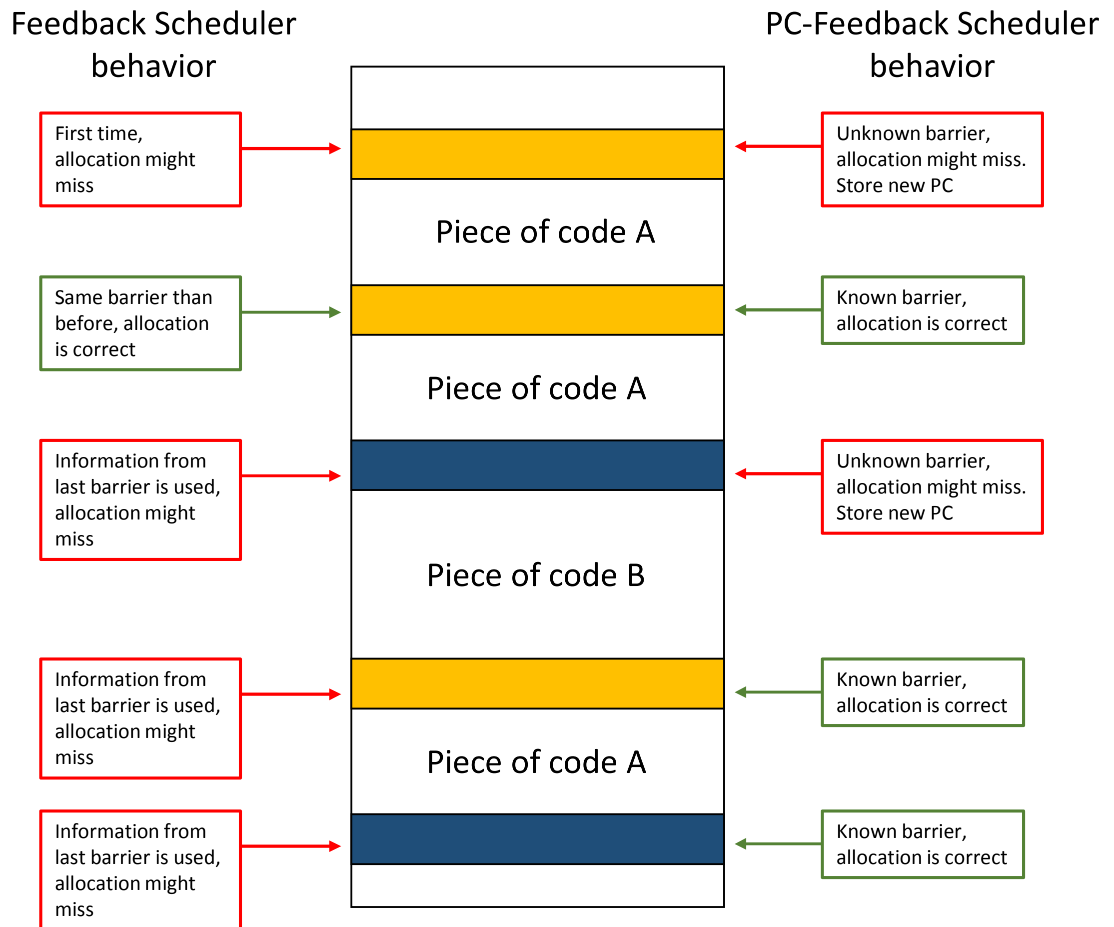
The PC-Feedback works as follows:

1. The first time a barrier is found, the scheduler does not have any information of that PC, thus it does not change the allocation of threads. The scheduler, then, gets the information from the hardware counter in each DAP, indexes it by the barrier PC and stores them for future use in a memory table.
2. If the barrier exists in the memory table, the scheduler loads the information from the threads, sort the threads by the number of instructions executed in the reconfigurable array, and schedules the threads to the DAPs: the threads that execute more instructions on the reconfigurable unit are allocated on the larger DAPs and the rest on the medium and smaller.

The advantage of the PC-Feedback scheduler is that it does not try to guess the correct allocation by using the last barrier behavior. This scheduler only changes threads and DAPs if it knows the behavior of the barrier that will be executed. The disadvantage is that this scheduler does nothing the first time a barrier is executed, because it does not have information on it. Thus, if an application has few repeating barriers, than the scheduler is rarely activated.

Figure 5.6 shows an example of the behavior of the Feedback and the PC-Feedback schedulers and compares them. In the example, the Feedback scheduler allocates the threads correctly only once, while the PC-Feedback is three times right, only missing when a new thread appears.

Figure 5.6: Behavior of the Feedback and the PC-Feedback schedulers compared while running a code trace. The red stripes represent always the same barrier (same PC), while the blue stripes represent a second barrier.



Source: the author

Figure 5.7: Pseudo code for PC-Feedback scheduler.

```
BarrierPC <= GetCurrentPC
If hasPCInfo (BarrierPC) then
        ThreadList <= GetThreadCountOnBarrier (BarrierPC)
   OrderByArrayInstructionCount (ThreadList, Descendant)
   While notEmpty ThreadList
        Thread <= Remove (ThreadList, First)
        If notEmpty LargeCoreList
                Allocate (Thread, Unqueue (LargeCoreQueue))
        If notEmpty MediumCoreList
                Allocate (Thread, Unqueue (MediumCoreList))
        If notEmpty SmallCoreList
                Allocate (Thread, Unqueue (SmallCoreList))
Else
        InsertBarrierInfo (BarrierPC, ThreadList)
```

Source: the author

6 METHODOLOGY

During this work, many tools and simulators were used, along with the creation of several scripts and other pieces of code. In this chapter, we will cover these, presenting the methodology used to reach the results in the following section. We will start presenting both the third party Simics simulator and the in house DAP simulator. Then, we will explain some of the scripts created to integrate both simulators, including a python script responsible for operating as the scheduler of the system. We will also present third party tools, like the cacti-p used to extract power consumption from the reconfiguration memory. Next, a section explaining the used configurations of both CReAMS and HARTMP. Lastly, we will discuss the benchmarks used in this work to validate our conclusions.

6.1 Simulators

6.1.1 Simics Simulator

This work uses Virtutech's Simics (MAGNUSSON et al., 2002), a full system simulation platform. Simics is capable of simulating many components of a system with enough performance to run unmodified operating systems and applications. It provides a deterministic, controlled and virtualized environment capable of simulating a great variety of ISAs, including the Alpha, x86-64, ARM, MIPS and SPARCV8 (used in this work). We use Simics as an instruction level simulator mostly to generate a trace of the executed instructions aiming to feed the input of the DAP simulator.

For this work, we have used an environment with a Linux operating system running over a single SparcV8 processor. Thus, although running on a single core SparcV8, the Linux operating system can still spawn many threads for each application, creating the multithread environment needed for HARTMP. Each of the simulated benchmarks were developed using either the OpenMP (DORTA et al., 2005) or PThreads (BUTENHOF, 1997) parallel programming interfaces, which allow for dynamic thread spawn during the execution. Details on the benchmarks are shown in the section 6.5. When Simics executes an application, it generates an output trace file with the instructions of the many threads created by the operating

system. These threads are eventually split in individual trace files and used as input for the DAP simulators, as we will explain on the following subsections.

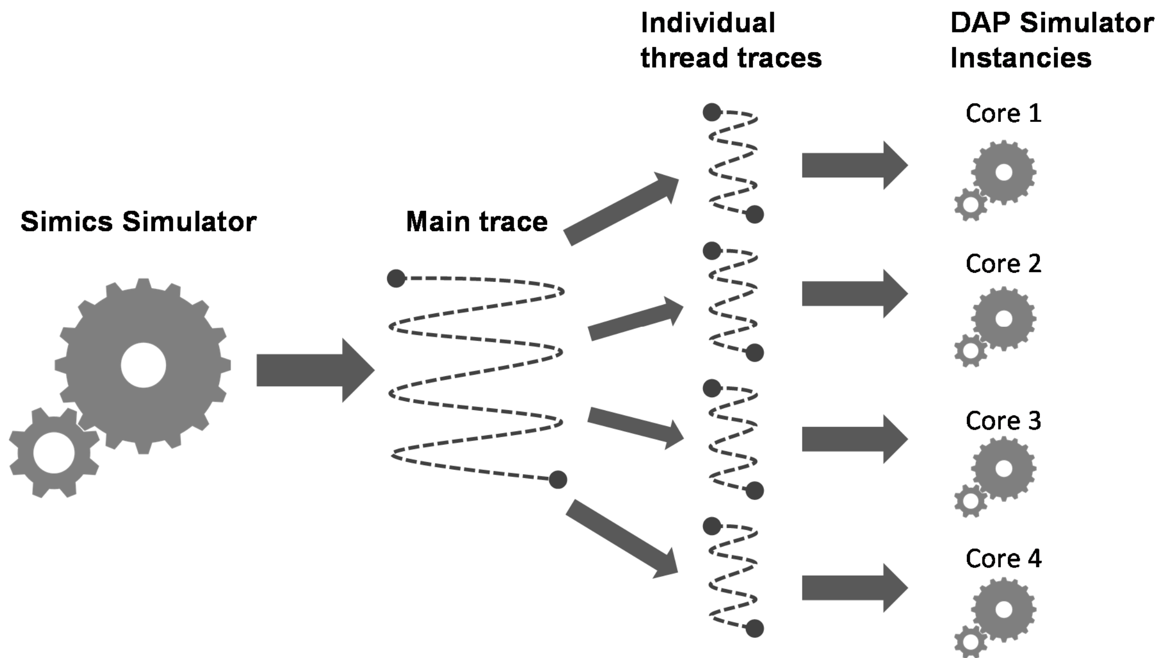
6.1.2 DAP Simulator

The DAP simulator is an in house instruction level simulation platform capable of reproducing the entire algorithm of the DAP, including instruction translation, configuration building, dependencies check, reconfiguration misses and all other features described in section 4.2. The simulator is developed in C++ in a modular project. The dimensions of the reconfigurable unit (including number of functional units, reconfiguration memory lines and input/output length) are completely customizable: the simulator receives a set of input parameters to describe the reconfigurable unit and a trace file containing the instructions of a single thread.

The simulator works as an interpreter: it reads each line of the input trace file, decodes the instruction and starts the DAP algorithm. If the current PC is indexed by the address cache, a configuration is loaded to the array and the simulator accounts the necessary number of cycles and accesses to the functional units/memories. If not, the instruction is accounted as executing on the SparcV8 processor while a new configuration is built on the reconfigurable memory. Details on the DAP simulator can be found in (RUTZIG, 2012).

In this work, we have always worked on multithreaded applications. To deal with multithreading, for each thread of the application, a new instance of the DAP simulator is created, as illustrated in the Figure 6.1, thus if a four thread application is being executed, four instances of the simulator are created. This is a way to simulate the parallel execution of the applications in a multithreaded environment. At each synchronization point, the DAP simulator writes in a result file the collected data for number of executed cycles, power and energy consumption. At the end of the applications execution, a special script (the backward) is called to parse and generate the final results using the partial results of each synchronization point of each DAP.

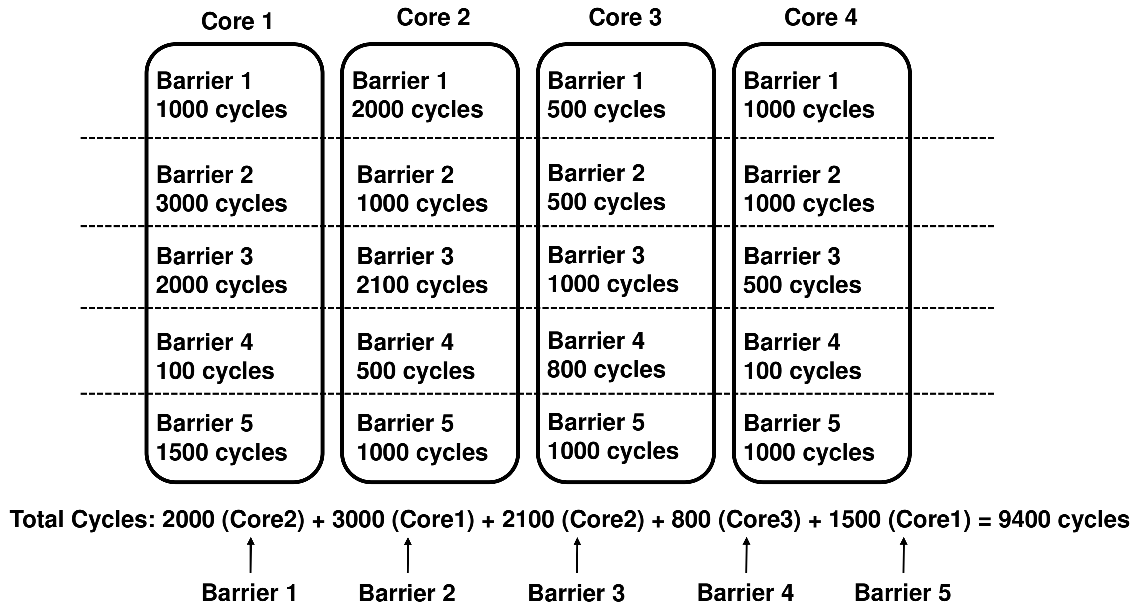
Figure 6.1: Virtutech's Simics generates a trace file with the instructions of all the threads. This main trace is split to create individual trace files for each thread, which are used to feed the inputs of the DAP simulator. One instance of a DAP simulator is created for each trace file (thread).



Source: the author

If a HARTMP system is composed of four DAPs (four running threads), then the backward script needs to read four files for the partial results of each DAP. For performance measurement, the backwards script always considers the DAP that has taken the most number of cycles, as illustrated in the Figure 6.2. For power and energy measurements, all the DAPs must be accounted, as they are always operational, independently which one is the fastest or the slowest.

Figure 6.2: Operation of the backwards script for performance. At each barrier, the core that took most number of cycles is accounted, as the others need to wait for the slowest core to reach the synchronization point.



Source: the author

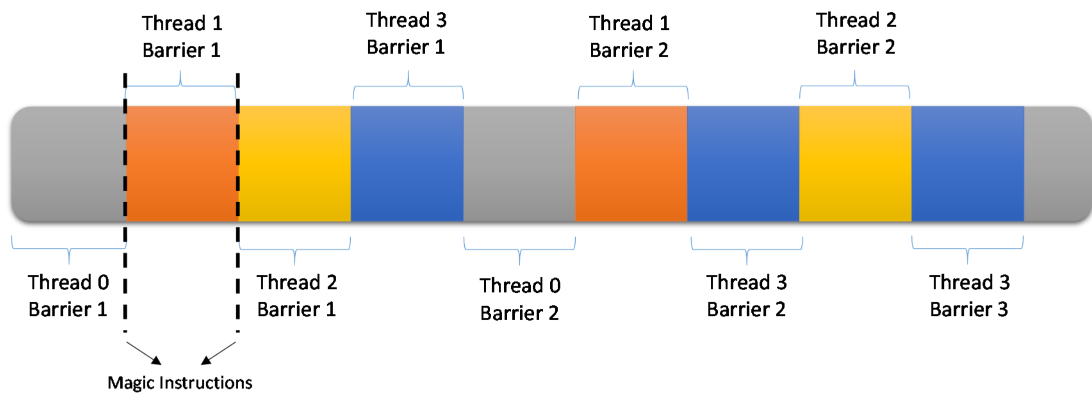
6.2 Scheduling

6.2.1 Scheduling scripts

As already discussed, Simics is responsible for generating a main trace file with the instructions of all threads. However, the DAP simulator needs a trace file of each thread for each DAP. Between these two processes, there is a script that reads the main trace file and splits it into many other files, sorting the instructions of each thread. This script also has a secondary functionality: it schedules the threads into DAPs.

The trace file generated by Simics contains all the instructions of all the threads as if they were executed in a single DAP processor, i.e. they are all mixed, as illustrated in Figure 6.3. The trace also has “magic instructions” inserted by Simics to indicate when threads change. For instance, in the Figure 6.3, between the threads 0 and 1 and threads 1 and 2 in barrier 1, magic instructions are inserted to identify which thread owns the following instructions. This is important for the scheduling script to sort the instructions in separated files. Figure 6.4

Figure 6.3: Application trace generated by Simics. It contains all the instructions of all the executed threads.



Source: the author

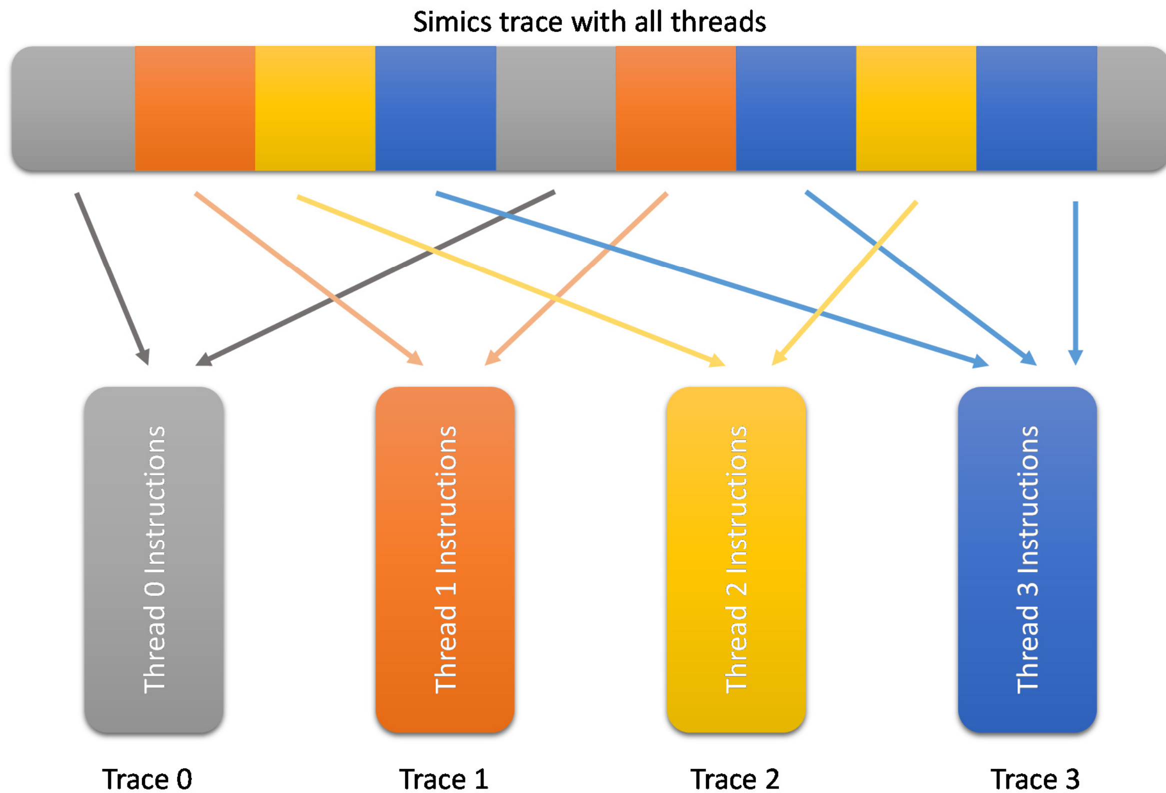
illustrates how the main trace file is split to generate individual traces for each thread. Then, as shown in the Figure 6.1, these individual traces feed the input for each instance of the DAP simulators.

Another function of this script is to schedule the threads between DAPs. For instance, in the static scheduler, the script simply reads the main trace file until it finds a special instruction indicating which thread owns the following instructions, then the script starts writing in the appropriate individual trace file (representing that thread). When the script finds a new special instruction, it stops writing in that file and starts writing in the file of the new thread.

If the IC-Driven scheduler is being used, an account of the number of instructions written on each file is kept and when a barrier instruction for each is found, the script swap the way it writes in the files. Suppose thread 1 is being written in the input file of DAP 1 and must change with thread 2 (being written in DAP 2), then the script will start writing the instructions of thread 1 in the input file of DAP 2 and thread 2 in the input file of DAP 1. This behavior is illustrated in the Figure 6.5.

All other algorithms, but the Oracle, are developed in the same manner. The Oracle needs to execute all threads in all DAPs to decide the best combination, thus it is better applied after the simulation has finished and all the results are at hand. A complete explanation of the Oracle methodology is given at the following subsections.

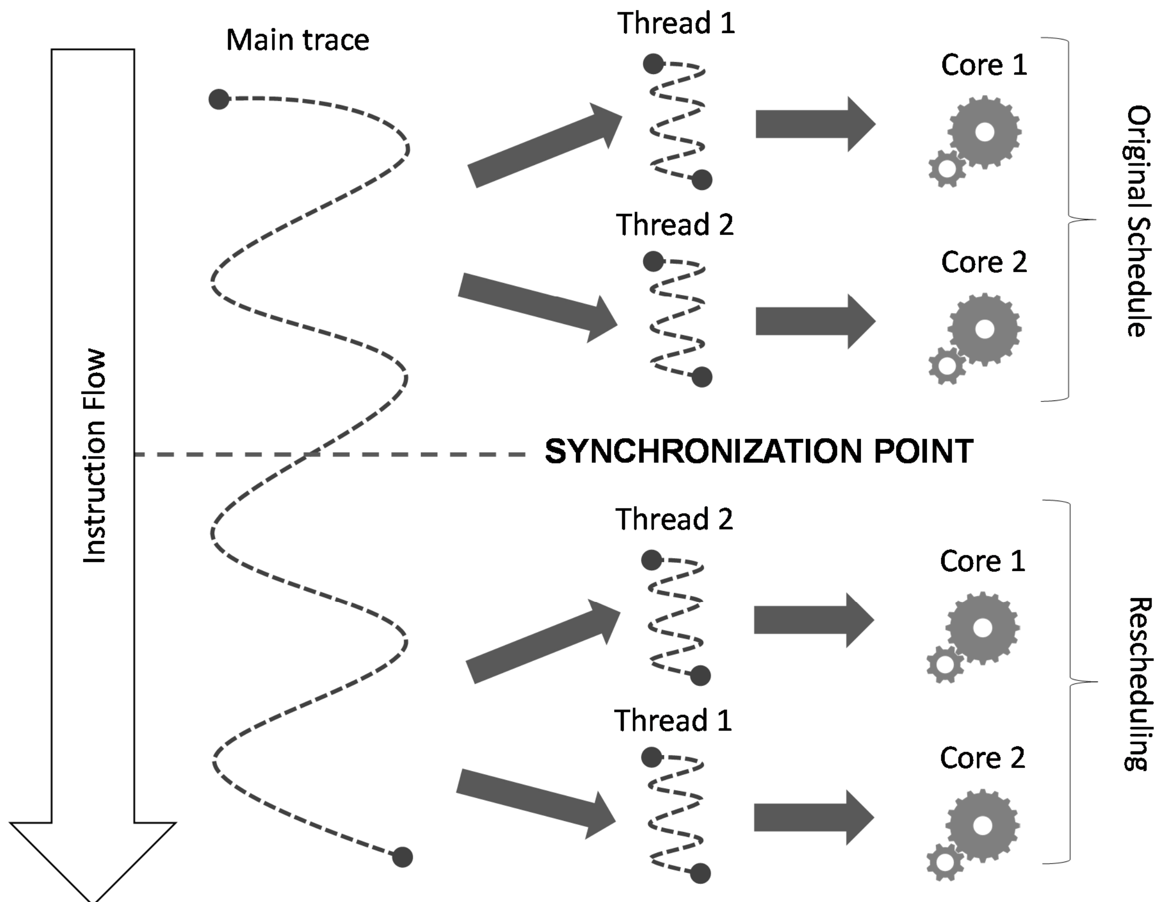
Figure 6.4: Main trace generated by Simics is split in many other files, one for each thread.



Source: the author

Simics, the scheduling script and the DAP simulator all run in parallel, for two reasons: the size of the traces generated by Simics can occupy tens of gigabytes of disk storage and time of simulation. Each of the three processes creates files for the other to consume, they need another process for communication. Mkfifo is used for this propose. Mkfifo is a named pipe: a Linux process that manages automatically a first-in-first-out (FIFO) behavior between two processes. Processes can access the FIFO information as any other normal file, with the difference that when an input is read, it is then destroyed. Thus, Simics works as a producer for the scheduling script, inserting new lines on the main trace file. When new lines are written, the scheduling scripts reads them (which causes them to be erased from the main trace) and starts the scheduling algorithm. If no new lines are available on the main trace file, the scheduling script stays on halt until a new line is written. The scheduling script, then, produces the individual

Figure 6.5: Behavior of the scheduling script when using the IC-Driven Algorithm. After the synchronization point, the Thread 1 is written on the input file of Core 2 and Thread 2 on the input of Core 1.



Source: the author

trace files for each thread, which are consumed by the DAP simulators in the same manner. If no new lines are available in one of the files, the respective simulator stay on halt.

6.2.2 The Oracle Methodology

The Oracle does not rely on the dynamic allocation provided by the scheduling script, as it needs information from the past to decide the best possible combination of thread-DAP. Thus, to reach the Oracle results, the benchmarks were simulated using the basic (static) script. However, instead of executing the traditional HARTMP environment, we have executed all

threads on each size of a HARTMP DAP and then run the Oracle algorithm we have developed on all obtained results to find the best allocation.

Figure 6.6 illustrates this methodology. First, all threads run on each DAP size, thus for an N-Threaded application, 3N result files are generated. Then, the Oracle algorithm (as described in section 5.1) is applied to generate the final results with the best allocation.

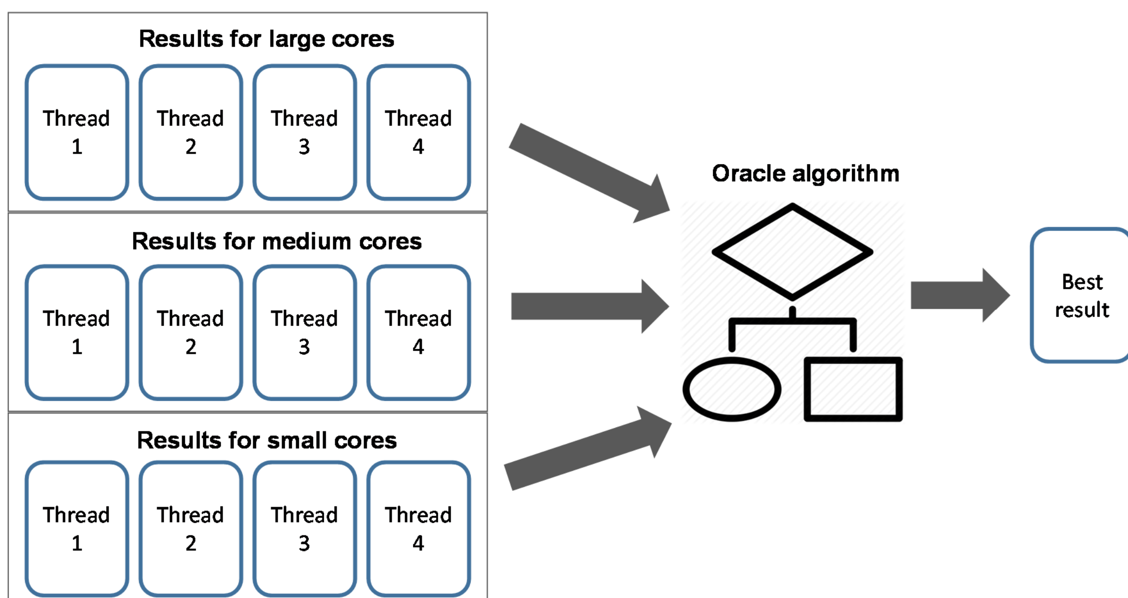
6.3 Third party tools

6.3.1 Synopsys Design Compiler

Synopsys Design Compiler(“Synopsys Design Compiler”, 2010) is a tool capable of synthesizing an ASIC circuit through a hardware description language, such as Verilog, SystemVerilog and VHDL. The tool generates a final gate-level netlist, considering many design constraints, like floorplanning and timing.

This tool was used previously in (RUTZIG, 2012) to describe all the reconfigurable unit functional units, interconnection and other components. All circuits are implemented in VHDL

Figure 6.6: Oracle methodology example for a 4-threaded application. Each core size has to run all the threads, and then the Oracle algorithm uses their results to find the best combination of allocation for performance.



Source: the author

and synthesized to CMOS 90nm technology using Synopsys Design Compiler to extract area and power measurements.

6.3.2 CACTI-P

CACTI-P (LI et al., 2011) is an architecture-level integrated power, area and timing modeling framework for SRAM-based memories with advanced power reduction techniques. It is a version of the well-known CACTI (WILTON; JOUPPI, 1996) tool from HP Labs, but supporting modeling of major leakage power reduction techniques including power-gating, long channel devices, and Hi-k metal gate devices. The main advantage of CACTI-P for our work is that this tool allows for configuring the SRAM-based memories (the caches) for minimum power consumption, instead of the standard minimum delay.

The classic CACTI tool prioritizes the minimum access delay when modeling the SRAM cells. This would give a cache memory with the highest possible frequency of operation. However, as already discussed in section 4.2, our HARTMP system runs at 600MHz. Consider the equation for dynamic power consumption on a switching circuit bellow:

$$P_d = \alpha C V_{dd}^2 f_{clk}$$

In this equation, α is an activity factor related to the power consumed when switching a gate from 0 to 1 (low-to-high), C is the capacitance of the gates, V_{dd} is the operating voltage of the circuit and f_{clk} is the operating frequency. As we can see, if the frequency is reduced, the power of the memories reduces proportionally. However, as the frequency is reduced, the operating voltage of the circuit can also be lowered, which impacts the power consumption almost at a squared rate (V_{dd} has a power 2 impact on the dynamic power). The CACTI-P tool allows the designer to set a determined operating voltage for the circuit and gives the fastest possible operating frequency, considering other necessary constraints (like the size of the cache, the number of lines, associativity).

We have used the CACTI-P tool to model and extract the dynamic power consumption of the reconfiguration memories in all the proposed configurations of DAP in our work. We have aimed to create SRAM memories with the minimum possible power consumption, but still respecting constrains of size and the frequency of 600MHz.

6.4 HARTMP and CReAMS configurations

In this work, we propose to compare – in both performance and energy consumption – the HARTMP, a physically heterogeneous reconfigurable multicore system with its homogeneous counterpart, CReAMS (RUTZIG; BECK; CARRO, 2013). Furthermore, we also evaluate several scheduling algorithms to find the best ways to allocate threads into the HARTMP DAPs. The configurations used in this work for HARTMP were based on the best configurations found in our previous work (SOUZA, 2014), in which many different variations of these systems were compared with three configurations of CReAMS (also used in this work). In the following subsections, we will present the chosen configurations, both for CReAMS and HARTMP.

6.4.1 CReAMS Configurations

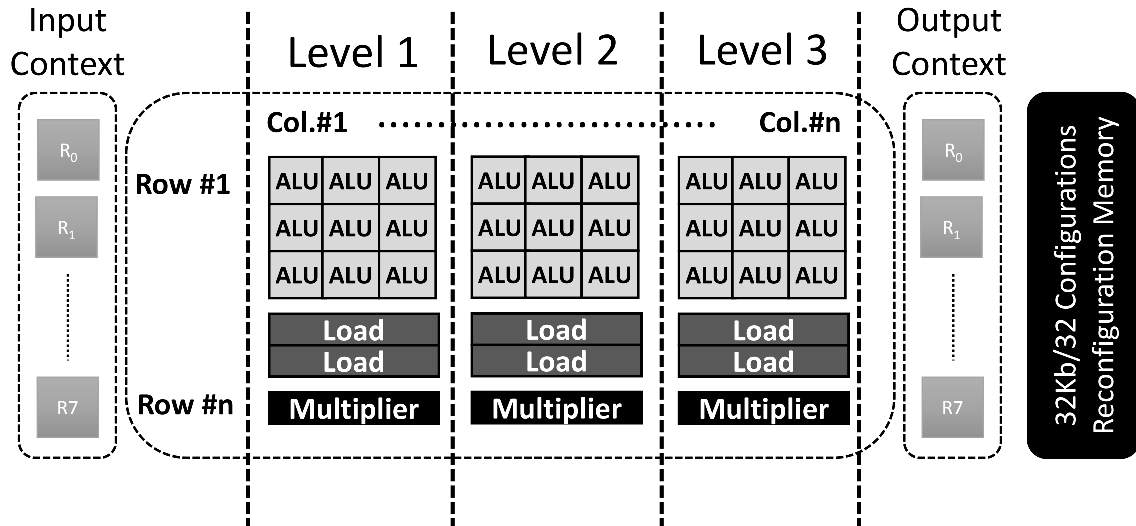
Table I shows the list of resources available for each of the tested configurations of CReAMS. To better understand this table, we will illustrate in the Figure 6.7 the Ho1 (from Homogeneous 1) configuration. As we can see in the Table I and in the Figure 6.7, Ho1 has three levels (as seen in section 4.2, each level takes a processor cycle to complete). In each level, there are one multiplier, two memory access ports and nine arithmetic and logic units (three in each row, as three ALUs can execute in sequence in a cycle). Thus, there is a total of

Table I: The configuration of the reconfigurable unit on the CReAMS systems.

Homogeneous – CReAMS									
Config	**Levels**	**Mult per Level**	**Mult Total**	**Load/Store per Level**	**Load/Store Total**	**ALU per Level**	**ALU Total**	**Reconf Cache**	**Input Context**
Ho1	3	1	3	2	6	9	27	32 Conf 32Kb	8
Ho2	5	2	10	4	20	15	75	64 Conf 64Kb	16
Ho3	8	4	32	5	40	18	144	128 Conf 256Kb	24

Source: the author

Figure 6.7: Example of a DAP for the Ho1 configuration.



Source: the author

three multipliers, six memory access units and twenty-seven ALUs inside the reconfigurable datapath. The input and output context have eight registers each and the reconfiguration table has 32Kb, holding up to thirty-two configurations at a time.

Similarly, the configurations Ho2 and Ho3 are presented in the Table I. Notice that not only the number of functional units increase, but also the number of levels, the input context and the reconfiguration memory; allowing for even bigger configurations to be held. As can be observed, the Ho1 configuration is the smallest, while the Ho2 is medium sized and the Ho3 has very large DAP sizes.

6.4.2 HARTMP Configurations

Table II shows the list of resources on each type of DAP in the HARTMP configurations. The two proposed configurations He1 and He2 have three distinct sized DAPs each: small, medium and large DAPs. On both, 25% of the DAPs of the system are composed of small DAPs, other 25% are of medium DAPs and the last 50% are composed of large DAPs. For example, a 4-DAP configuration of He1 has on small, one medium and two large DAPs, while an 8-DAP configuration has two small, two medium and four large DAPs.

Table II: The configuration of the reconfigurable unit on the HARTMP systems.

Heterogeneous – HARTMP									
He1									
Config	**Levels**	**Mult per Level**	**Mult Total**	**Load/Store per Level**	**Load/Store Total**	**ALU per Level**	**ALU Total**	**Reconf Cache**	**Input Context**
Small (25%)	3	1	3	2	6	9	27	32 Conf 32Kb	8
Medium (25%)	5	2	10	2	10	12	60	64 Conf 64Kb	14
Large (50%)	8	2	16	2	16	12	96	128 Conf 128Kb	20
He2									
Config	**Levels**	**Mult per Level**	**Mult Total**	**Load/Store per Level**	**Load/Store Total**	**ALU per Level**	**ALU Total**	**Reconf Cache**	**Input Context**
Small (25%)	3	2	6	4	12	18	54	32 Conf 32Kb	16
Medium (25%)	5	4	20	4	20	24	120	64 Conf 128Kb	28
Large (50%)	8	4	32	4	32	24	192	128 Conf 256Kb	32

Source: the author

6.4.3 Area parity between HARTMP and CReAMS configurations

Table III contains the DAP ratio between the heterogeneous and the homogeneous processors. For instance, the intersection of Ho1 and He1 shows a ratio of 2:1, meaning that the area of two DAPs of Ho1 is approximately the same as one DAP of He1. The intersection of Ho2 with He1 is 1:1, so both configurations have almost the same area, while a ratio of 1:2 in Ho3 and He1 means that for each DAP of the homogeneous version, two average heterogeneous DAPs could be used. We will use these parities of area to compare processors. For instance, we will compare a 4-DAP He1 (HARTMP) processor with an 8-DAP Ho1 (CReAMS) processor.

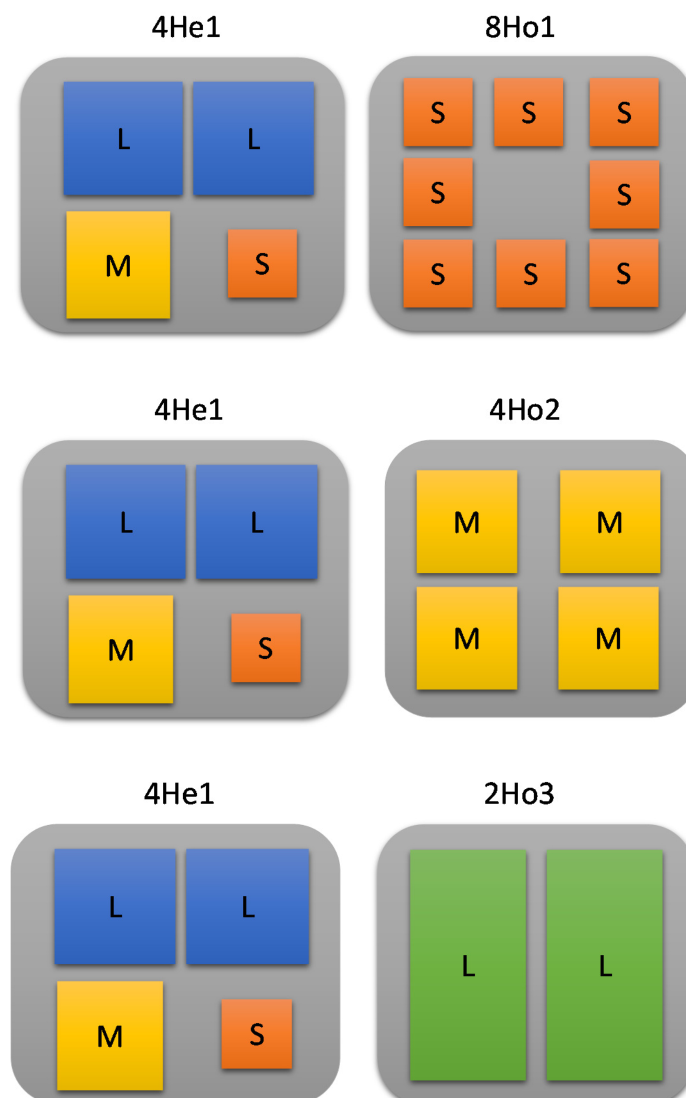
Table III: DAP Ratio between HARTMP and CReAMS configurations

	Ho1	**Ho2**	**Ho3**
He1	2 Ho1 : 1 He1	1 Ho2 : 1 He1	1 Ho3 : 2 He1
He2	4 Ho1 : 1 He2	2 Ho2 : 1 He2	1 Ho3 : 1 He2

Source: the author

Figure 6.8 shows an illustration of the DAP ratio between He1 version and the three homogeneous versions. The 4-DAP configuration of He1 has approximately the same area as a 8-DAP Ho1 processor, a 4-DAP Ho2 processor and a 2-DAP Ho3 processor.

Figure 6.8: Illustration of the DAP Ratio between the He1 configuration and the three homogeneous configurations.



6.5 Benchmarks

A set of benchmarks was used to simulate the behavior of our systems and the proposed scheduling algorithms. To cover a wide range of applications, we have chosen benchmarks from a series of different suites with distinct behaviors. From the parallel suites SPLASH-2 (WOO et al., 1995) and PARSEC (BIENIA et al., 2008) we have selected *FFT* and *swaptions*, respectively. From SPEC OMPM2001 (DIXIT, 1993), *equake* was selected to evaluate HARTMP efficiency over an originally single-threaded application that was parallelized to take advantage of multiprocessing environments. Finally, we have selected three applications (*susan edges, susan corners* and *susan smoothing*) from the MiBench suite (GUTHAUS et al., 2001), which reflects a traditional embedded scenario.

The benchmark *equake* from the SPEC suite was parallelized using OpenMP (CHAPMAN; JOST; VAN DER PAS, 2008) by Rutzig in (RUTZIG, 2012). This library provides methods that discover, at run time, the number of processors of the underlying multiprocessor architecture. Thus, it can take full advantage of the available resources even when the platform changes (e.g. processors are added), with no need for source or code modifications and recompilation.

A previous study on the used applications was done in (RUTZIG, 2012) to characterize their potential on obtaining performance improvements when TLP or ILP exploitation is applied. A simplification of the results is presented in the Table IV, along with a summary of information about the benchmarks. In this table, the more '+' signs a benchmark has, the more it can exploit the given parallelism, while the more '-' it has, the less it can exploit the parallelism. For instance, *equake* from the SPEC suite is bad for both TLP and ILP exploitation, while *susan smoothing* can exploit both parallelisms in good levels.

To characterize the ILP, the sizes of the basic blocks of each application were analyzed. The bigger the mean size of an application's basic block is, the higher is the probability of ILP exploitation inside the reconfigurable unit. To characterize TLP, the percentage of the entire application that is executed in parallel when in a multithreaded environment is applied was measured. The higher is the distribution of load between the DAPs, the higher the TLP is.

Table IV: List of benchmarks used in this work with their expected ILP and TLP exploitation. The more '+' signals a benchmark has, the more it can exploit the given parallelism, while the more '-' the benchmark has, the less it can exploit.

Benchmark	**Suite**	**ILP**	**TLP**
Equake	SPEC OMPM2001	---	---
Susan edges	MiBench	+++	-
FFT	SPLASH-2	+	+
Susan corners	MiBench	+++	+
Susan smoothing	MiBench	++	++
Swaptions	PARSEC	-	+++

Source: the author

7 RESULTS

In this chapter, the many results of the simulations made on both the HARTMP and CReAMS systems will be presented. First, we show the results for the comparison of the two configurations of HARTMP system given in section 6.4.2 against the three configurations of CReAMS given in section 6.4.1, considering performance and energy consumption. After showing that HARTMP has advantages over its homogeneous counterpart and in which scenarios they appear, we focus on exploring several scheduling algorithms (described in section 5) and their performances.

7.1 HARTMP vs CReAMS

In this work, we have simulated two versions of a heterogeneous system (HARTMP) and compared then with other three versions of its homogeneous counterpart system (CReAMS) in both performance and energy consumption. We have also used the EDP (Energy-Delay Product) metric to have a better understanding on the tradeoff between performance gains and reductions in energy consumption and in which cases improving one of the metrics paid off for an eventual reduction on the other. All results in this section were extracted using the Oracle scheduling algorithm, thus they show the best possible performance gains of HARTMP over CReAMS when running the given benchmarks, which, in fact, represent the potential of the heterogeneous system. The following subsections will explore and explain each of these results.

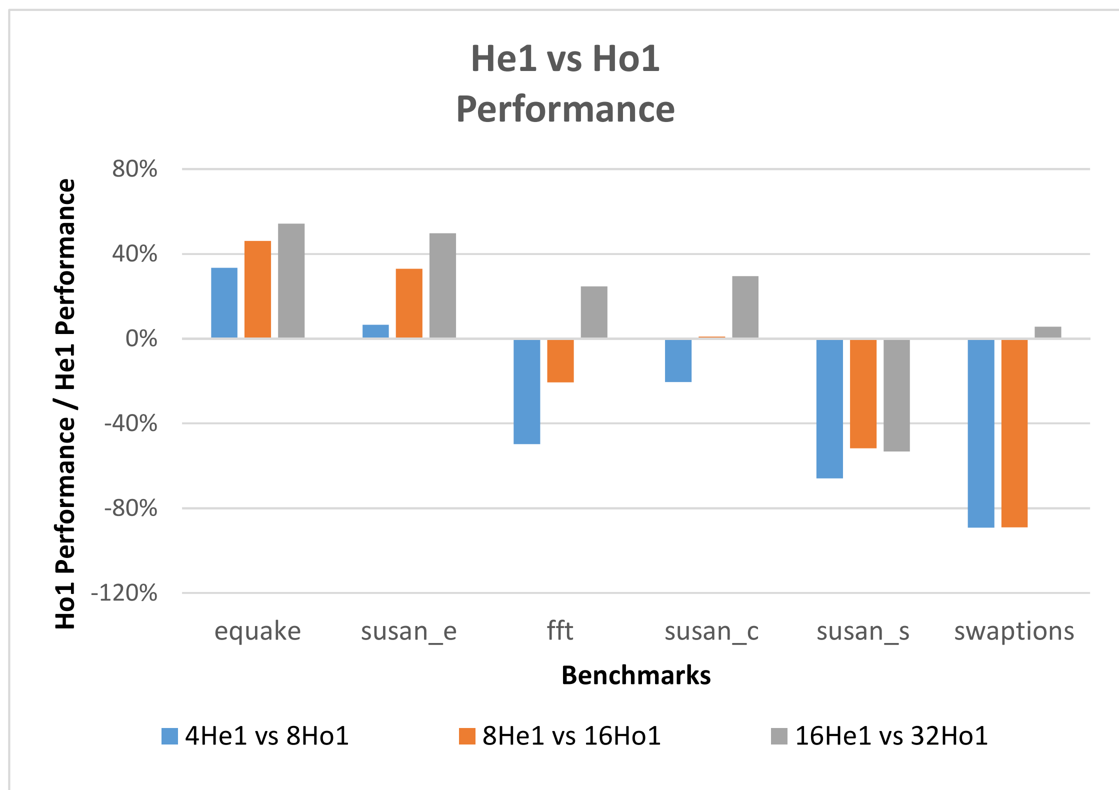
This section will present the results in charts containing the gains of each configuration in relation to the other in percentage. In APPENDIX A, the same results are shown in tables with the values in absolute numbers.

7.1.1 He1 Configuration

7.1.1.1 Performance Evaluation

Figure 7.1 compares the performance (in execution time) of HARTMP and CReAMS. In this chart, the X-axis shows the evaluated benchmarks, which are also disposed in order of TLP exploitation: the leftmost benchmark (*equake*) has the lowest TLP, while the rightmost (*swaptions*) has the highest levels of parallelism between threads. The Y-axis shows the percentage in which HARTMP or CReAMS is superior in relation to the other. If the bar in the Y-axis is positive (above 0%), then HARTMP has better performance, while if it is negative (below 0%), then CReAMS is superior. All of the following charts presented in this section have the same characteristics.

Figure 7.1: Performance comparison between He1 and Ho1. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS. Unbalanced applications have advantages in a heterogeneous environment, while the balanced ones take advantage of the extra cores.



Source: the author

As shown in Table III, the DAP ratio for these configurations (Ho1 and He1) is of two homogenous DAPs for each average heterogeneous DAP[1], thus, in this comparison, CReAMS always has double the number of DAPs than HARTMP. As HARTMP has less DAPs, it is expected that it will perform worst in applications that have high TLP, which is exactly what is observed in the Figure 7.1. As in applications like *equake* and *susan edges* the load balance of applications is low, the HARTMP has more opportunities to correctly arrange the threads on the appropriate DAPs. Even if CReAMS has more DAPs, the Ho1 configuration is composed only of small arrays, thus the threads with higher load will run on the same DAPs than threads with lower loads, creating a bottleneck on the system. On the other hand, applications with higher levels of thread parallelism (like *susan smoothing* and *swaptions*) clearly take more advantage of the simpler (but plentiful) DAPs in CReAMS.

Another interesting behavior to notice is the point in which each application reaches a limit of TLP exploitation, so having more DAPs is not enough to maintain performance gains. For instance, *FFT* has better performance when running on CReAMS until the number of DAPs is increased to 32 in Ho1 and 16 in He1. From that point on, running *FFT* on HARTMP is faster, because the limits of TLP exploitation are extrapolated and having more DAPs increases the performance marginally. HARTMP, on the other hand, has larger DAPs that can be used to efficiently allocate the threads, thus the 16-DAP He1 version is faster than the 32-DAP Ho1. This behavior is even more evident on *swaptions*. This application has almost perfect load balancing between DAPs until 16 threads are spawned. If more threads are created, they are practically not used (due to the way the application was developed), which explains the sudden change in performance.

Figure 7.2 shows the comparison in performance for the He1 and Ho2 configurations. Both have the same ratio of area, meaning that each DAP of the homogeneous versions has, approximately, the same size of an average heterogeneous DAP. As the parities of these configurations have the same number of DAPs, it is expected that both will exploit the same TLP and the source of gains in performance will be mostly from ILP exploitation. As we can see in the Figure 7.2, HARTMP has better performance in most of the applications. This is a

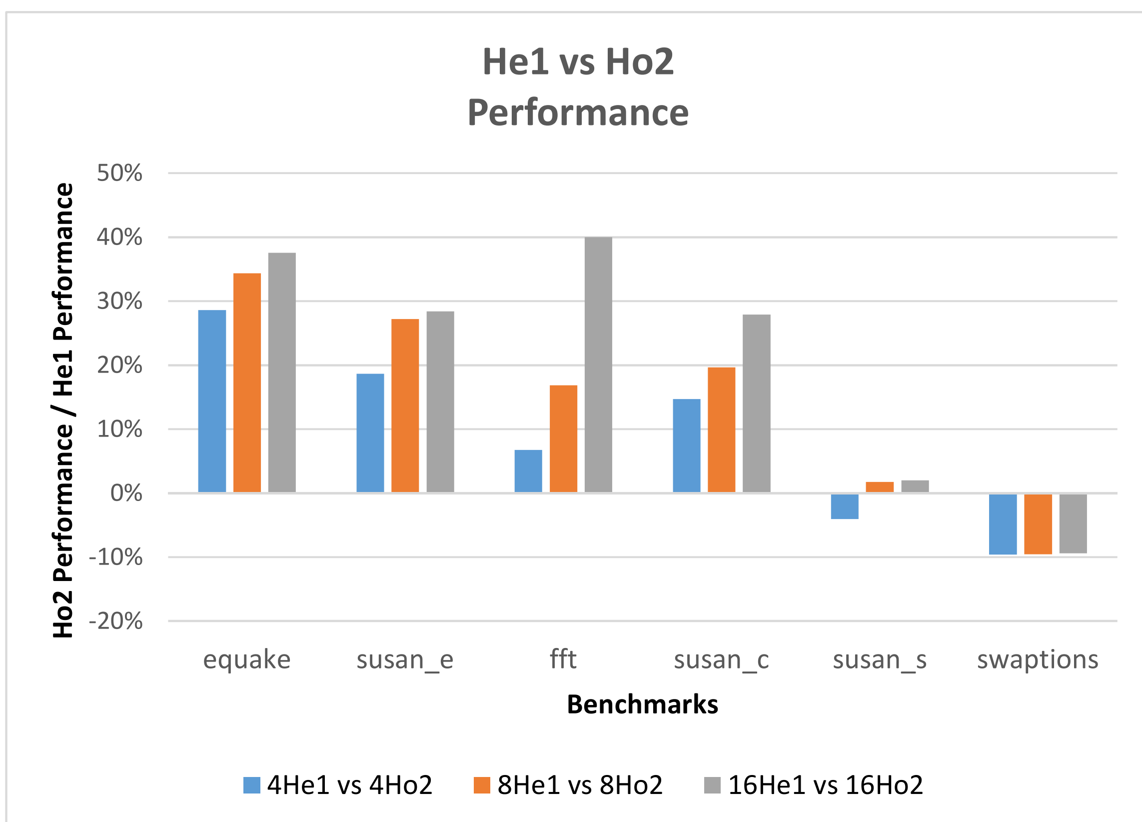
---

[1] An average heterogeneous core is the mean size among all the cores in a heterogeneous configuration. In the configurations used in this work, the average heterogeneous core is the mean size between the small, the medium and the large cores of the given configuration.

result of the combination of larger DAPs and the Oracle scheduler, which is capable of exploiting the most of the heterogeneity. As HARTMP has larger DAPs, it can use these to run threads with higher loads of sequential code, whose instructions can run in parallel, while using the smaller DAPs to run lighter threads. Furthermore, as the number of DAPs grows, more large DAPs the HARTMP configurations have, increasing even further the gains.

The CReAMS configuration is composed only of medium DAPs, which can create a bottleneck if the application has unbalanced threads. On the other hand, the most parallel applications in our list (*susan smoothing* and *swaptions*) have marginal improvements for HARTMP, as the threads are very well balanced. Indeed, in *swaptions*, the threads are so well balanced until 16 threads that the performance improvements in CReAMS are constant. In this application, CReAMS is superior because the small DAPs present in HARTMP hold back the execution of the faster DAPs, creating a bottleneck.

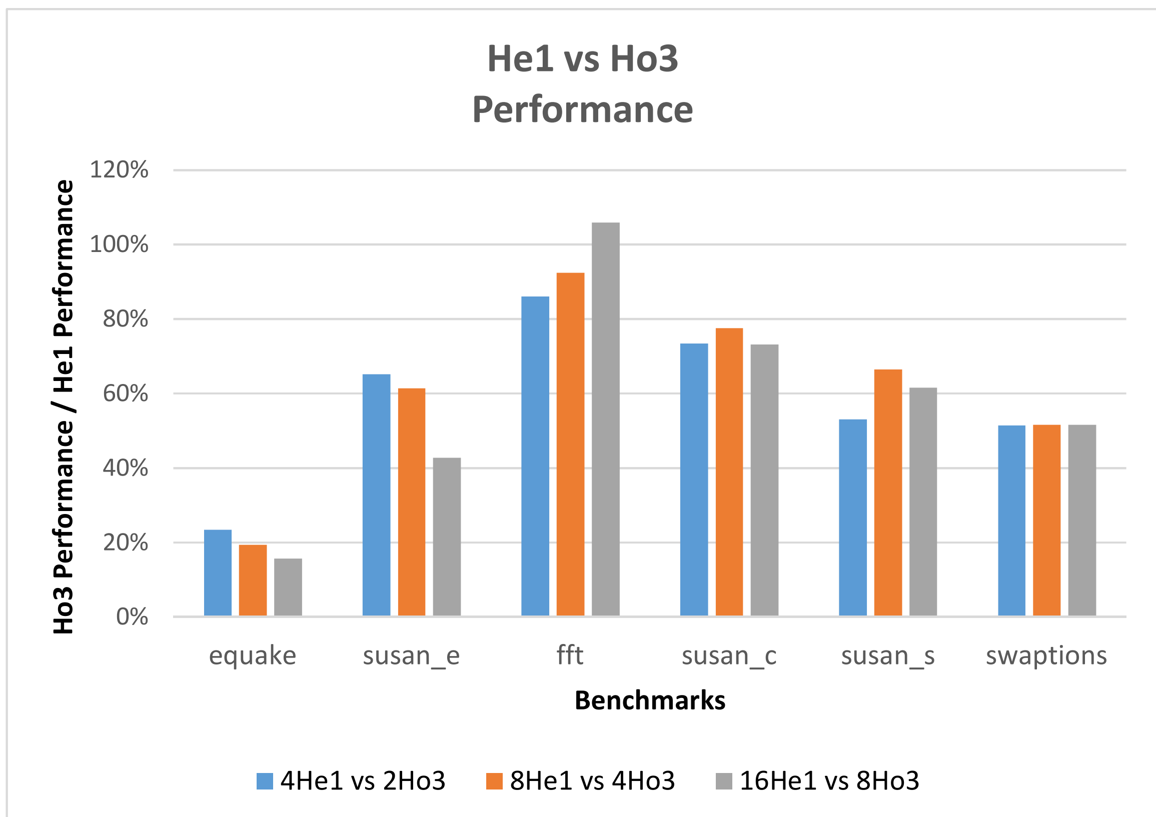
Figure 7.2: Performance comparison between He1 and Ho2. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS. The heterogeneous environment is better for applications with some inbalance in the threads. However, in a perfect balanced application, the small cores of He1 configuration holds the performance of the larger, thus being the homogeneous version (composed of medium cores only) better.



Source: the author

Figure 7.3 shows the comparison of configurations He1 and Ho3. The ratio of these configurations is of two average heterogeneous DAPs for each homogeneous DAP. Thus, in this scenario the HARTMP system has two main advantages: it has more DAPs, allowing for greater TLP exploitation and it can efficiently schedule threads to the DAPs. Indeed, in this scenario the HARTMP system is superior to CReAMS in all the applications, from high to low TLP and ILP, as shown in the Figure 7.3. In most applications, however, as the number of DAPs increases, the gains of HARTMP are diminished. This is due the same reasons as in the Figure 7.1: as the limits of TLP are reached, the impact of increasing the number of DAPs is much smaller and increasing the capacity of exploiting ILP becomes more advantageous. As the CReAMS Ho3 configuration is composed of only large DAPs, it can exploit more ILP from applications than HARTMP.

Figure 7.3: Performance comparison between He1 and Ho3. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS. The He1 configuration can efficiently exploit the ILP from threads while exploiting more TLP as well.



Source: the author

7.1.1.2 Energy Evaluation

Energy consumption is given by the total power of the circuit multiplied by the execution time (in seconds), as in the equation:

$$E_{total} = P_{total} \times t_{execution}$$

Thus, energy consumption is directly proportional to the performance of the circuit (the faster the system, lower is the execution time) and the power consumed by the components (usually, complex circuits have higher power usage). In this subsection, we will analyze the behavior of energy consumption on the simulated configurations considering these two parameters: execution time and circuit size. To complement the analysis of power and energy results, we have added tables in APPENDIX B with the consumption of each component of the reconfigurable unit and memories used.

Figure 7.4 shows the energy consumption comparison for the configurations He1 and Ho1. Ho1 is composed of only small DAPs, thus its DAP has, approximately, half the size of an average heterogeneous DAP. Furthermore, the Ho1 DAP is much simpler, having small cache sizes for the reconfiguration memory. As we have seen in subsection 4.4.3, having smaller caches greatly reduces the power usage of the DAP. On the other hand, Ho1 configuration has double the number of DAPs than He1, which, when fully used, can consume much more power. As we can see in the Figure 7.4, the HARTMP He1 configurations is more energy efficient than the CReAMS Ho1. This is due to two main reasons: HARTMP is faster in most of the applications when comparing these two configurations (seen in subsection 7.1.1.1) and; CReAMS has more DAPs than HARTMP, thus, even when CReAMS is faster (high TLP applications), it has double the DAPs working and consuming power. The peak in energy savings on *swaption* is due to the fast change in performance seen in this application, because of TLP limits being reached.

Figure 7.4: Energy comparison between He1 and Ho1. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS. The homogeneous version has double the number of GPPs and L1 caches, generating an overhead in energy consumption, even when the homogeneous version has better performance.

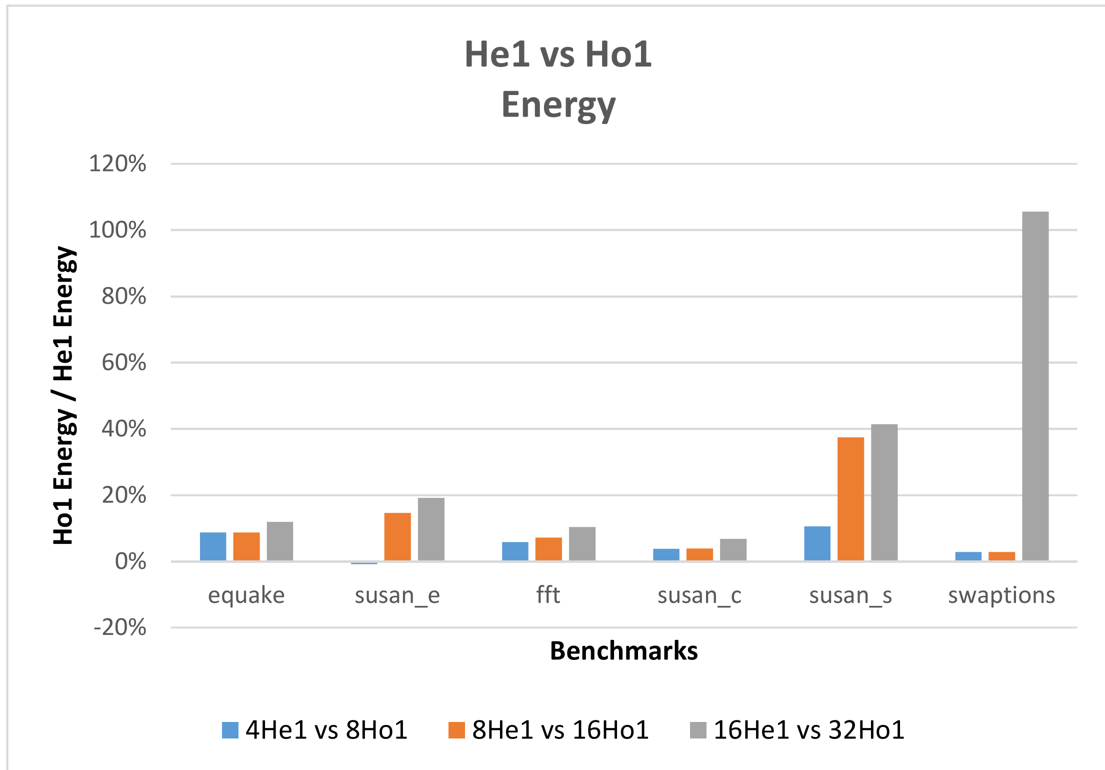
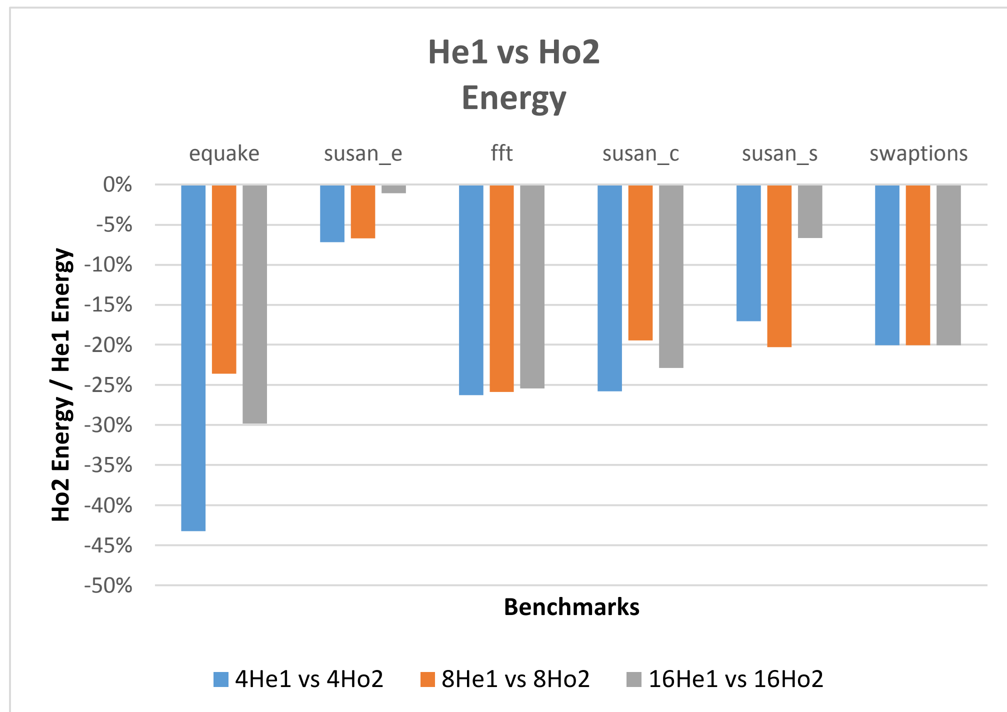


Figure 7.5 shows the comparison of energy consumption between configurations He1 and Ho2. These two configurations have an area ratio of one homogeneous DAP for each average heterogeneous DAPs, thus in the area parity analysis both have the same number of DAPs. As we can see in the figure, the homogeneous configurations have better energy savings in all scenarios. The Ho2 configuration is composed of medium DAPs that have smaller power usage than the average DAP of the He1. For instance, one access to the Ho2 reconfiguration memory has a power usage of 84.74mW, while an access on the average He1 DAP consumes 108.75mW, which, combined with the small performances gains, can explain the near 20% overhead in energy for the heterogeneous configuration.

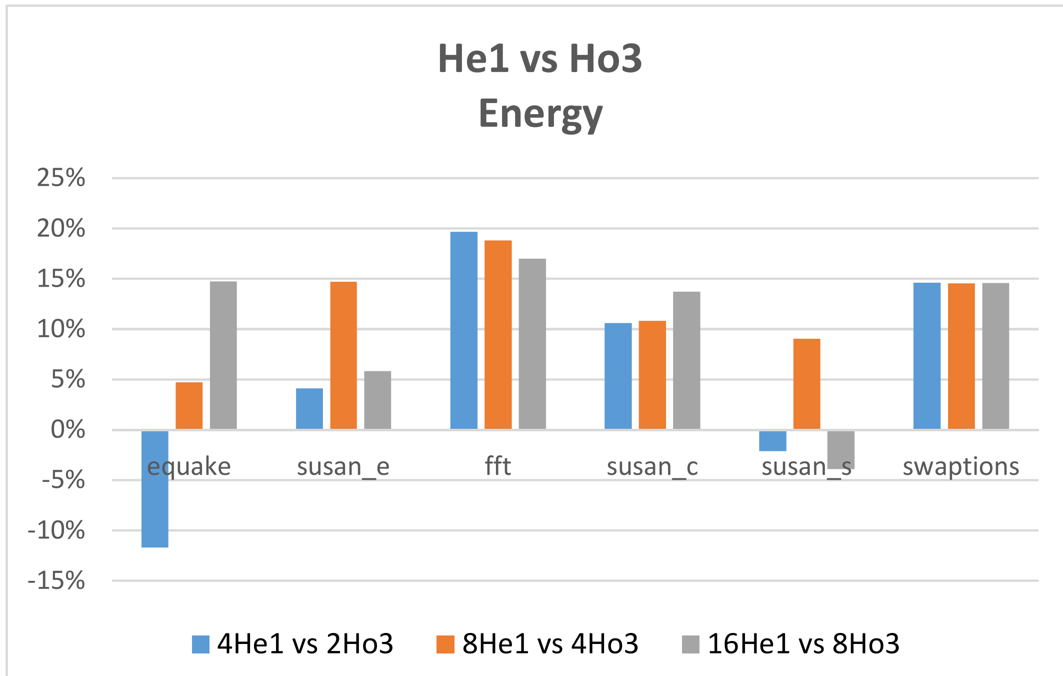
Figure 7.5: Energy comparison between He1 and Ho2. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS. The larger cores of the heterogeneous critically compromises the energy consumption of the system.



Source: the author

Figure 7.6 shows the comparison in energy consumption between configurations He1 and Ho3. The area ratio in this combination is of two average heterogeneous DAPs for each homogeneous DAP. The results show energy gains (about 15%) for the HARTMP configuration in most of the applications. The main reasons are that the heterogeneous version has better performance, thus achieving the end of execution faster (as seen in subsection 7.1.1.1); and the power usage of configuration Ho3 is quite high. Ho3 is a homogeneous configuration of large DAPs, in which a memory access for the reconfiguration cache consumes 244.08mW of power. Although this value is more than two times higher than the average He1 DAP (108.75mW) we must consider that the heterogeneous version still has double the number of DAPs than the homogeneous, which explains why energy gains are of roughly 15%.

Figure 7.6: Energy comparison between He1 and Ho3. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS. The larger caches in the Ho3 configuration critically compromises the energy consumption of the system.

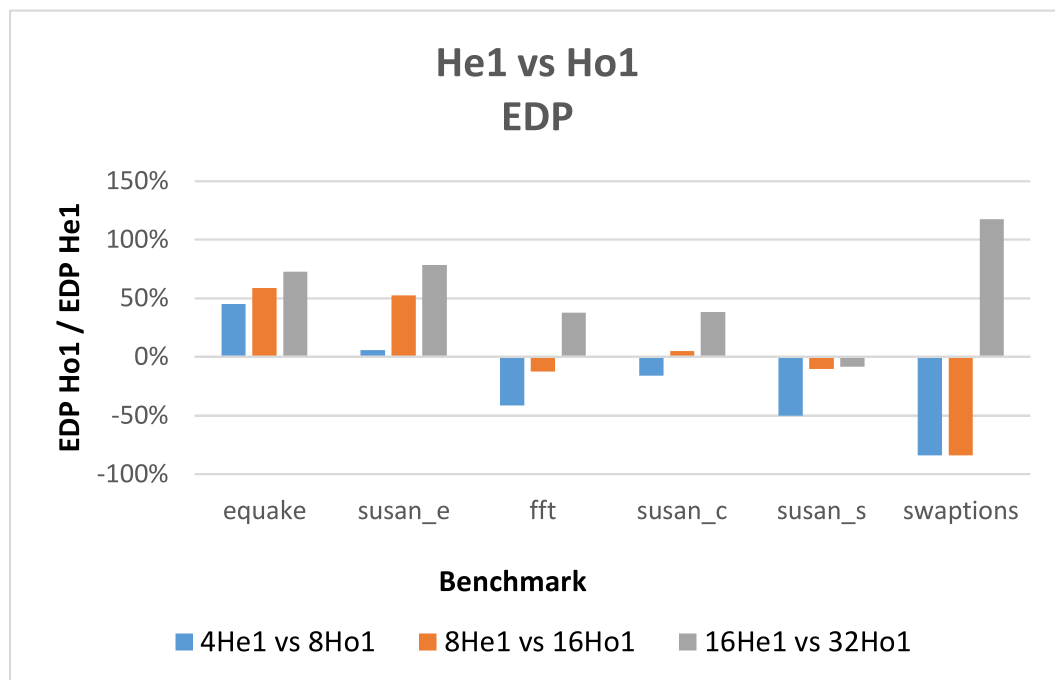


7.1.1.3 EDP Evaluation

EDP, or Energy-Delay Product, is a metric for evaluating the tradeoff between performance and energy consumption of a system. It is the product of the application execution time by the hardware energy consumption. Thus, the lowest is the EDP, the best is the performance per Watt of the circuit, because either it takes less time and less energy to execute the application, or one of these parameters is decreased to a point where it compensates the increase on the other. The following subsections will show the EDP comparison between the HARTMP and CReAMS configurations.

Figure 7.7 shows the comparison of EDP between configurations He1 and Ho1. As we have seen in subsections 7.1.1.1 and 7.1.1.2, in this combination, HARTMP has better energy consumption in all applications and better performance in most the cases. Thus, when making the product of these results, HARTMP shows good EDP improvements in almost every application. *Susan smoothing* is the only application in which CReAMS has better results. If we analyze the previous subsections, it can be observed that this application takes much advantage of the extra DAPs in Ho1 due to its ability to exploit TLP. However, it also has higher energy consumption as result of the same extra DAPs. The performance gains, though, compensates for the higher energy consumption, and as result CReAMS has better EDP than HARTMP in *susan smoothing*.

Figure 7.7: EDP comparison between He1 and Ho1. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS. Due to small energy consumption changes, the EDP is a reflection of the performance gains.

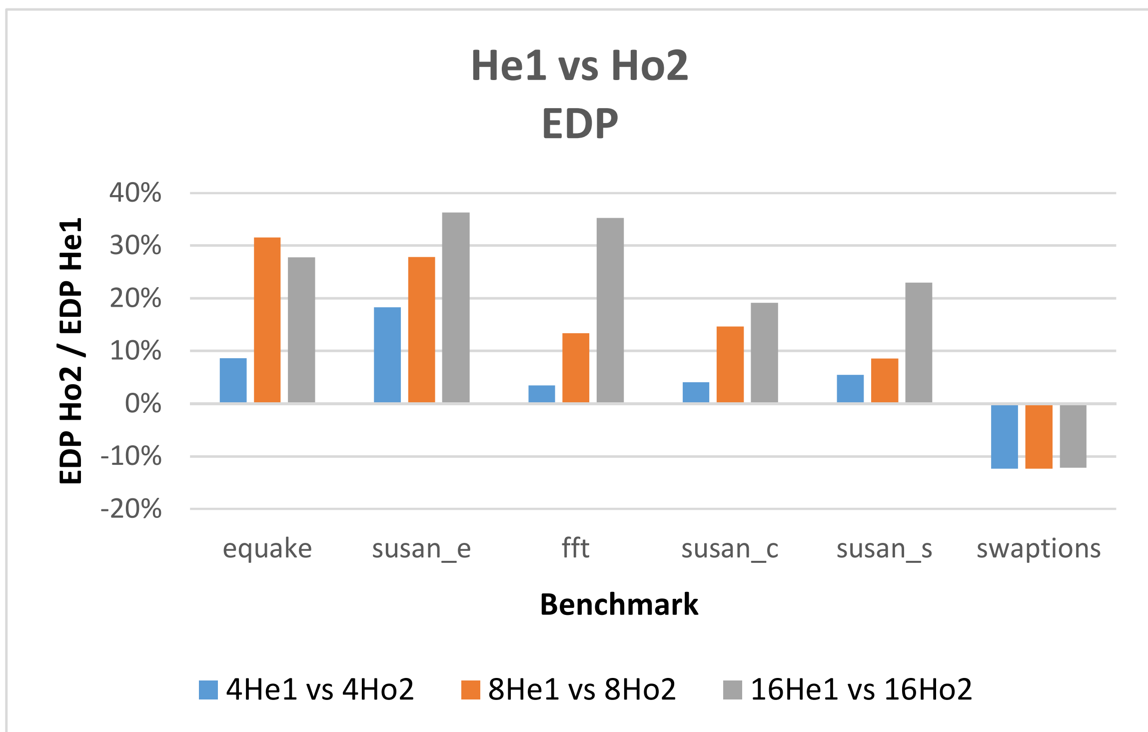


Source: the author

Figure 7.8 shows the comparison of EDP between configurations He1 and Ho2. As seen in previous subsections, in this scenario, HARTMP has better performance in almost every application. On the other hand, it also consumes more energy due to its larger DAPs with power hungry reconfiguration caches. However, when the EDP metric is evaluated, we can observe that the tradeoff between the performance gains and the increase of energy is positive for HARTMP. In other words, the extra energy consumption is compensated by the performance

gains. In this scenario, the only exception is the application *swaptions*. This application does not perform well with HARTMP, as its threads are extremely homogenous until 16 DAPs and the heterogeneous system has small DAPs to hold the execution time back. Furthermore, the application also consumes more energy when running on the heterogeneous environment, thus the bad EDP performance for HARTMP.

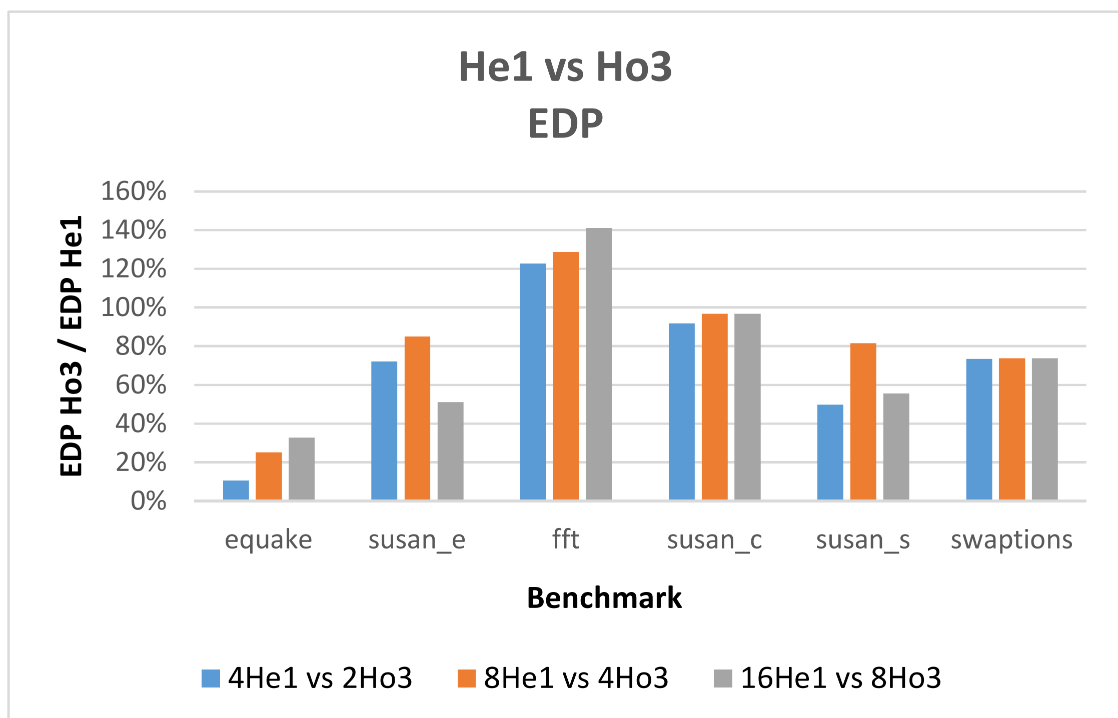
Figure 7.8: EDP comparison between He1 and Ho2. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS. This scenario shows that although the energy consumption is much higher in the heterogeneous version, the performance gains are enough to compensate it.



Source: the author

Finally, in the Figure 7.9 the comparison in EDP between configurations He1 and Ho3 is shown. As seen in previous subsections, in this combination the HARTMP version is superior in both energy consumption and performance, thus providing huge advantages in EDP. There is no real tradeoff to analyze in this case. EDP is up to 2.4 times better for HARTMP in the *FFT* application and at least 10% in *equake*, having an average of 75% EDP improvement.

Figure 7.9: EDP comparison between He1 and Ho3. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS.



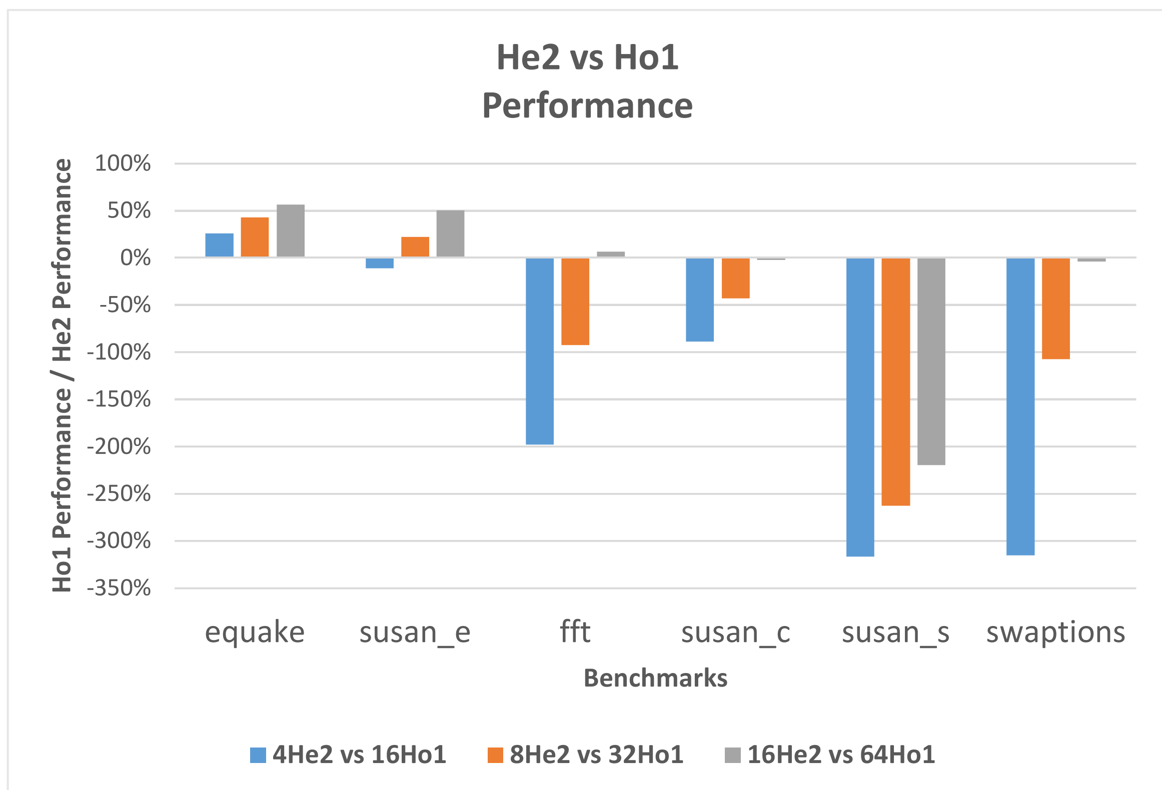
Source: the author

7.1.2 He2 Configurations

7.1.2.1 Performance Evaluation

Figure 7.10 shows the comparison between configurations He2 and Ho1. The average DAP of HARTMP He2 configuration is much larger than the one in He1. In fact, it is double the size. We have created this configuration in order to evaluate if having DAPs capable of greatly exploiting ILP would bring advantages in a heterogeneous system. The area ratio between configurations He2 and Ho1 is of four homogeneous DAPs for each average heterogeneous, thus CReAMS, in this scenario, can exploit much more TLP than HARTMP, but much less ILP. The results in the Figure 7.10 show performance improvements for HARTMP only in the applications that poorly exploit TLP, like *equake* and *susan edges*. For the applications that can exploit TLP, even in a reduced number of threads, the CReAMS configuration is clearly better,

Figure 7.10: Performance comparison between He2 and Ho1. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS.

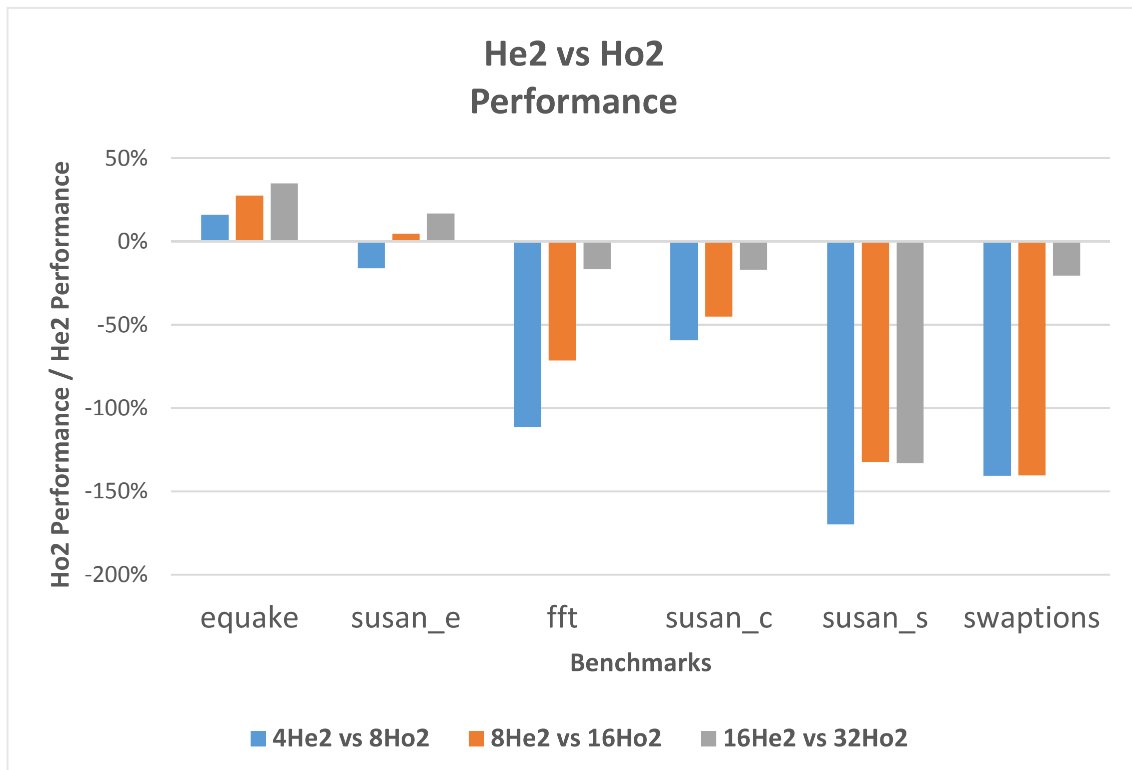


Source: the author

due to its high numbers of simpler DAPs. Even in *swaptions*, which the TLP limit is rapidly reached at 16 DAPs, CReAMS still has the advantage. However, it is possible to observe a clear trend in which the advantage of using the extra DAPs of CReAMS decreases as the number of DAPs grows. For instance, in the *FFT* application, the TLP limit is reached at 64 CReAMS DAPs and HARTMP has a slight advantage in performance.

Figure 7.11 shows the comparison between configurations He2 and Ho2. In this scenario, the area ratio between configurations is of two homogeneous DAPs for each average heterogeneous DAP. Again, we have very similar results from He2 vs Ho1 scenario, however the number of DAPs in the homogeneous configuration is now reduced. Thus, although CReAMS is still superior in applications with high TLP, the performance advantage of the homogeneous version decreases. In this combination, HARTMP is up to 35% better in the *equake* configuration, due to the efficient scheduling of the unbalanced threads of this application. On the other hand, CReAMS is up to 170% better in *susan smoothing*, as result of the extra DAPs in the homogeneous version and the highly balanced threads of the application.

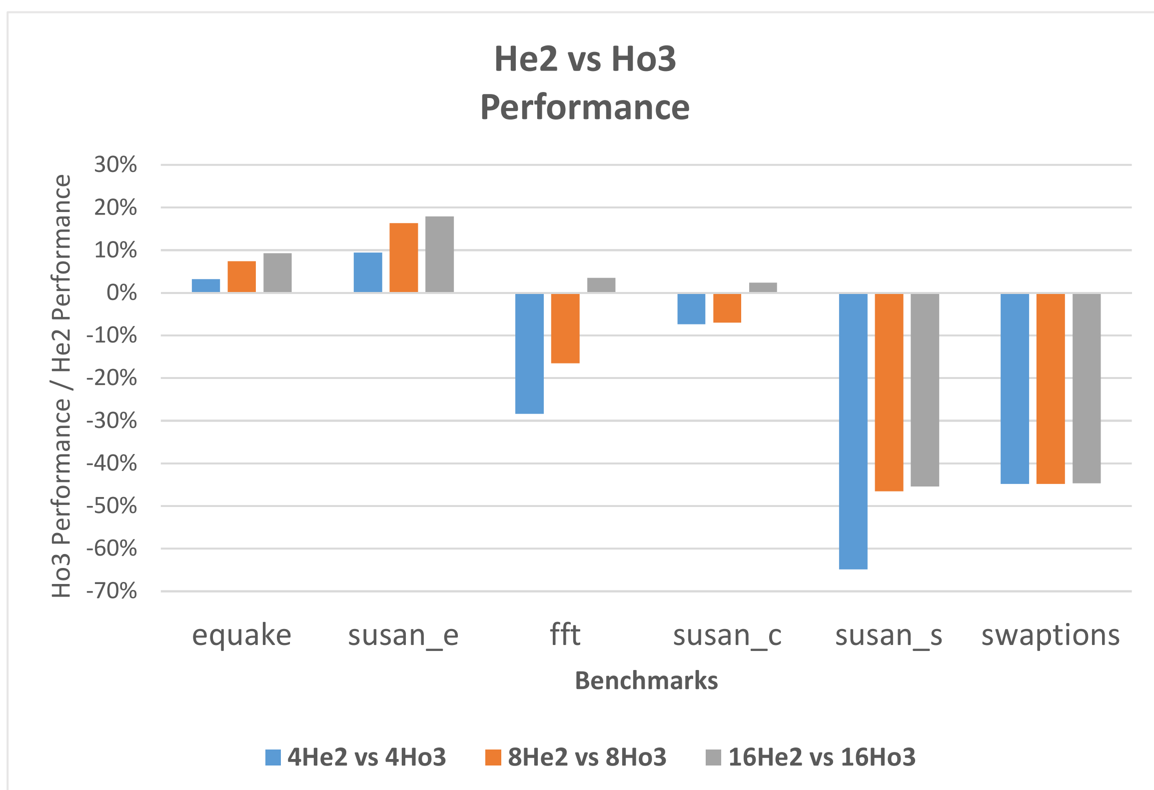
Figure 7.11: Performance comparison between He2 and Ho2. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS.



Source: the author

Finally, in the Figure 7.12, the comparison between configurations He2 and Ho3 is shown. In this combination, the area ratio is one homogeneous DAP for each average HARTMP DAP. This is possible because the large DAP of He2 is much larger than Ho3, then the average heterogeneous DAP has approximately the same size as a Ho3 DAP. Thus, both configurations should be able to exploit the same TLP, differing only from the capacity of exploiting ILP and the ability to allocate the threads efficiently. In this scenario, the homogeneous configuration is composed of only large DAPs, while the heterogeneous has smaller DAPs. This reflects in the results: CReAMS is superior in applications where the threads have good load balance as it does not have the small DAPs to hold the larger back: as all threads have the same load, the small DAPs from He2 take more time to finish their job, forcing the larger cores to wait. This is more evident in applications that demand for high ILP and TLP hardware, such as *susan smoothing*. This application can take advantage of all DAPs in the configurations and uses many resources inside the reconfigurable array, due to its big basic blocks (i.e.: more ILP is exposed). The small DAPs in HARTMP create a bottleneck for this application, as the threads have similar load and the ones that are running on the larger DAPs have to wait for the ones that are running

Figure 7.12: Performance comparison between He2 and Ho3. Bars over 0% means that the HARTMP version has better performance, while below 0% means advantages for CReAMS.



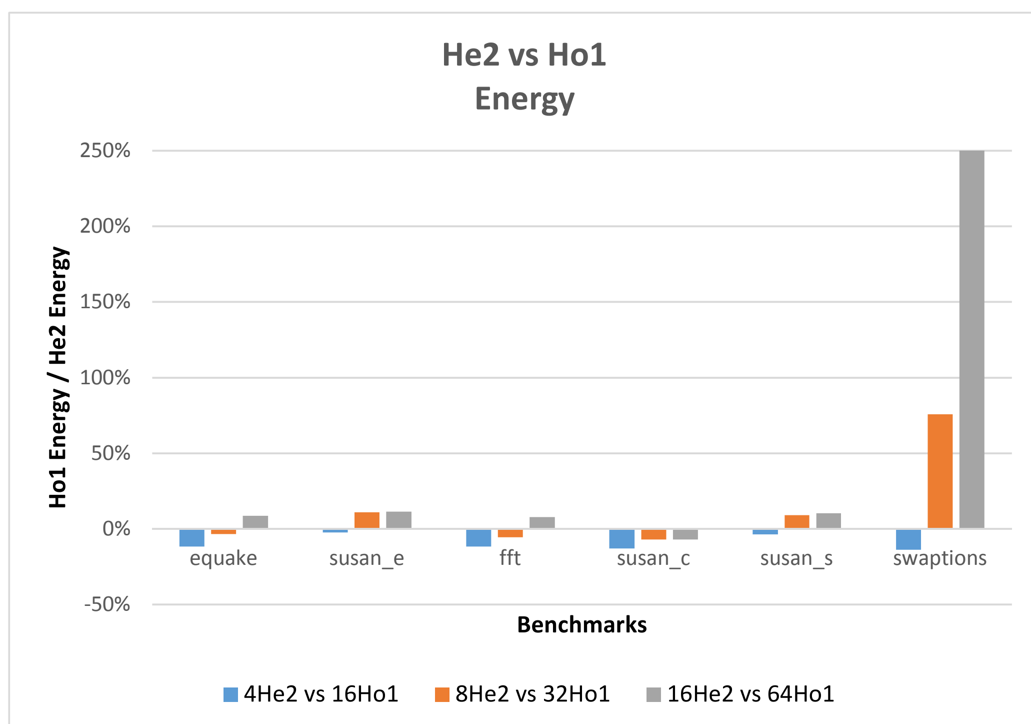
Source: the author

on the smaller ones. However, in applications with unbalanced threads, HARTMP performs better, as result of the efficient ILP exploitation in its larger DAPs.

7.1.2.2 Energy Evaluation

He2 is composed of very large DAPs. The small DAP in He2 is comparable in size with the medium DAP in He1 and the large DAP in He2 is twice the size of the large DAP in He1. Although this provides more room for exploiting ILP inside the reconfigurable unit, these large sizes also create a huge power overhead. Figure 7.13 shows the comparison in energy consumption between configurations He2 and Ho1. The area ratio in this combination is of four homogeneous DAPs for each average heterogeneous DAP. In this scenario, although configuration Ho1 has four times more DAPs than He2, the homogenous version still has energy advantages in many cases. The He2 configuration is too big and power hungry, thus energy consumption varies from -10% (10% smaller in CReAMS) to 10% (10% smaller in HARTMP) in most of the applications. The only case in which HARTMP has high energy

Figure 7.13: Energy comparison between He2 and Ho1. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS.

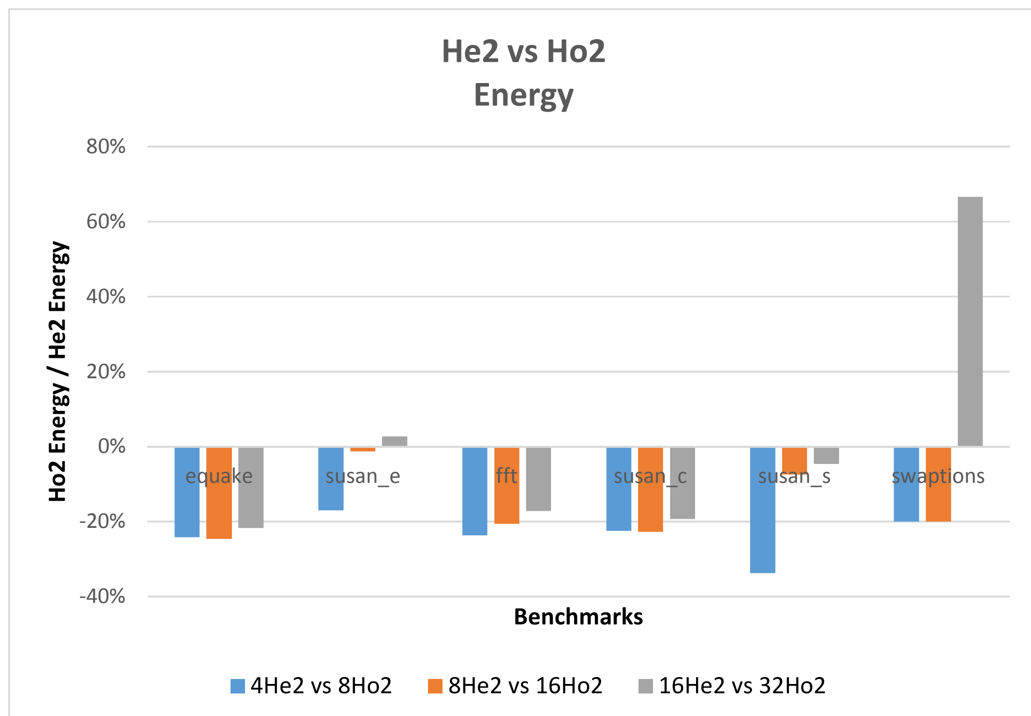


Source: the author

savings is on *swaptions* and this is due to the performance of this application, which reaches its TLP exploitation limits in 32 threads (as seen in subsection7.1.2.1).

Figure 7.14 shows the comparison in energy consumption between configurations He2 and Ho2. The area ratio in this combination is of two homogeneous DAPs for each average heterogeneous DAP. In this comparison, we can see how high the energy consumption of the large DAPs impact on HARTMP results: In most of the cases, HARTMP consumes 20% more energy than CReAMS. Only in applications that reach TLP limits the HARTMP performs better, as seen in *susan edges* and *swaptions* with 16-DAP He2 configuration.

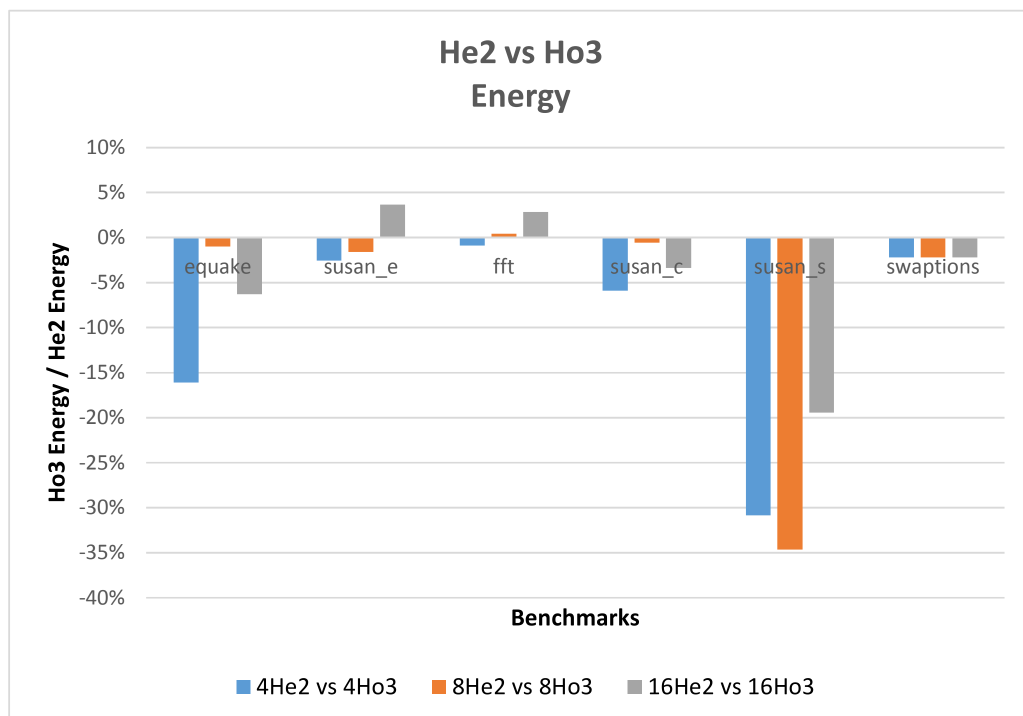
Figure 7.14: Energy comparison between He2 and Ho2. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS.



Source: the author

Finally, in the Figure 7.15 the comparison in energy consumption between configurations He2 and Ho3 is shown. In this combination, the area ratio is 1:1, which means that both configurations share the same number of DAPs for the same area. For this reason, in this scenario, the high average consumption of energy of the He2 configuration is made explicit, as CReAMS has smaller energy consumption in most of the cases. Again, only when thread unbalance is reached (like in 16-DAP *FFT* and *susan edges*) that HARTMP has advantage over CReAMS, as it can allocate the threads more efficiently over the DAPs.

Figure 7.15: Energy comparison between He2 and Ho3. Bars over 0% means that the HARTMP version has better energy consumption, while below 0% means advantages for CReAMS.



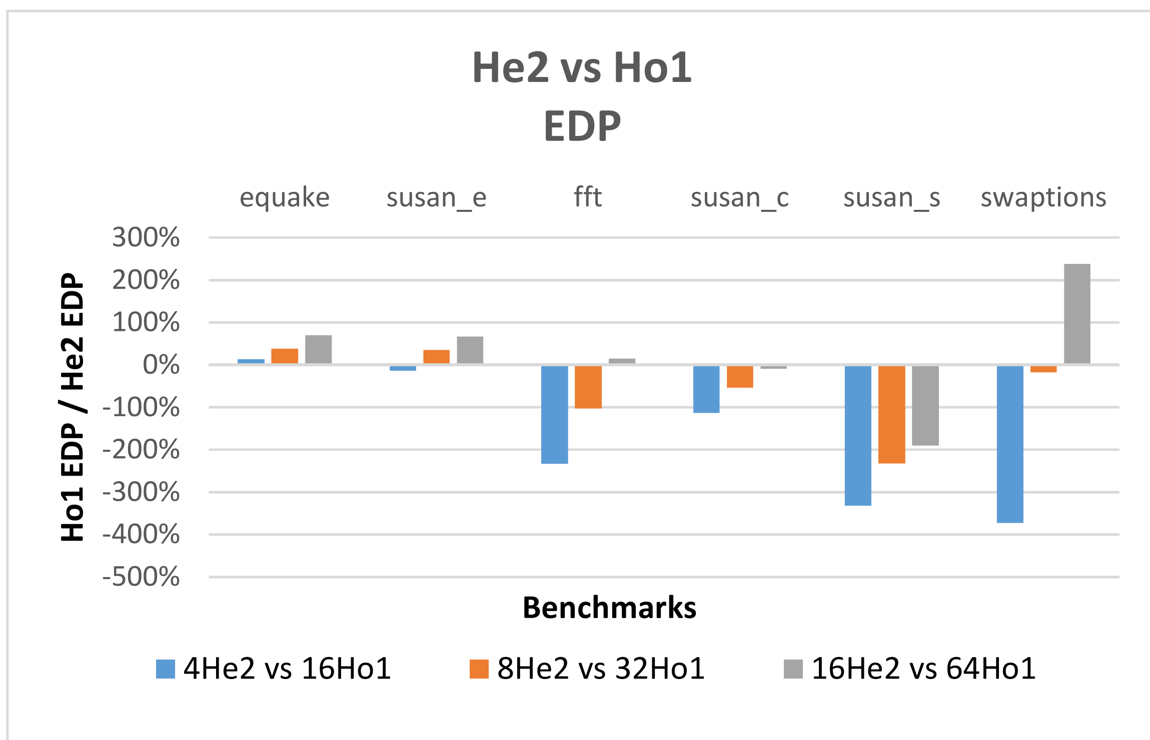
Source: the author

7.1.2.3 EDP Evaluation

In the previous subsections 7.1.2.1 and 7.1.2.2, we have seen that configuration He2 has small performance advantages and, in most of the cases, it consumes more energy as a result of its huge circuit complexity. In this subsection, we will analyze in which cases the tradeoff between complexity and energy consumption can still give some advantages to the HARTMP configuration.

Figure 7.16 shows the comparison of EDP between configurations He2 and Ho1. As seen in previous subsections, this combination brings good advantages for HARTMP when low TLP applications are executed; however, it also brings huge performance overhead in highly balanced applications, due to the ratio of DAPs in each system: HARTMP has 4x less DAPs than CReAMS. When energy is considered, HARTMP can be less power hungry than CReAMS, and this is mostly due to the reduced number of DAPs present in the heterogeneous configuration. Although, the savings are too small to affect the results in EDP. What we can

Figure 7.16: EDP comparison between He2 and Ho1. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS.

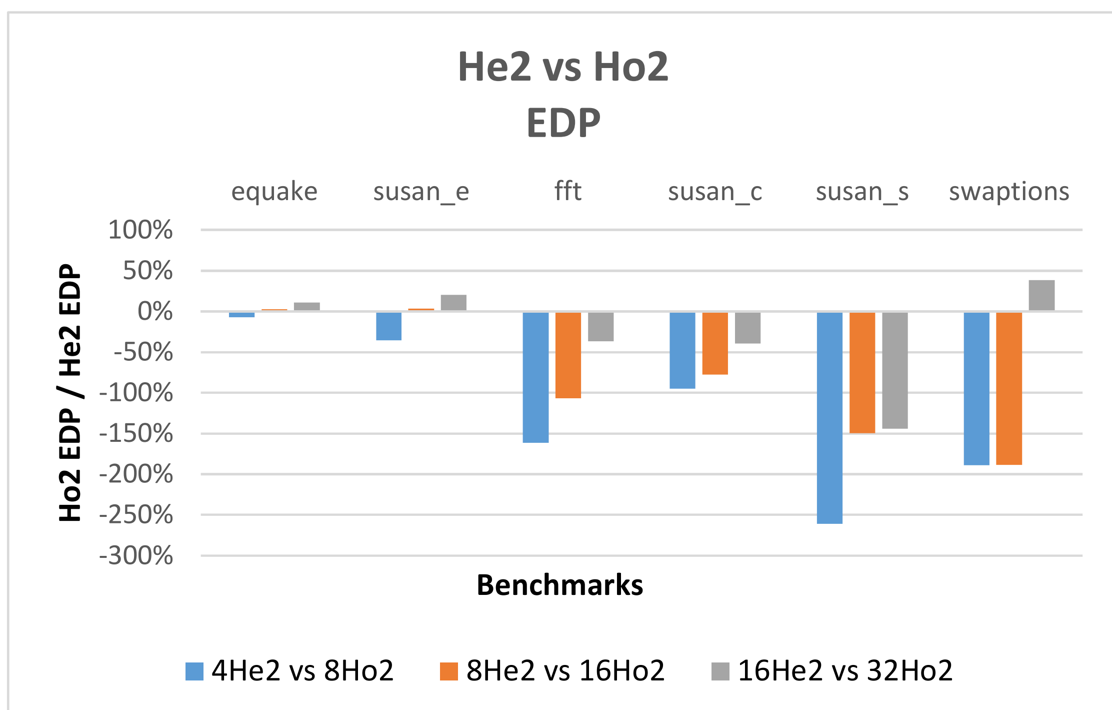


Source: the author

observe in the Figure 7.16 is a reflection of the results of performance seen in subsection 7.1.2.1 when we compare He2 and Ho1. *Equake* and *susan edges* are the most unbalanced applications and take advantage of the heterogeneous environment, where the highly sequential threads are allocated to larger DAPs. Furthermore, the energy consumption of HARTMP is, in general, smaller in this applications (as seen in subsection 7.1.2.2). Thus, EDP results for these applications show advantages for HARTMP. However, in applications that the threads have almost the same load, like in *susan smoothing* and *swaptions*, the small DAPs in great number can execute the application much faster and using less power. Thus, for these applications, CReAMS has better EDP.

Figure 7.17 shows the comparison of EDP between configurations He2 and Ho2. The heterogeneous version is much more power hungry than the homogeneous in this combination and consumes more energy in almost every scenario (subsection 7.1.2.2). However, He2 has some performance advantages over Ho2 in unbalanced applications. This tradeoff between higher power consumption and higher performance makes up when evaluating the EDP. As shown in the Figure 7.17, unbalanced applications, such as *equake* and *susan edges*, have better EDP in HARTMP. On the other hand, applications that can distribute better the load among

Figure 7.17: EDP comparison between He2 and Ho2. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS.



Source: the author

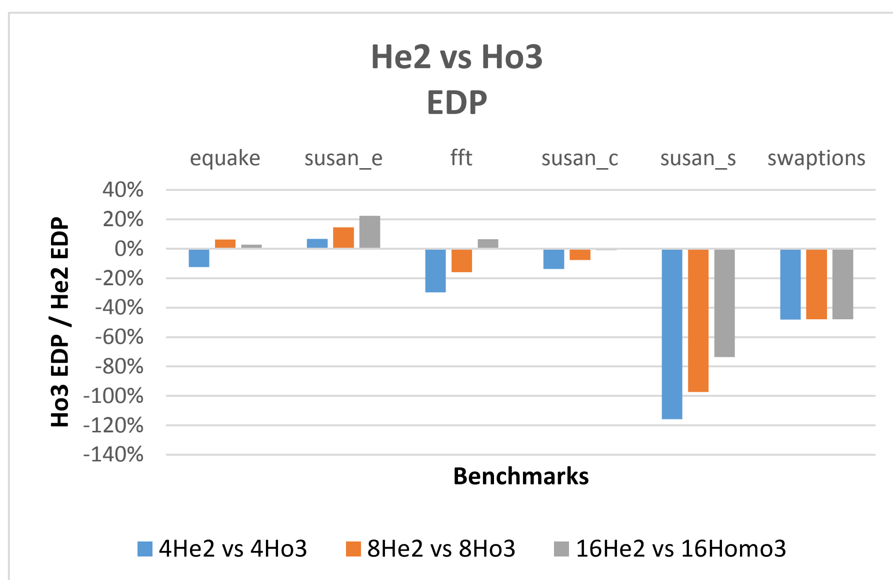
DAPs still have huge EDP advantages with CReAMS, as result of the extra DAPs present in the homogeneous configuration.

Finally, Figure 7.18 shows the comparison of EDP between configurations He2 and Ho3. In this combination, both system have the same number of processor DAPs, thus exploiting the same levels of thread parallelism. Still, HARTMP has advantages in unbalanced applications due to thread allocation and CReAMS in balanced applications, as result of having only large DAPs. When energy is considered, HARTMP is more power hungry in almost every scenario. However, when evaluating the tradeoff using the EDP metric, we see in the Figure 7.18 that the performance gains of HARTMP can still compensate the energy loss in unbalanced applications like *equake* and *susan edges.*

7.2 HARTMP Schedulers

In the section 7.1, we have shown results comparing the HARTMP heterogeneous system with the homogeneous CReAMS and we have demonstrated that HARTMP can outperform CReAMS in many scenarios and applications, especially in configuration He1, for both performance and energy consumption. In cases where HARTMP was worse than CReAMS in

Figure 7.18: EDP comparison between He2 and Ho3. Bars over 0% means that the HARTMP version has better EDP, while below 0% means advantages for CReAMS.



Source: the author

energy consumption, the heterogeneous configuration had advantages over performance, and vice versa. Thus, when considering the Energy-Delay Product (the tradeoff) between the two evaluated parameters, HARTMP was superior in almost every case.

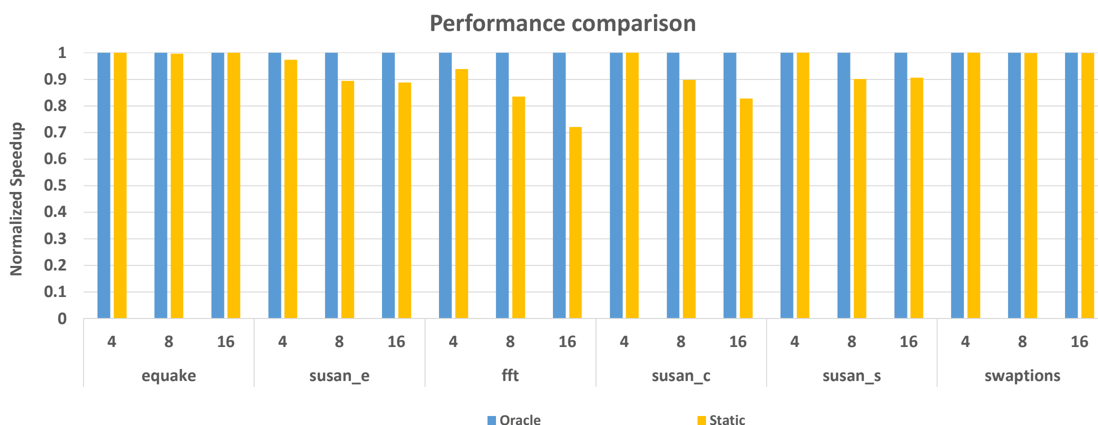
In this section, we will study the performance of the schedulers proposed in the chapter 5. We will use the He1 HARTMP configuration to extract the results of performance – in cycle count – of every scheduler. Then, we will normalize all the schedulers results according to the Oracle's results, which has always the best possible performance. From these data, we will, then, analyze and extract conclusions over the behavior and performance of each scheduler.

7.2.1 The Static Scheduler

The static scheduler is the simplest allocation algorithm used in this work. When a thread is created, it is allocated in the next free DAP and it executes in this DAP until the end of the application. This has two main advantages: it introduces no overhead in context change, if it is considered; and, most of the time, it naturally allocates the most sequential thread (the master thread) on a large DAP because of the order of priority of the processors.

As we can see in the Figure 7.19, the static scheduler has good performance in many applications. In *equake* the static scheduler has almost perfect performance because this application is not very parallelizable, which means that most of the work is done in the master

Figure 7.19: Performance comparison between the Oracle and the Static schedulers. The speedup is normalized to the Oracle's performance. The static scheduler can reach near Oracle performance on very unbalanced applications (due to most of the work being executed on a large core) and very balanced applications (as all cores execute the same load, no scheduling is necessary).



Source: the author

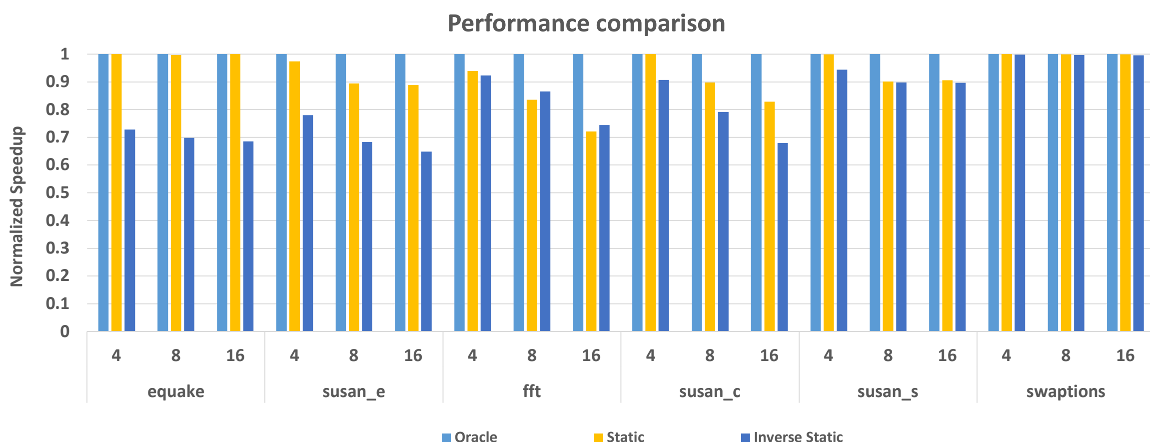
thread, which is always allocated in a large DAP. Thus, in this application, there is no need to change DAPs during the execution, as all the other threads have small loads. In *swaptions* the allocation is also perfect because this application is extremely well balanced. As all the threads perform the same work, changing DAP will not affect the applications performance, as the small DAP in the heterogeneous environment will always be the system bottleneck. *Susan smoothing* has similar behavior due to its balanced threads, reaching 90% of the Oracle's performance. However, applications such as *FFT*, which reaches the TLP exploitation limits faster, losing performance as the number of threads grows. In 16-DAPs, this scheduler reaches only 70% of the possible performance.

7.2.2 The Inverse Static Scheduler

As seen in the last subsection, the simple static scheduler can reach great performance even in very unbalanced applications such as *equake*. The pattern of allocation for the static scheduler follows a simple logic of allocating the firsts threads into the larger cores, so we analyzed the individual result files of the DAPs and found out that the firsts threads had bigger loads than the last ones in unbalanced applications. This explains why the static scheduler can have such good performance in both highly balanced and unbalanced applications. However, we wished to find out the real impact in performance of this allocation pattern. Thus, we have created a simple modification of the static scheduler: the inverse static scheduler. It works the same way as the simple static, but it allocates the first created threads (including the master thread) in the small DAPs, then medium and lastly the large.

As we can observe in the Figure 7.20, the inverse static suffers great performance loss in the unbalanced applications. The performance of *equake* falls from 100% the Oracle`s in the static to only 70% in the inverse static. Now, the master thread, which does most of the job, is entirely executed in a small DAP, creating a 30% overhead in performance. Other applications, such as *susan edges* and *susan corners*, also have performance decrease, due to their first created threads, which have heavier loads than the last. The only applications that have almost

Figure 7.20: Performance comparison between the Oracle, the Static and the Inverse Static schedulers. The speedup is normalized to the Oracle's performance. The inverse statics shows the impact of allocating the thread 0 into a small core.



Source: the author

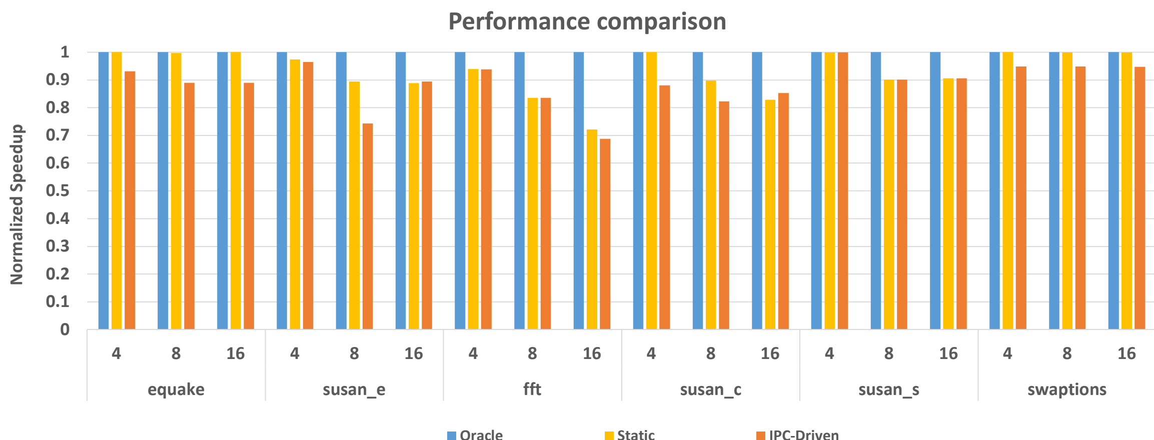
no performance change are *susan smoothing* and *swaptions*, corroborating to the previous conclusions: these applications are so well balanced that it does not matter where they execute, the small DAPs in HARTMP will always be their bottleneck.

7.2.3 The IC-Driven Scheduler

The IC (Instruction Count)-Driven scheduler was the first attempt to create a real scheduler based on data from the threads to allocate jobs efficiently through the heterogeneous DAPs. Although the static scheduler has great performance for some of the applications, a heterogeneous system still needs an efficient allocation algorithm to exploit fully its benefits. Thus, we have used, from (RUTZIG, 2012), a scheduler that sorts threads in DAPs through one of the most natural metrics one could choose: the number of instructions each thread executes. In the section 5.4, we showed that this algorithm simply counts the number of instructions a thread has executed in between two barriers to decide the allocation.

Figure 7.21 shows the performance of the IC-Driven Scheduler compared to the Oracle and the static. We can observe that this algorithm has lower or equal performance to the static in every scenario (but in one case of *susan edges*). There are two explanations for this behavior. First, this scheduler uses an information from the past to decide the future. In other words, it

Figure 7.21: Performance comparison between the Oracle, the Static and the IC-Driven schedulers. The speedup is normalized to the Oracle's performance. The IC-Driven is a first real scheduling attempt and showed the importance of using a good metric for allocation.



Source: the author

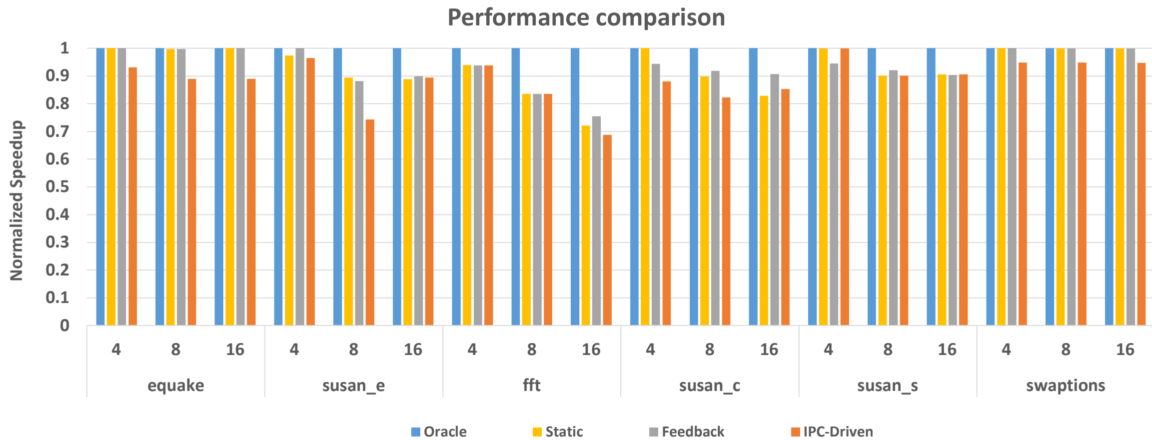
uses the number of instructions from the barrier it has just executed to decide the allocation of the next barrier. Thus, if the thread changes its behavior, then the algorithm allocates it wrongly. Another reason for the poor performance is the metric used to determine the load in a thread. The way each DAP in a HARTMP system exploits ILP is through the reconfigurable array. However, it is not every instruction that is executed in the reconfigurable unit, but just the ones supported by the array and those already saved in a configuration (a basic block). Thus, a thread might run many instructions, but if they are not supported by the array, then they are always executed in the SPARC-V8 processor.

7.2.4 The Feedback Scheduler

The IC-Driven scheduler has demonstrated poor performance over both the Oracle and the Static schedulers. To create a better scheduling algorithm for the HARTMP system, we have worked on the known problems of the IC-Driven algorithm: using information from the past to predict the future and using the thread instruction count as metric for allocation. Firstly, we have engaged in the simplest of the problems, which is the metric allocation, by using hardware counters to feedback the algorithm with information from inside the reconfigurable unit. Thus, the new algorithm, called Feedback scheduler, is the same as the IC-Driven, but it uses the number of instructions executed in the reconfigurable logic. In other words, it takes into account only the instructions that were accelerated by the heterogeneous hardware.

Figure 7.22 shows the performance comparison between the Oracle, Static, Feedback and IC-Driven schedulers. We can see that the new Feedback scheduler has better performance than the IC-Driven in almost every scenario (losing only in *susan smoothing*), which corroborates with the assumption that using the instruction count inside the array is a much more efficient metric than the total number of instructions. Furthermore, the Feedback scheduler has better performance than the static in applications such as *susan edges*, *fft* and *susan corners,* which are applications with unbalanced threads when the number of DAPs increases (the master thread does not do most of the job).

Figure 7.22: Performance comparison between the Oracle, the Static the IC-Driven and the Feedback schedulers. The speedup is normalized to the Oracle's performance. The Feedback reaches near Oracle performance as the static do and has better overall performance in other applications.



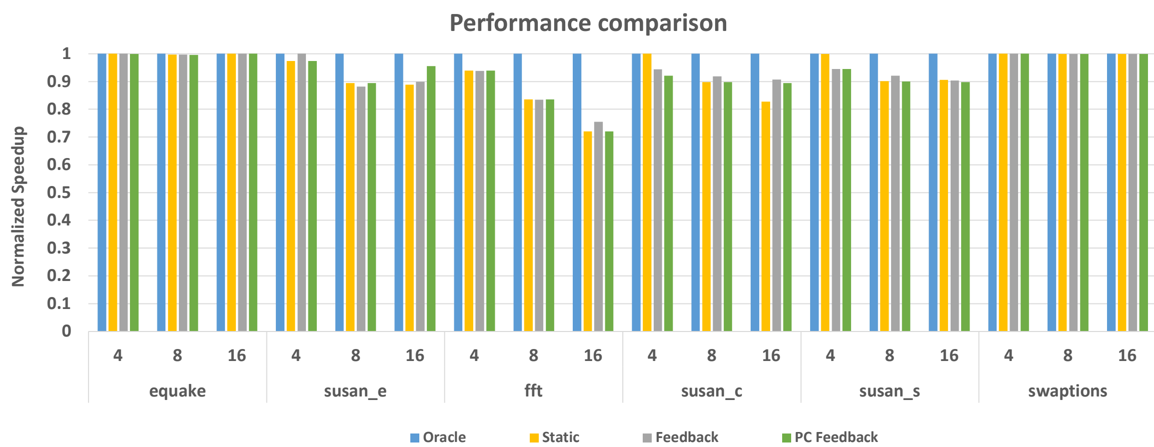
Source: the author

7.2.5 The PC-Feedback Scheduler

To engage the second problem of the IC-Driven scheduler (using information from the past to instantly predict the future), we have proposed one more algorithm: the PC-Feedback scheduler. This scheduler stores the PC of barriers and its array instruction count every time the application goes through the barrier. Next time this barrier is to be executed (the PC is reached), the scheduler will know the behavior of that barrier and uses this information to correctly allocate the threads. Thus, this scheduler only changes DAPs if the thread goes through a barrier at least twice, as the first time it does not have information about the barrier to perform a thread swap.

Figure 7.23 shows the performance comparison of the Oracle, Static, Feedback and PC-Feedback scheduler. The PC-Feedback has better performance in applications that execute the same instructions many times: the more a barrier is found, the more this scheduler allocates rightly. In *FFT* the PC-Feedback has exactly the same performance as the static algorithm, because *FFT* never executes the same barrier twice, thus it never changes the allocation of the reads. S*usan edges* is the application in which this algorithm has its better performance over the other, reaching 95% of the Oracle's performance in a 16-DAP system.

Figure 7.23: Performance comparison between the Oracle, Static, Feedback, and PC-Feedback schedulers. The speedup is normalized to the Oracle's performance. The Feedback-PC shows better performance in some applications, however the improvement over the Feedback is only marginal.



8 CONCLUSIONS

Conclusion on this work might be drawn both for the best usages of a HARTMP system and on the schedulers designed to work on this processor. We will list an abstract of the analysis made in these two main studies in the following sections.

8.1 HARTMP usage

In this work, we have proposed a novel heterogeneous reconfigurable multicore system that is able to accelerate any kind of application in a completely transparent manner: the HARTMP. HARTMP has demonstrated to have many advantages on its homogenous counterpart (CReAMS) in both performance and energy consumption. We have made several analysis in two different versions of HARTMP against three distinct configurations of CReAMS and shown that if the heterogeneous system is correctly scaled it can be much more efficient when considering the tradeoffs of performance gains with energy consumption.

The main conclusions extracted from the analysis of the experiments results are the following:

1. A HARTMP system is usually faster when executing applications with highly unbalanced threads. The schedulers are able to efficiently allocate the most sequential threads into the larger DAPs, while keeping the most parallel ones in the smaller DAPs.
2. A HARTMP system can greatly decrease the performance of highly balanced applications. This is due to the small DAPs present in this heterogeneous version. When a thread is executing and reaches a barrier, it must stop and wait for all the other threads to synchronize. This means that if an application has all threads with the same, the smaller DAPs in heterogeneous system will hold back the larger ones, as the threads running on it will take more time to reach the barriers.
3. When the heterogeneous system has less DAPs than the homogeneous, the applications that exploit much TLP will have poor performance on the heterogeneous processor. This is a known problem for systems that exchange number of DAPs for complexity. When HARTMP has more DAPs than CReAMS,

than it has advantages in every scenario, as result of efficiently exploiting ILP and having extra DAPs to exploit TLP.

4. Relating to the configurations proposed; He1 proves to be much more balanced than He2. This suggests that investing in simpler DAPs, but in high quantities, brings more advantages than having few, but too complex DAPs. He2 DAPs proved to be too complex, big and power hungry, do not compensating for its performance gains.
5. HARTMP has also shown great EDP measurements. In He1 configuration, it has shown advantages in all the applications.

8.2 Schedulers

This work also proposes six different scheduling algorithms. We have presented the performance of each of this schedulers running on a HARTMP configuration. Following, we show the conclusions that can be drawn of these algorithms:

1. The Oracle scheduler is a predictive algorithm that always outputs the best allocation for performance. It was used to prove the potential of our heterogeneous system. However, it needs to execute all the threads in all the DAPs and them analyze all the possible combinations of results, looking for the best one. This is not viable in a real system, thus this scheduler is for proof of concept only.
2. The Static scheduler proved to be extremely efficient on two classes of applications: those with perfect load balance between threads and those with very poor balance. The first class is a very restrict set of applications. Normally, a software developer is not able to perfectly parallelize its application. However, the second class of applications is extremely common. The simplicity of this scheduler shows that it can be used in many simple systems that do not require complex thread allocation.
3. The inverse static scheduler was successfully used to prove the origins of the good performance of the normal static scheduler. However, due to its poor performance in all applications, it should not be used in a HARTMP system.
4. The IC-Driven Scheduler was a first attempt to create a real scheduler for the HARTMP system. It has very poor performance even when compared to the static,

however, it was useful to point the main characteristics a scheduler should have on a HARTMP system.

5. The Feedback scheduler has a better approach than the IC-Driven as it tackles the allocation problem with specific HARTMP characteristics. Instead of using general data from the processor, it uses data that comes directly from the reconfigurable units, which are the circuitry responsible for creating heterogeneity in our system. This scheduler has better performance than the static in most of the scenarios.
6. The PC-Feedback is an attempt to improve the Feedback scheduler by storing the characteristics of each barrier. It has better performance than the original Feedback allocator does, however it is a marginal difference.

9 FUTURE WORK

Many are the possible branches from exploiting more performance and energy gains from HARTMP. These vary from creating new and more efficient scheduling algorithms to changing the reconfigurable unit organization. In the next sections, we will present some ideas with great potential of becoming new points of research for our reconfigurable system.

9.1 A Big Little HARTMP

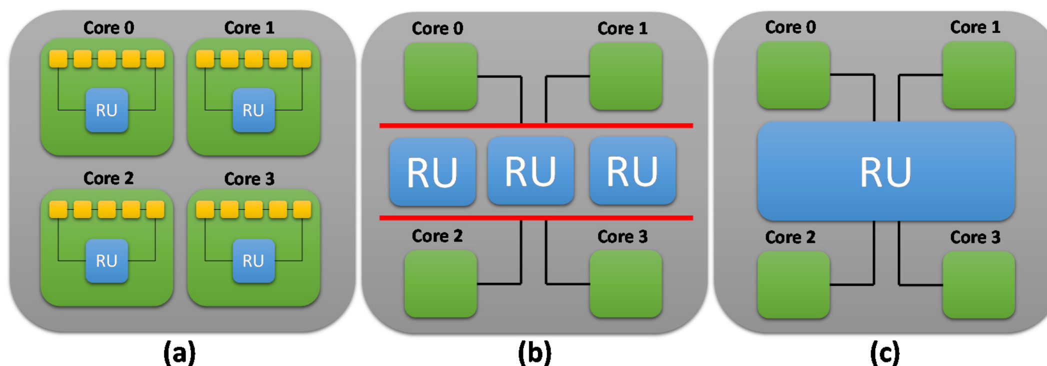
In our methodology, we have created configurations with small, medium and large DAPs in order to exploit more heterogeneity from the applications. However, in applications with highly unbalanced loads – which are quite common – the medium DAPs are usually underused and could be easily exchanged with more simpler and less power hungry DAPs to exploit more TLP. An analysis could be made in order to verify which is the tradeoff between keeping the medium DAPs and exchanging it by more small DAPs (or even more large DAPs).

Furthermore, it is a tendency of the industry to adopt a large and a small cores when applying heterogeneity in same ISA processors. The idea is to use the same strategy as the ARM big.LITTLE, migrating threads from one kind of DAP to another. This could also be used for applications in which energy is the main constraint, swapping threads from DAPs with the main objective of reducing power usage.

9.2 Sharing the reconfigurable unit

Figure 9.1 shows examples of how the reconfigurable logic can be shared between the DAPs of a processor. In the Figure 9.1(a), each processor DAP has its own reconfigurable fabric (CReAMS and HARTMP strategy). In this model, heterogeneity can be easily achieved by designing different reconfigurable fabrics for each DAP, i.e. each DAP will have a distinct number of units available in the reconfigurable logic, proving different levels of instruction parallelism exploitation. However, this model adds the burden of allocating the right threads to the right DAPs, which is a complex task to be solved and wished to be avoided.

Figure 9.1: Examples of shared reconfigurable units. (a) Every core has its own reconfigurable datapath. (b) Multiple reconfigurable fabrics are shared between many cores. (c) One reconfigurable logic is shared between many cores.



Source: the author

In the Figure 9.1(b), the DAPs do not have a fixed reconfigurable logic, but they have access to a bank of reconfigurable fabrics that can be used to accelerate the application at hand. To reach heterogeneity, each reconfigurable block in the bank can be designed in distinct sizes, being able to exploit different levels of instruction parallelism. In this approach, the task of deciding the best reconfigurable fabric to be used can be assigned to the DAP – or a special controller – as at this point, the system knows the size of the basic block that needs to be scheduled for a reconfigurable fabric.

Finally, in the Figure 9.1(c), a big reconfigurable fabric is shared among all the DAPs. A DAP can individually use this fabric at a time. However, it is more efficient to create a system in which the fabric is dynamically divided between the DAPs, accordingly to the threads needs. In this approach, there is no need for a scheduler at all, as the reconfigurable logic is virtually heterogeneous. Furthermore, as the threads share the same memory space, the DAPs can even share their reconfigurable contexts between themselves. For instance, if an application executes a “for loop” in parallel, all the threads will be executing the same basic block, but with distinct input data. Thus, all the DAPs can load the same basic block into their reconfigurable space.

There are many possibilities for research based on the sharing of reconfigurable datapaths. Some are highlighted bellow:

- **Simultaneous multithreading support:** For some applications, the upper bound for ILP exploitation is small, even when using reconfigurable logic, leaving many units in the datapath unused and wasting resources. One could analyze the viability of using SMT, to optimize the usage of the reconfigurable

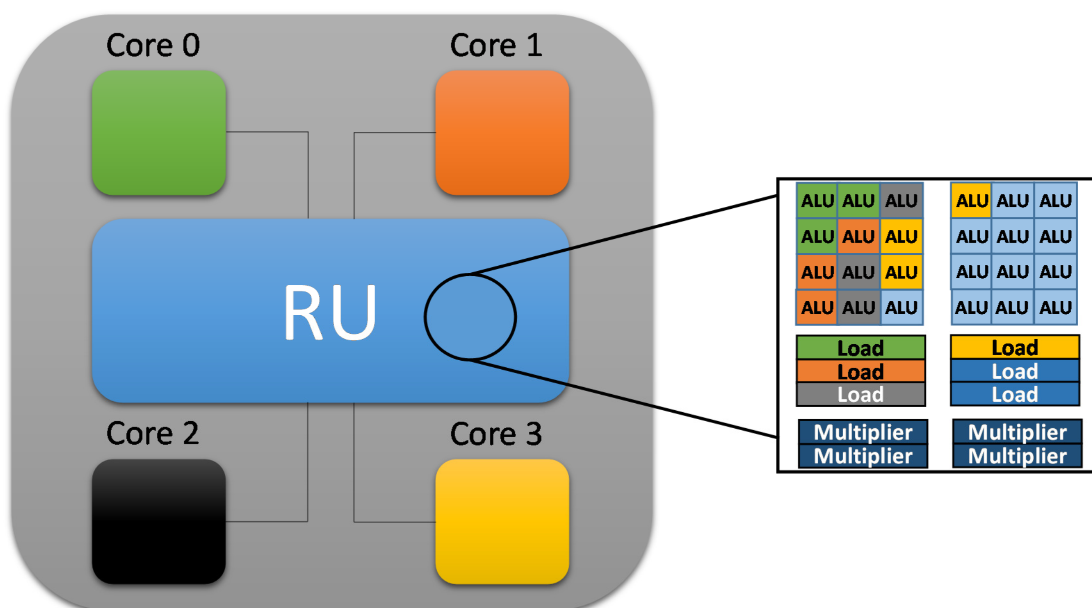
datapath. In the Figure 9.2, all the DAPs are assigning their tasks to run on the same reconfigurable datapath, thus four threads are running simultaneously, without distinction for the reconfigurable block.

- **Dynamic reconfigurable fabric:** A shared reconfigurable block can support many heterogeneous threads by either applying SMT, as in the Figure 9.2, or by dynamically reserving a space in the datapath for each demanding DAP. Figure 9.3 illustrates this concept.

- **Sharing the reconfigurable fabric:** In the Figure 9.1, three models for sharing the reconfigurable datapath are shown. However, other models can also be evaluated, e.g. sharing a reconfigurable fabric between each two DAPs or mixing DAPs with private reconfigurable logic with DAPs that share the reconfigurability. There is a huge space for research on the tradeoffs of each proposed model.

- **Data communication and synchronization:** Determining how the communication between the reconfigurable fabric and the many DAPs of the

Figure 9.2: SMT in a shared reconfigurable logic. Each core assigns its threads to execute together on the datapath

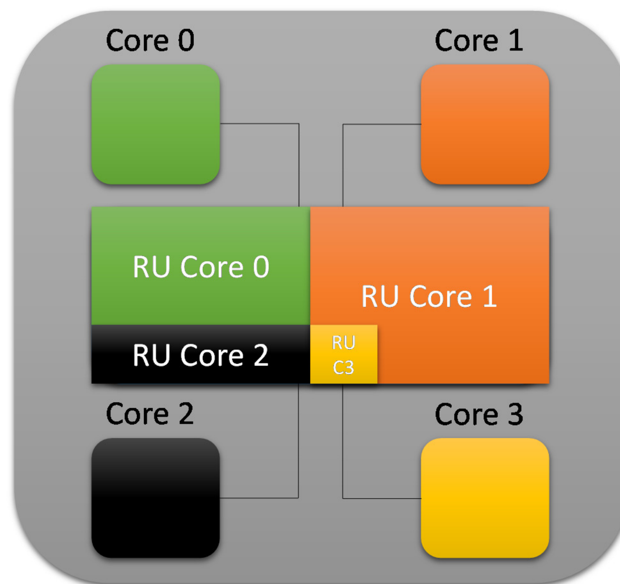


Source: the author

system is another challenge of this project. Some alternatives include the use of monolithic or segmented buses, Network on Chip (NoC) and a ring network. A deep analysis of the best solution is important.

- **Sharing the reconfigurable context:** The instructions that one thread executes in a DAP can also be executed by the others (like in a parallel loop). This suggests that all DAPs in the processor can share the same context memory for stored translated basic blocks. The advantages of sharing the context include: avoiding translating the same basic block many times; having a centralized reconfigurable cache eliminates the need of keeping cache coherence when swapping threads between DAPs; centralized reconfigurable cache can be placed near (have a dedicated bus) the reconfigurable fabric, simplifying the communication. The impact of sharing the reconfigurable context between DAPs and having private caches must be analyzed.

Figure 9.3: Cores dynamically request private spaces in a shared reconfigurable datapath.



Source: the author

9.3 Exploiting new Schedulers

Scheduling threads in any heterogeneous environment is not a trivial task and it is not different in HARTMP. We have evaluated several schedulers in our system, but none of them has reached the desired performance for all the applications. A general scheduler, capable of reaching near-oracle performance for all kinds of applications without inserting too much complexity in the system is most desired.

There are many kinds of schedulers that could be analyzed. For instance, phase schedulers, in which each phase determines a behavior for the scheduler. Genetic algorithm based schedulers, which use an evolution algorithm to converge to the best performance. Other are the self-adaptive algorithms that are online schedulers capable of matching computational demands of threads to cores.

Another possibility is to adjust the existing algorithms to, instead of optimizing performance, optimize energy consumption, or even Energy-Delay Product. This can be very useful in environments where energy consumption is a stronger constraint than an execution deadline, or even when a good tradeoff between the two are necessary.

REFERENCES

ACKLAND, B. et al. A single-chip 1.6 billion 16-b MAC/s multiprocessor DSP. IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE, San Diego, 1999. **Proceedings…** [S.L.: s.n.], 1999, p.537-540

BALAKRISHNAN, S. et al. The impact of performance asymmetry in emerging multicore architectures. INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, Wisconsin, 2005. **Proceedings…** New York: IEEE, 2005, p. 506–517

BECCHI, M.; CROWLEY, P. Dynamic thread assignment on heterogeneous multiprocessor architectures. CONFERENCE ON COMPUTING FRONTIERS, Ischia, 2007. **Proceedings…** New York: ACM, 2006, p. 29-40

BECK, A. C. S. et al. Transparent reconfigurable acceleration for heterogeneous embedded applications. DESING, AUTOMATION AND TEST IN EUROPE (DATE), Munich, 2008. **Proceedings…** New York: IEEE, 2008, p. 1208-1213

BECK, A. C. S.; CARRO, L. **Dynamic Reconfigurable Architectures and Transparent Optimization Techniques**. 1 ed., New York, NY: Springer New York, 2010.

BECK, A. C. S.; LANG LISBÔA, C. A.; CARRO, L. **Adaptable Embedded Systems**. 1. ed. New York, NY: Springer New York, 2013.

BECK, A. C. S.; RUTZIG, M. B.; CARRO, L. A transparent and adaptive reconfigurable system. **Microprocessors and Microsystems**, v. 38, n. 5, p. 509–524, jul. 2014.

BIENIA, C. et al. The PARSEC Benchmark Suite : Characterization and Architectural Implications. INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, Toronto, 2008. **Proceedings…** New York: ACM 2008, p. 72–81

big.LITTLE Technology. Disponível em: <https://www.arm.com/products/processors/technologies/biglittleprocessing.php>. Acesso em: 4 jan. 2016.

BINGFENG MEI et al. Architecture Exploration for a Reconfigurable Architecture Template. **IEEE Design and Test of Computers**, v. 22, n. 2, p. 90–101, fev. 2005.

BUTENHOF, D. R. **Programming with POSIX Threads**. 1. ed., Nashua, NH: Addison-Wesley, 1997.

CHAPMAN, B.; JOST, G.; VAN DER PAS, R. **Using OpenMP: Portable Shared Memory Parallel Programming**. 1 ed., Cambridge, MA: The MIT Press, 2008.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software, **ACM Computing Surveys (csuR)**, v. 34, n. 2, p. 171-210, 2002.

DIXIT, K. M. The SPEC Benchmarks. In: **Computer Benchmarks**. v. 17, p. 149–163, 1993.

DORTA, A. J. et al. The OpenMP Source Code Repository. EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING, Lugano, 2005. **Proceedings…** New York: IEEE, 2005, p. 244-250

GUTHAUS, M. R. et al. MiBench : A free, commercially representative embedded benchmark suite. INTERNATIONAL WORKSHOP ON WORKLOAD CHARACTERIZATION, Austin, 2001. **Proceedings…** New York: IEEE, 2001 p. 3–14

HENESSY, J. L.; DAVID A. PATTERSON. **Computer Architecture: A Quantitative**

Approach. 5. ed. Morgan Kaufmann, 2011.

JEFF, B. **big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling**. p. 1–13, 2013.

KUMAR, R. et al. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURES (MICRO), San Diego, 2003. **Proceedings…** New York: IEEE, 2003, p. 81-92.

LEE, J. et al. Thread tailor. **ACM SIGARCH Computer Architecture News**, v. 38, n. 3, p. 270, 19 jun. 2010.

LI, S. et al. CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN**,** San Jose, 2011. **Proceedings…** New York: IEEE Press, 2011, p. 694-701

LYSECKY, R.; STITT, G.; VAHID, F. Warp Processors **ACM Transactions on Design Automation of Electronic Systems**, 2006.

MAGNUSSON, P. S. et al. Simics: A full system simulation platform. **Computer**, v. 35, n. 2, p. 50–58, 2002.

MOGUL, J. C. et al. Using asymmetric single-ISA CMPs to save energy on operating systems. **IEEE Micro**, v. 28, n. 3, p. 26–41, 2008.

OLUKOTUN, K.; HAMMOND, L. The future of microprocessors. **Queue**, v. 3, n. 7, p. 26, set. 2005.

RUTZIG, M. B. **A transparent and energy aware reconfigurable multiprocessor platform for efficient ILP and TLP exploitation**. Tese(Doutorado). Universidade Federal do Rio Grande do Sul, Porto Alegre, 2012.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Simultaneous ILP and TLP Exploitation. DESIGN, AUTOMATION & TEST IN EUROPE CONFERENCE & EXIBITION (DATE)**,** Grenoble, 2013. **Proceedings…** New York: IEEE, 2013, p. 1559-1564.

SHELEPOV, D.; ALCAIDE, J. S. HASS: a scheduler for heterogeneous multicore systems. **Operating Systems**, v. 43, n. 2, p. 66–75, 2009.

SOUZA, J. D. **Evaluation of heterogeneous CReAMS**. 2014. 78 f. TCC(Graduação) - Curso de Engenharia de Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2014.

SRINIVASAN, S. et al. A Study on Polymorphing Superscalar Processor Dynamically to Improve Power Efficiency. IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, Natal, 2013. **Proceedings...** New York: IEEE, 2013 p. 46–51,

Synopsys Design Compiler. Disponível em: < http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx>. Acesso em: 4 jan. 2016.

TEODORESCU, R.; TORRELLAS, J. Variation-aware application scheduling and power management for chip multiprocessors. INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, Beijing, 2008. **Proceedings…** New York: IEEE, 2008, p. 363-374

WATKINS, M. A; ALBONESI, D. H. Enabling Parallelization via a Reconfigurable Chip Multiprocessor. **Pespma 2010-Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture**, v. 2, p. 59–68, 2010.

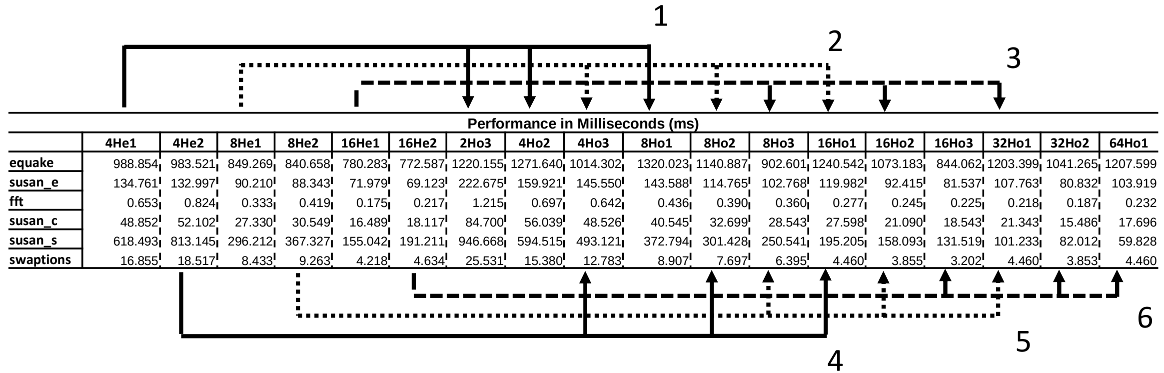
WILTON, S. J. E.; JOUPPI, N. P. CACTI: an enhanced cache access and cycle time model. **IEEE Journal of Solid-State Circuits**, v. 31, n. 5, p. 677–688, maio 1996.

WOLF, W.; JERRAYA, A. A.; MARTIN, G. Multiprocessor system-on-chip (MPSoC) technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 27, n. 10, p. 1701–1713, 2008.

WOO, S. C. et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, Santa Margherita Ligure, 1995. **Proceedings…** New York: IEEE, 1995, p. 24–36

APPENDIX A

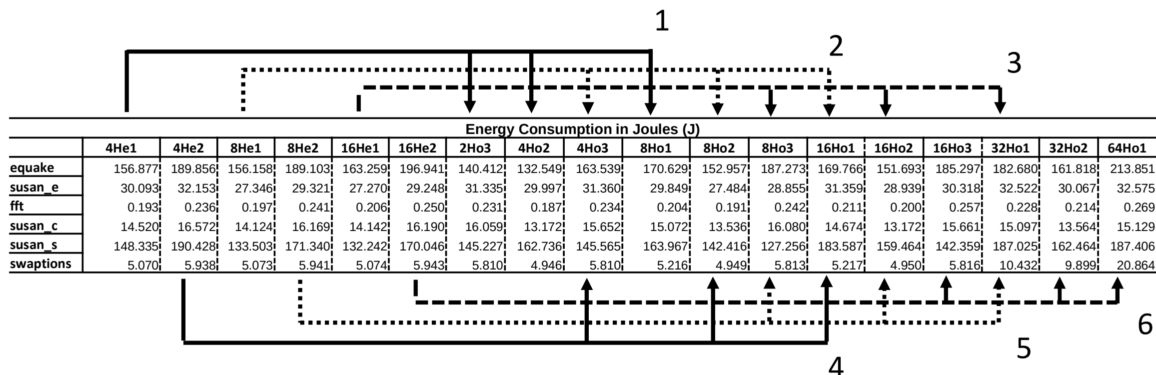
Table V: Execution time (in milliseconds) of each application in each configuration. The arrows indicate the area parities between configurations. For instance, arrow 1 shows that 4He1 (a 4-core He1 configuration) has approximately the same area as 2Ho3, 4Ho2 and 8Ho1.



Performance in Milliseconds (ms)																		
	4He1	4He2	8He1	8He2	16He1	16He2	2Ho3	4Ho2	4Ho3	8Ho1	8Ho2	8Ho3	16Ho1	16Ho2	16Ho3	32Ho1	32Ho2	64Ho1
equake	988.854	983.521	849.269	840.658	780.283	772.587	1220.155	1271.640	1014.302	1320.023	1140.887	902.601	1240.542	1073.183	844.062	1203.399	1041.265	1207.599
susan_e	134.761	132.997	90.210	88.343	71.979	69.123	222.675	159.921	145.550	143.588	114.765	102.768	119.982	92.415	81.537	107.763	80.832	103.919
fft	0.653	0.824	0.333	0.419	0.175	0.217	1.215	0.697	0.642	0.436	0.390	0.360	0.277	0.245	0.225	0.218	0.187	0.232
susan_c	48.852	52.102	27.330	30.549	16.489	18.117	84.700	56.039	48.526	40.545	32.699	28.543	27.598	21.090	18.543	21.343	15.486	17.696
susan_s	618.493	813.145	296.212	367.327	155.042	191.211	946.668	594.515	493.121	372.794	301.428	250.541	195.205	158.093	131.519	101.233	82.012	59.828
swaptions	16.855	18.517	8.433	9.263	4.218	4.634	25.531	15.380	12.783	8.907	7.697	6.395	4.460	3.855	3.202	4.460	3.853	4.460

Source: the author

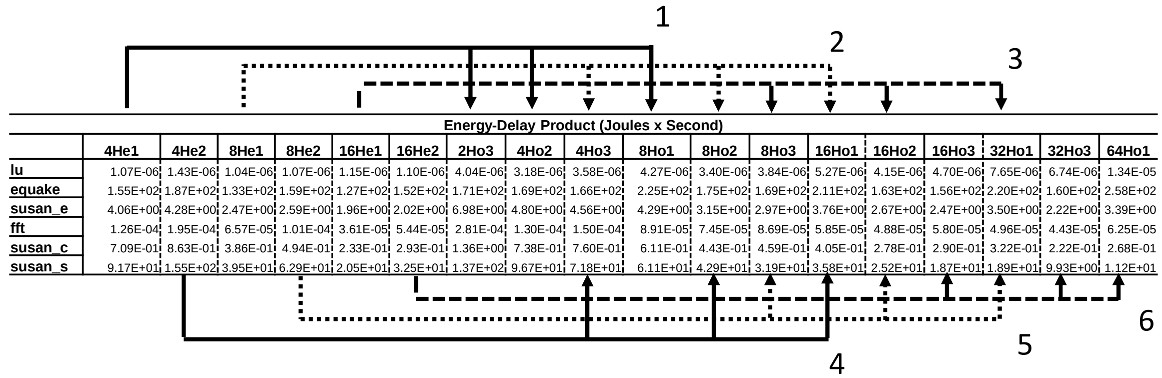
Table VI: Energy consumption (in Joules) of each application in each configuration. The arrows indicate the area parities between configurations. For instance, arrow 1 shows that 4He1 (a 4-core He1 configuration) has approximately the same area as 2Ho3, 4Ho2 and 8Ho1.



Energy Consumption in Joules (J)																		
	4He1	4He2	8He1	8He2	16He1	16He2	2Ho3	4Ho2	4Ho3	8Ho1	8Ho2	8Ho3	16Ho1	16Ho2	16Ho3	32Ho1	32Ho2	64Ho1
equake	156.877	189.856	156.158	189.103	163.259	196.941	140.412	132.549	163.539	170.629	152.957	187.273	169.766	151.693	185.297	182.680	161.818	213.851
susan_e	30.093	32.153	27.346	29.321	27.270	29.248	31.335	29.997	31.360	29.849	27.484	28.855	31.359	28.939	30.318	32.522	30.067	32.575
fft	0.193	0.236	0.197	0.241	0.206	0.250	0.231	0.187	0.234	0.204	0.191	0.242	0.211	0.200	0.257	0.228	0.214	0.269
susan_c	14.520	16.572	14.124	16.169	14.142	16.190	16.059	13.172	15.652	15.072	13.536	16.080	14.674	13.172	15.661	15.097	13.564	15.129
susan_s	148.335	190.428	133.503	171.340	132.242	170.046	145.227	162.736	145.565	163.967	142.416	127.256	183.587	159.464	142.359	187.025	162.464	187.406
swaptions	5.070	5.938	5.073	5.941	5.074	5.943	5.810	4.946	5.810	5.216	4.949	5.813	5.217	4.950	5.816	10.432	9.899	20.864

Source: the author

Table VII: Energy-Delay Product (Joules x Seconds) of each application in each configuration. The arrows indicate the area parities between configurations. For instance, arrow 1 shows that 4He1 (a 4-core He1 configuration) has approximately the same area as 2Ho3, 4Ho2 and 8Ho1.



Energy-Delay Product (Joules x Second)																		
	4He1	4He2	8He1	8He2	16He1	16He2	2Ho3	4Ho2	4Ho3	8Ho1	8Ho2	8Ho3	16Ho1	16Ho2	16Ho3	32Ho1	32Ho3	64Ho1
lu	1.07E-06	1.43E-06	1.04E-06	1.07E-06	1.15E-06	1.10E-06	4.04E-06	3.18E-06	3.58E-06	4.27E-06	3.40E-06	3.84E-06	5.27E-06	4.15E-06	4.70E-06	7.65E-06	6.74E-06	1.34E-05
equake	1.55E+02	1.87E+02	1.33E+02	1.59E+02	1.27E+02	1.52E+02	1.71E+02	1.69E+02	1.66E+02	2.25E+02	1.75E+02	1.69E+02	2.11E+02	1.63E+02	1.56E+02	2.20E+02	1.60E+02	2.58E+02
susan_e	4.06E+00	4.28E+00	2.47E+00	2.59E+00	1.96E+00	2.02E+00	6.98E+00	4.80E+00	4.56E+00	4.29E+00	3.15E+00	2.97E+00	3.76E+00	2.67E+00	2.47E+00	3.50E+00	2.22E+00	3.39E+00
fft	1.26E-04	1.95E-04	6.57E-05	1.01E-04	3.61E-05	5.44E-05	2.81E-04	1.30E-04	1.50E-04	8.91E-05	7.45E-05	8.69E-05	5.85E-05	4.88E-05	5.80E-05	4.96E-05	4.43E-05	6.25E-05
susan_c	7.09E-01	8.63E-01	3.86E-01	4.94E-01	2.33E-01	2.93E-01	1.36E+00	7.38E-01	7.60E-01	6.11E-01	4.43E-01	4.59E-01	4.05E-01	2.78E-01	2.90E-01	3.22E-01	2.22E-01	2.68E-01
susan_s	9.17E+01	1.55E+02	3.95E+01	6.29E+01	2.05E+01	3.25E+01	1.37E+02	9.67E+01	7.18E+01	6.11E+01	4.29E+01	3.19E+01	3.58E+01	2.52E+01	1.87E+01	1.89E+01	9.93E+00	1.12E+01

Source: the author

APPENDIX B

This appendix contains details about the power consumption of the components of the reconfigurable unit used in this work. The power for the components of the reconfigurable array, shown in Table VIII, were extracted using the Synopsys Design Compiler ("Synopsys Design Compiler", 2010) by Rutzig in (RUTZIG, 2012). The power for the reconfiguration memory were extracted using CACTI-P (LI et al., 2011). Table X brings the data for power consumption for each access in the reconfiguration memory of each of the types of DAP (large, medium and small) present in the heterogeneous He1 configuration, while Table XI shows these values for the He2 configuration.

Table VIII: Power consumption of each of the components of the reconfigurable array

Array Power Consumption	
Unit	Power (mW)
SparcV8	44.5
ALU	8.5
Multiplier	13.6
Load/Store	0.1
Float Point	25.5

Source: the author

Table IX: Power consumption for each access in the reconfiguration memory of the homogeneous configurations.

Homogeneous Reconfiguration Memory	
DAP	Power (mW)
Ho1	71.08
Ho2	84.74
Ho3	244.08

Source: the author

Table X: Power consumption for each access in the reconfiguration memory of the different DAPs in the heterogeneous He1 configurations.

He1 Reconfiguration Memory	
DAP	Power (mW)
Large	139.59
Medium	84.75
Small	71.08
Average	108.7525

Source: the author

Table XI: Power consumption for each access in the reconfiguration memory of the different DAPs in the heterogeneous He2 configurations.

He2 Reconfiguration Memory	
DAP	Power (mW)
Large	265.93
Medium	139.59
Small	71.33
Average	185.695

Source: the author

APPENDIX C

This appendix brings the ratio of area for each component on the DAPs for all the configurations studied (both homogeneous and heterogeneous).

Table XII: Area ratio for components inside the He1 DAPs.

He1 Area Ratio					
Small DAP		Medium DAP		Large DAP	
Reconfigurable Array	14.64%	Reconfigurable Array	14.35%	Reconfigurable Array	12.52%
Context memory	60.69%	Context memory	73.75%	Context memory	82.85%
I/D Caches	20.80%	I/D Caches	10.03%	I/D Caches	3.91%
Sparc	3.86%	Sparc	1.86%	Sparc	0.73%

Table XIII: Area ratio for components inside the He2 DAPs.

He2 Area Ratio					
Small DAP		Medium DAP		Large DAP	
Reconfigurable Array	29.59%	Reconfigurable Array	32.60%	Reconfigurable Array	22.02%
Context memory	51.38%	Context memory	59.88%	Context memory	75.19%
I/D Caches	16.05%	I/D Caches	6.34%	I/D Caches	2.36%
Sparc	2.98%	Sparc	1.18%	Sparc	0.44%

Table XIV: Area ratio for components inside the Ho DAPs.

Homogeneous Area Ratio					
Ho1		Ho2		Ho3	
Reconfigurable Array	14.64%	Reconfigurable Array	19.03%	Reconfigurable Array	17.97%
Context memory	60.69%	Context memory	70.63%	Context memory	78.88%
I/D Caches	20.80%	I/D Caches	8.72%	I/D Caches	2.66%
Sparc	3.86%	Sparc	1.62%	Sparc	0.49%

APPENDIX D

Introdução

Os sistemas embarcados tornaram-se populares graças a grande disseminação do uso de smartphones e outros *gadgets* em nossa sociedade. Tais sistemas são capazes de executar diversos tipos de aplicações em uma grande variedade de funcionalidades, abrangendo um ambiente extremamente heterogêneo de operação. Para atingir altos níveis de desempenho, sistemas multinúcleos foram propostos para o ambiente de embarcados. Ter diversos núcleos permite que o processador explore o potencial de paralelismo entre as *threads* de uma aplicação. Além de multinúcleos, os sistemas propostos são homogêneos em arquitetura (possuem o mesmo conjunto de instruções entre os núcleos) e em organização (são fisicamente iguais) e são normalmente compostos de núcleos superescalares, os quais são capazes de, individualmente, explorar o paralelismo entre as instruções da aplicação.

Contudo, sistemas multinúcleos homogêneos não são ineficientes quando são usados em ambientes de execução heterogênea. Como todos os núcleos tem o mesmo poder de processamento, alguns recursos do processador podem ser desperdiçados ao executar aplicações de diferentes cargas de trabalho. Para adaptar a carga requisitada pelas aplicações ao processador, sistemas multinúcleo heterogêneos foram propostos. Dentre esses, os sistemas mais comuns são os SoCs (*System-on-Chip* – Sistema em um Chip), que são chips compostos de núcleos com circuitos específicos para executar determinada aplicação (decodificação de áudio, processamento gráfico, filtro de sinais, etc). A grande desvantagem dos SoCs é que eles são heterogêneos não somente em organização, mas também em arquitetura: cada um dos circuitos de aplicação específica trabalha com um conjunto de instruções diferente, dificultando o processo de desenvolvimento de software para o sistema. Outro exemplo de multinúcleo heterogêneo são os processadores big.LITTLE da ARM. Estes processadores são compostos de núcleos superescalares de mesma arquitetura, mas com diferentes organizações (diferentes capacidades de processamento), de forma a criar um ambiente mais eficiente. Contudo, processadores superescalares de processamento geral não são tão energeticamente eficientes como os circuitos de aplicação específica dos SoC, além de que criar diferentes organizações para estes sistemas exigem mudanças de alta complexidade no projeto do circuito.

Este trabalho propõe um sistema heterogêneo reconfigurável, dinâmico e transparente, o qual é completamente diferente dos sistemas heterogêneos citados e atualmente utilizados na

indústria: o HARTMP (*Heterogeneous Arrays for Reconfigurable and Transparent Multicore Processing* – Arranjos Heterogêneos para Processamento Multinúcleo Reconfigurável e Transparente). Sistemas reconfiguráveis são capazes de se adaptar de acordo com a aplicação que está sendo executada, simulando o comportamento de um circuito de aplicação específica. Além disso, o circuito reconfigurável de tais sistemas é composto de células extremamente regulares, de forma que mudar a capacidade de processamento de cada núcleo não exige mudanças complexas no projeto do sistema. O HARTMP é dinâmico, pois é capaz de interpretar, durante a execução, o comportamento da aplicação e reconhecer os pontos de código passíveis de aceleração. Além disso, é transparente, pois todo o processo de reconfiguração é transparente ao programador, não sendo necessárias modificações nos códigos fonte originais. Tais características são alcançadas graças a um sistema de tradução binária, capaz de manter a arquitetura do sistema homogênea. Por outro lado, a eficiência dos sistemas heterogêneos é altamente dependente da capacidade de alocar corretamente as tarefas das aplicações nos diferentes núcleos do processador.

Este trabalho avalia os ganhos de desempenho e de energia dados pela heterogeneidade do HARTMP. Para isso, este sistema heterogêneo é comparado com um sistema reconfigurável idêntico ao HARTMP, contudo de núcleos homogêneos. Também é avaliado o comportamento de diversos escalonadores executando no HARTMP, de forma a identificar algumas das principais características que definem uma alocação eficiente neste sistema.

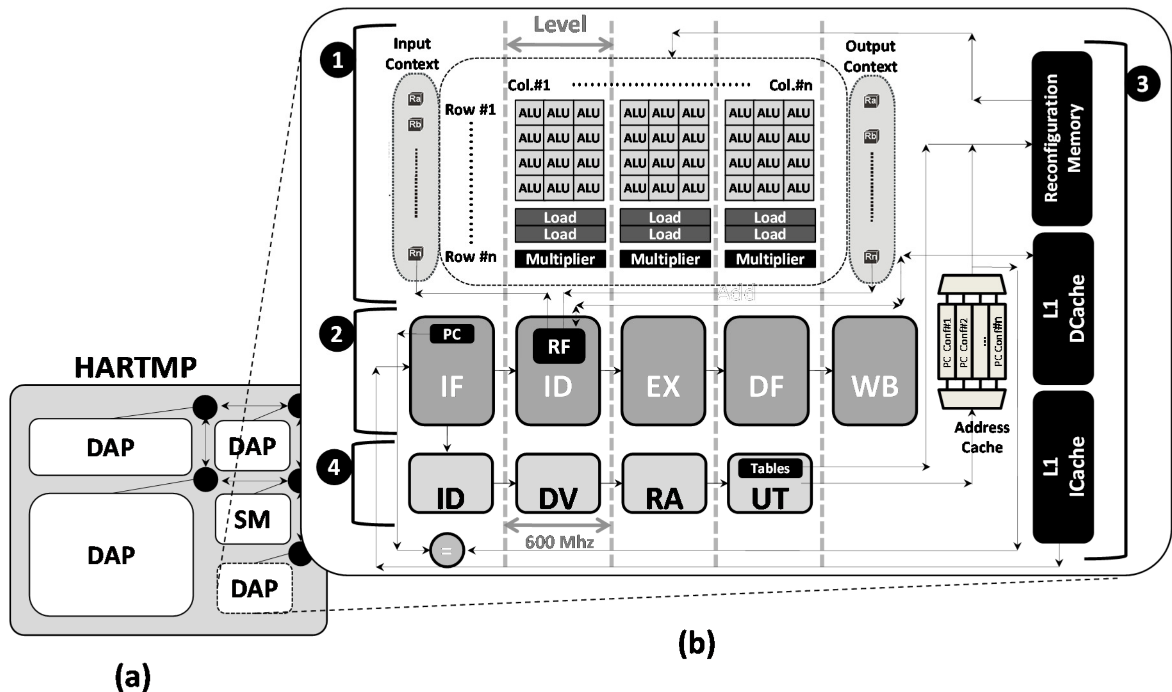
HARTMP

Uma visão geral do HARTMP é apresentada na Figura 1. O paralelismo em nível de thread é explorado pela replicação de DAPs (*Dynamic Adaptive Processors*) e cada DAP possui um tamanho diferente, sendo capaz de explorar diferentes níveis de paralelismo entre as instruções. A comunicação entre os DAPs é feita através de uma NoC (*Network on Chip*) de topologia xy com uma memória cache L2 compartilhada.

DAP

Um Dynamic Adaptive Processor é dividido em quatro blocos, ilustrados na Figura 1. Estes blocos são discutidos a seguir:

Figura 1. (a) Organização do HARTMP (b) DAP



Fonte: o autor

Bloco 1 – Unidade Funcional Reconfigurável

Um arranjo de unidades funcionais ligadas entre si através de multiplexadores. Tais multiplexadores podem ser reconfigurados, de forma que o caminho percorrido pelas entradas e saídas de cada unidade funcional representem a execução combinacional de um bloco de instruções de uma aplicação.

Bloco 2 - Processador Pipeline

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

Bloco 3 – Memória de reconfiguração

O HARTMP explora três técnicas amplamente difundidas no meio científico: a reconfigurabilidade, a tradução binária e o reuso. O reuso é uma técnica que explora a natureza de execução do modelo Von-Neumann, cujo o contador de programa determina o fluxo de execução de uma aplicação. Desta forma, laços ou saltos que resultam em repetições no código da aplicação que é executado.

O HARTMP se beneficia desta característica de reuso das aplicações realizando a tradução binária destes trechos de código, analisando as dependências entre as instruções destes blocos e armazenando as configurações formadas em uma cache especial, chamada de memória de reconfiguração. Esta abordagem evita a necessidade de repetir a análise das dependências cada vez que um trecho de código é reexecutado, uma das principais desvantagens de processadores superescalares.

A memória de reconfiguração armazena, em cada posição, os dados necessários para uma configuração na UFR, incluindo os bits para configurar os multiplexadores, as unidades funcionais e os dados de operadores imediatos das instruções.

Bloco 4 – Detector Dinâmico em Hardware

Como discutido anteriormente, uma das grandes imposições dos SoCs é a incompatibilidade binária entre os diferentes aceleradores presentes no chip. A tradução binária é apresentada como uma forma de manter compatibilidade de software entre sistemas com diferentes conjuntos de instruções. Em (BECK, RUTZIG, *et al.*, 2008), os autores mostram como a tradução binária também pode ser utilizada com ótimos resultados, ambos para desempenho e consumo energético, em sistemas reconfiguráveis.

O Detector Dinâmico em Hardware (DDH) é capaz de prover compatibilidade de software traduzindo sequências de instruções, durante a execução da aplicação, para que estas sejam executadas futuramente em um acelerador, neste caso, na lógica reconfigurável do DAP.

O DDH avalia as instruções executadas no processador de *pipeline* e as agrupa em blocos, chamados de configurações da unidade funcional reconfigurável (UFR). Cada instrução executada pelo processador base é analisada pelo DDH de forma a poder executá-la no bloco reconfigurável. Se for possível, a instrução é alocada na configuração sendo montada na UFR. Quando uma configuração é concluída, ela é armazenada em uma cache de reconfigurações e indexada pelo valor do contador de programa da primeira instrução desta configuração. Quando o contador alcançar novamente este valor (ou seja, quando o bloco for executado novamente), o aplicação não é executada no processador base, mas sim no mecanismo reconfigurável, seguindo os seguintes passos:

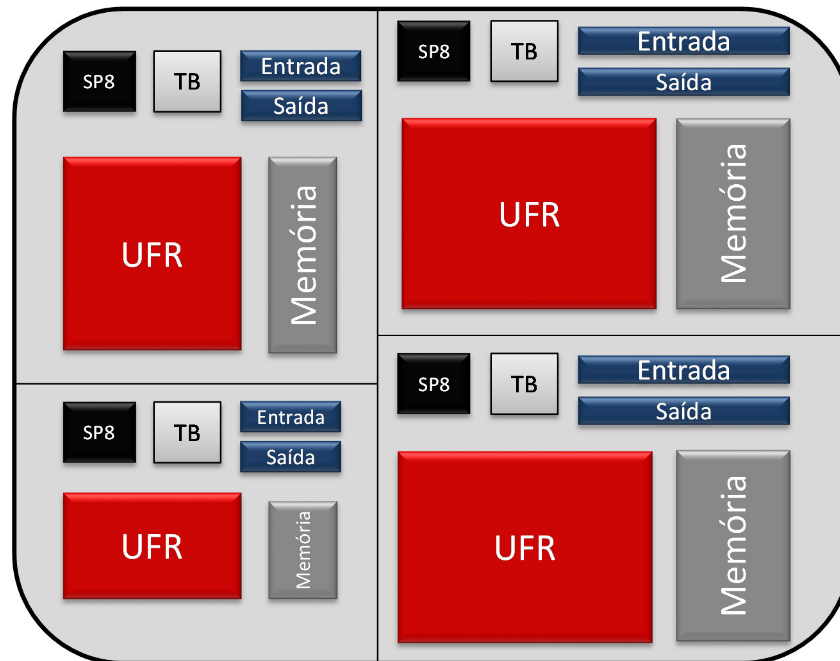
- A configuração é buscada na cache de reconfigurações;
- Reconfigura-se a UFR com os bits buscados na cache;

- Os valores dos registradores utilizados pela configuração são carregados do banco de registradores do processador para o contexto de entrada da UFR;
- A execução do bloco é feita na UFR;

Ao término da execução, o valor do contador de programa é atualizado para apontar para o fim do bloco e os valores do banco de registradores do processador são atualizados com o contexto de saída da UFR. Diferente dos processadores superescalares, a abordagem feita pelo DDH não exige que as dependências de dados sejam reavaliadas a cada execução de uma determinada sequência de instruções. Esta checagem é feita apenas uma vez, durante a criação da configuração, a qual é armazenada na cache de reconfiguração para uso futuro.

Em um sistema HARTMP, cada um dos DAPs (núcleos do processador) possui componentes de tamanho fixo e outros de tamanho variável. Como já explicado anteriormente, aumentar a capacidade de explorar paralelismo dentro da unidade reconfigurável é mais simples do que aumentar a complexidade do processador de *pipeline.* Desta forma, o processador é mantido inalterado para todos os núcleos. Como o algoritmo de tradução binária é o mesmo para todos os tamanhos de UFR, este componente também é mantido fixo. Os demais componentes que compõem a lógica reconfigurável (os registradores de contexto, a memória de reconfiguração e a UFR) são variados, para que cada núcleo tenha diferentes capacidades de processamento. A vantagem de se alterar esses três componentes é que todos eles possuem um circuito altamente regular, ou seja, de fácil replicação, o que facilita o trabalho de criar configurações de diversos tamanhos.

Figura 2: Exemplo de sistema HARTMP. O processador de *pipeline* (SparcV8) e o tradutor binário tem tamanhos fixos para todos núcleos. Os contextos de entrada e saída, a memória de reconfiguração e a unidade funcional reconfigurável possuem tamanhos variáveis para cada núcleo.



Fonte: o autor

Escalonadores

Um dos pontos críticos dos sistemas heterogêneos é escalonar, nos recursos disponíveis, de forma correta e eficiente, o algoritmo em execução. Neste trabalho, seis escalonadores foram implementados para analisar o impacto da alocação no HARTMP. Um destes algoritmos é um escalonador preditivo, capaz de sempre encontrar a melhor solução de alocação para obter o melhor desempenho do sistema. Utilizamos este algoritmo para comparar com os demais e obter o a relação de performance entre o melhor cenário e os demais.

Oráculo

O escalonador Oráculo é um algoritmo preditivo capaz de determinar a melhor alocação para o melhor desempenho. Neste escalonador, todas as *threads* da aplicação são executadas em todos os núcleos do sistema de forma que no fim da execução, o resultado de cada etapa da aplicação é obtido. Após, uma combinação de todas as possíveis alocações das threads é feita e o melhor resultado possível é extraído. O Oráculo não é um algoritmo implementável em um sistema real, ele é usado apenas para medir o potencial do sistema heterogêneo e também para determinar o quão perto do Oráculo os demais algoritmo de alocação podem chegar.

Estático

O escalonador estático aloca as *threads* criadas no primeiro núcleo livre do sistema e as mantém neste núcleo até o fim da execução. Em outras palavras, o algoritmo nunca muda a alocação da aplicação, sendo assim uma solução sem complexidade alguma e de baixo custo. Outra vantagem do algoritmo estático é que ele sempre aloca a *master thread* (a primeira *thread* criada e que usualmente contém a maior carga de trabalho de uma aplicação) em um núcleo grande, ou seja, a *máster thread* sempre será alocada no melhor núcleo possível.

Estático Inverso

Para obter o impacto de alocação da *master thread*, o algoritmo estático inverso foi criado. Este alocador trabalha da mesma forma que o algoritmo estático, contudo ao invés de alocar a *master thread* em um núcleo grande, ele a aloca em um núcleo pequeno, de forma que esta *thread* será alocada no pior núcleo possível.

IC-Driven

O escalonador IC-Driven (do inglês *Instruction Counter Driven:* dirigido ao contador de instruções) é um algoritmo simples que adiciona um contador de instruções para cada *thread* da aplicação. Quando as *threads* atingem um ponto de sincronização no código (um ponto onde todas as *threads* devem se encontrar para garantir a validade dos dados), este algoritmo é ativado. As *threads* que tiverem a maior contagem de instruções executadas serão realocadas para núcleos maiores, enquanto que aquelas que tiverem o menor número serão realocadas para núcleos menores.

Apesar de simples, este algoritmo apresenta duas desvantagens. A primeira é que ele se baseia em informações do passado recente para decidir a alocação presente. Quando um ponto de sincronização é atingido, este algoritmo utiliza contagens do ponto anterior para decidir a alocação do ponto corrente, contudo nada garante que o ponto corrente terá o mesmo comportamento do ponto anterior. A segunda desvantagem é devia as características de exploração de paralelismo das instruções no sistema HARTMP. Neste sistema, apenas as instruções que são executadas pela lógica reconfigurável é que são aceleradas, aquelas que são executadas no *pipeline* do processador não exploram qualquer tipo de paralelismo. Além disso, toda a heterogeneidade de um sistema HARTMP está na lógica reconfigurável, por tanto, utilizar todas as instruções executadas não é a melhor métrica a ser escolhida.

Feedback

O escalonador de *Feedback* (em português: retorno) é um algoritmo muito similar ao IC-Driven, contudo ele utiliza dados retornados por contadores de hardware presentes em cada DAP como métrica de escalonamento. Os contadores de hardware retornam ao escalonador quantas instruções foram executadas dentro da UFR apenas, de forma que o algoritmo agora utiliza apenas informações dos trechos de programa que são de fato acelerados, ignorando aqueles que são executados no processador de *pipeline*, que não tem exploração de paralelismo em instruções.

Assim como o alocador *IC-Driven*, este algoritmo também se baseia em dados do ponto de sincronização anterior para escalonar as *threads* do ponto corrente. Como discutido anteriormente, não há garantia – contudo – de que o ponto corrente tenha o mesmo comportamento do ponto anterior.

PC-Feedback

No escalonador *PC-Feedback* (do inglês *Program Counter Feedback:* retorno de contador de programa), assim como no de *Feedback*, a métrica para alocação de *threads* também é o número de instruções executadas na UFR. Contudo, este algoritmo utiliza a técnica de reuso para tentar corrigir a desvantagem de utilizar o comportamento do ponto de sincronização anterior para determinar a alocação do ponto corrente.

Neste escalonador, cada vez que um ponto de sincronização é executado, o algoritmo analisa o número de instruções de cada *thread* e armazena este valor associado ao contador de programa daquele ponto. Na próxima vez que aquele ponto de sincronização for executado, o alocador irá ter informações prévias de comportamento daquele ponto e saberá com mais precisão como fazer a alocação. Se um ponto de sincronização que ainda não foi analisado for executado, o algoritmo não faz nenhuma mudança na alocação.

Metodologia

Diversos *benchmarks* de diferentes características e níveis de exploração de paralelismo em instruções e em *threads* foram selecionados para cobrir um amplo espaço de aplicação. Todos esses *benchmarks* foram paralelizados usando as interfaces de programação OpenMP e Pthreads, ambas amplamente usadas por desenvolvedores de software. Destas aplicações, foram extraídos resultados de desempenho e energia para os sistemas heterogêneo (HARTMP) e sua versão homogênea.

Para comparar as diferentes versões, foi usado um sistema de paridade de área, de forma que apenas sistemas que ocupem o mesmo espaço em chip sejam comparados. Ou seja, se um sistema homogêneo composto apenas de núcleos pequenos for comparado com um heterogêneo com núcleos pequenos, médios e grandes, esse sistema homogêneo pode ter mais núcleos do que o heterogêneo para manter a paridade de área.

Um ambiente de simulação é usado para extrair os resultados das aplicações. A plataforma de simulação Simics é usada para gerar o *trace* de execução das aplicações, enquanto um script é responsável por ler esse *trace*, classificar as instruções de cada *thread* e escalonar essas instruções para simuladores individuais de DAP (simuladores de precisão de ciclo que emulam o comportamento do DAP).

Para extração de dados de potência, área e caminho crítico, todos os componentes do DAP foram descritos em VHDL e modelados usando a ferramenta Synopsys Design Compiler. O consumo de potência e a área dos componentes de memória foram extraídos usado a ferramenta CACTI-P.

Resultados

Dois conjuntos de análises são feitos neste trabalho. No primeiro, são comparados dois sistemas heterogêneos (He1 e He2) com outros três sistemas homogêneos (Ho1, Ho2 e Ho3). Cada um destes sistemas tem tamanhos de núcleos diferentes, de forma que a paridade de área entre estes sistemas é diferente. Na Tabela 1 a paridade de cada um dos sistemas é mostrada. Por exemplo, a paridade entre He1 e Ho1 é de dois núcleos de Ho1 para cada núcleo médio de He1, ou em outras palavras, para que o sistema homogêneo Ho1 tenha o mesmo tamanho do sistema heterogêneo He1 ele deve ter duas vezes mais núcleos. Todos os sistemas heterogêneos são simulados usando o algoritmo de alocação Oráculo, ou seja, os resultados mostrados são os melhores possíveis em relação ao desempenho.

No segundo conjunto de resultados, cada um dos escalonadores propostos são avaliados. O objetivo é verificar em quais aplicações cada algoritmo possui vantagens e desvantagens e qual dos escalonadores é capaz de atingir o desempenho mais próximo ao Oráculo.

Tabela 1: Relação de número de DAP entre configurações de HARTMP (heterogêneo) e CReAMS (homogêneo)

	Ho1	**Ho2**	**Ho3**
He1	2 Ho1 : 1 He1	1 Ho2 : 1 He1	1 Ho3 : 2 He1
He2	4 Ho1 : 1 He2	2 Ho2 : 1 He2	1 Ho3 : 1 He2

Fonte: o autor

Sistema Heterogêneo vs Homogêneo

Nesta análise, as duas versões de HARTMP são comparadas com as três versões de CReAMS. O que podemos observar é que um sistema HARTMP é, geralmente, mais rápido do que o homogêneo quando as aplicações executadas possuem *threads* altamente desbalanceadas (com pouco paralelismo entre si). Neste caso, o escalonador é capas de alocar eficientemente as *threads* com mais processamento sequencial nos núcleos maiores, enquanto que as *threads* mais paralelas são alocadas nos núcleos menores. Podemos ver uma melhora de até 59% em desempenho quando usando um sistema HARTMP em aplicações desbalanceadas.

Entretanto, quando as aplicações são altamente balanceadas, o sistema heterogêneo acaba perdendo em performance. Isso acontece pois durante a execução das aplicações todas as *threads* devem se encontrar em um determinado ponto e as *threads* que executam nos núcleos maiores sempre terão que aguardar por aquelas que executam no núcleos menores (pois todas as *threads* tem a mesma carga de trabalho). No caso do sistema homogêneo ser composto apenas por núcleos pequenos (Ho1), novamente ele tem vantagens sobre o heterogêneo, pois esta versão de CReAMS possui mais núcleos do que a versão de HARTMP, sendo capaz de explorar mais o paralelismo entre *threads*.

Quando a eficiência energética é considerada, o grande gargalo de energia encontrado foi a memória de reconfiguração. Quanto maior a UFR, maior deve ser esta memória (para armazenar configurações maiores), e o custo energético para acessar esta também aumenta. Neste caso, o sistema heterogêneo é mais eficiente quando o sistema homogêneo em questão tem núcleos grandes. Ganhos energéticos médios de 20% podem ser observados nos resultados.

Outro resultado avaliado foi o EDP (do inglês *Energy-Delay Product*: Produto Energia-Atraso), que é a multiplicação entre o tempo de execução de uma aplicação e a energia gasta neste tempo. O EDP é capaz de mostrar quando uma variação negativa em um dos parâmetros resulta em uma variação positiva de maior valor no outro. Por exemplo, se o tempo de execução de uma aplicação pode aumentar em um fator, mas se o consumo energético diminui por um fator maior, então o EDP irá mostrar que a diminuição do consumo compensa no aumento do tempo de execução. Os resultados mostram que, em todos os casos, há aplicações em que o HARTMP tem melhor EDP do que a versão homogênea, especialmente na versão He1.

Escalonadores

O escalonador Oráculo foi criar para mostrar o potencial do sistema (como discutido anteriormente) e também para servir de base de comparação com os outros algoritmos reais. Quando avaliamos o algoritmo estático, vemos que este é capaz de atingir 99% do desempenho do Oráculo em aplicações altamente desbalanceadas e nas altamente balanceadas também. No caso de aplicações desbalanceadas, o ganho se deve graças a alocação da *master thread* sempre ser feita em um núcleo grande (como discutido anteriormente), sendo que a alocação das demais threads pouco influencia nos resultados. No caso das aplicações balanceadas, a alocação não afeta o desempenho das aplicações, pois todas as *threads* desempenham o mesmo trabalho, de forma que o núcleo pequeno da configuração torna-se um gargalo no sistema. Contudo, para aplicações que não são altamente balanceadas ou desbalanceadas, o algoritmo estático não atinge o desempenho próximo do Oráculo desejado. Para medir o real impacto de alocar a *master thread* em um núcleo grande, criamos o algoritmo estático inverso, que aloca a *thread* principal em um núcleo pequeno. Este algoritmo atinge apenas 70% do desempenho do Oráculo, onde o estático comum atingia 99%, mostrando que o impacto é, de fato, alto.

O escalonador de IC-Driven foi proposto como uma solução inicial para incluir métricas de escalonamento. Nos resultados, este algoritmo se mostrou ineficiente, fazendo trocas desnecessárias e diminuindo o desempenho das aplicações. Em geral, este algoritmo é pior do que o estático. Contudo, ele serviu para mostrar os pontos fracos onde o escalonador deveria ser melhorado.

O escalonador de Feedback foi proposto com uma ideia bastante similar ao IC-Driven, contudo usando uma métrica de melhor escolha para o sistema HARTMP, buscando explorar os pontos onde esse sistema de fato se aproveita da heterogeneidade. Este escalonador mostrou ter o mesmo desempenho próximo ao Oráculo do algoritmo estático (em aplicações altamente balanceadas/desbalanceadas) e melhor desempenho nas aplicações de comportamento variado, sendo, assim, um algoritmo de propósito mais geral do que o estático. No caso do escalonador PC-Feedback, melhoras foram observadas em algumas aplicações, contudo se comparado ao escalonador Feedback, estas melhoras são muito marginais.

Conclusão

Este trabalho apresenta um sistema reconfigurável e transparente, capaz de prover heterogeneidade sem acrescentar qualquer complexidade tanto no projeto do sistema como na produção de software: o HARTMP. O sistema atinge a transparência usando um algoritmo de tradução binária entre o processador de *pipeline* e a UFR. A heterogeneidade é facilmente atingida replicando os núcleos do sistema e usando, em cada núcleo, UFRs de tamanhos diferentes. A UFR é composta de elementos projeto altamente regular, ou seja, de fácil replicação em hardware. O sistema provou ser mais eficiente tanto em desempenho e como em energia do que a versão homogênea em diversas aplicações.

Além disso, foram avaliados diversos escalonadores para o sistema heterogêneo. O escalonador estático atingiu desempenho próximo do perfeito em aplicações com alto balanceamento/desbalanceamento de carga entre as *threads*, mostrando-se como uma opção simples para alguns tipos de aplicação. Contudo, o algoritmo de Feedback se mostrou tão bom quando o estático e melhor nas demais aplicações, sendo uma escolha de melhor desempenho geral do que o estático.