Towards a Dynamic and Reconfigurable Multicore Heterogeneous System

Jeckson Dellagostin Souza, Luigi Carro, Antonio Carlos Schneider Beck
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Porto Alegre, Brasil
{jeckson.souza, carro, caco }@inf.ufrgs.br

Mateus Beck Rutzig
Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Santa Maria, Brasil
mateus@inf.ufsm.br

Abstract— Reconfigurable architectures are an alternative for classic superscalar organizations to the exploration of instruction level parallelism (ILP), while the multicore organizations are the most commonly used strategy to exploit thread level parallelism (TLP). This work extends a dynamic and transparent homogeneous multicore reconfigurable system (CReAMS) that explores both TLP and ILP, by making it heterogeneous, featuring cores with distinct resources. We will show that, for applications with low TLP, a heterogeneous configuration of CReAMS outperforms its homogeneous counterpart with the same chip area. The performed simulations prove the potential of using heterogeneous reconfigurable systems, showing speedups of up to 90%.

Keywords—reconfigurable system, multiprocessor, embedded systems, heterogeneous systems

I. INTRODUCTION

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

There are many ways which can be used to improve a processor's performance. Parallel execution is a common strategy used to accelerate a program and to efficiently use the resources of a processor. If a sequence of independent instructions (that do not operate over the same data set) is dispatched, the processor can allocate them to different functional units and process them concurrently. This provides performance gains due to the exploitation of the instruction level parallelism (ILP).

The superscalar approach is widely used to exploit ILP in both general purpose – as the Intel x86 architectures – and embedded processors – like the ARM architecture. However, it is expensive in terms of power consumption, since for each incoming instruction, the superscalar processor has to repeatedly evaluate data dependencies to exploit the instruction level parallelism. Another possible strategy to exploit ILP is dynamic reconfigurable architectures. They are projected to adapt themselves according to the application at hand, reconfiguring their datapath so that the stream of data through the functional units is optimized. The great advantage of these architectures is that they can also optimize data dependent instructions, besides executing the non-dependent ones concurrently.

On the other hand, the employment of multicore organizations, supported by the scheduler of operating systems, enables the simultaneous execution of application threads through many processor cores, thus thread level parallelism (TLP) is explored. Therefore, it is also possible to increase performance on reconfigurable organizations through the exploitation of TLP, by the replication of cores that have their own reconfigurable datapath each. CReAMS (Custom Reconfigurable Arrays for Multiprocessor Systems) is an example of such a system.

However, CReAMS is homogeneous: all cores are the same, which means that the reconfigurable array is also identical on every core. That leads to inefficiency problems when running applications with varied workload. When the cores are not under full usage, many of the functional units on the reconfigurable array are not used, wasting resources and power. With that being said, it is possible to build a heterogeneous CReAMS configuration – whose cores are different – which is the proposal of this work. On heterogeneous systems, threads that highly explore the ILP can be allocated to cores with bigger reconfigurable datapaths (that have more functional units), while threads with lower ILP can be dispatched to smaller ones.

In this work we will simulate a set of heterogeneous CReAMS configurations and compare them with other homogeneous configurations of same area. For this, we will use five different benchmarks – selected to cover a wide range of applications – to measure CReAMS performance on the configurations built. We will show that, depending on the application, a heterogeneous CReAMS better uses the chip area as the functional units are better allocated.

This paper is organized as follows. Section 2 shows a review of existing work on adaptable organizations. Section 3 demonstrates the CReAMS organization. Section 4 discusses about the differences between homogeneous and heterogeneous configurations of CReAMS. Section 5 presents

the simulation environment and the results. Finally, the last section draws conclusions and introduces future work.

II. Related Work

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

The reconfigurable logic can be loosely connected to the processor as an I/O peripheral (communication is done through the main memory), attached as a coprocessor (communication is done by coprocessor-like protocols) or tightly coupled as a functional unit (reconfigurable logic is inside the processor and share its resources, like its registers). Furthermore, the granularity of the reconfigurable logic determines its level of data manipulation. A fine-grained logic is implemented at bit level (like FPGAs) while a coarse-grained logic implements word level circuits (like ALUs that execute the same operation in parallel over many bits).

There are many proposed reconfigurable architectures in the literature, as listed in [1]. One can find different single- and multi- processing environments which applies some kind of adaptability to improve the performance of applications [8][9]. They can be either homogeneous or heterogeneous, considering their architecture (i.e. what ISA is implemented) and organization (i.e. if the processors that comprise the system are the same or not).

Watkins [2] 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 [3]. It extends the Warp Processing [4] system to support multiple-thread execution. In this case, one processor is totally dedicated to execute the operating system tasks needed to synchronize threads and to schedule their kernels in the accelerators. KAHRISMA [5] is another example of a totally heterogeneous architecture. It supports multiple instruction sets (RISC, 2- 4- and 6-issue VLIW, and EPIC) and fine and coarse-grained reconfigurable arrays. Software compilation, ISA partitioning, custom instructions selection and thread scheduling are made by a design time tool that decides, for each part of the application code, which assembly code will be generated, considering its dominant type of parallelism and resources availability. A run time system is responsible for code binding and for avoiding execution collisions in the available resources. Both ReMAPP and KAHRISMA are able to optimize multiple threads, but they break the binary compatibility.

Therefore, the advantages of using CReAMS [13] over the architectures reviewed above include:

- Unlike KAHRISMA and Thread Warping, CReAMS is physically homogeneous in both architecture and organization. Heterogeneity is achieved on the fly, without any human intervention, by employing a binary translation. It eases the software development process since a well-known tool chain (i.e. gcc) is used for any of its versions. Neither source code modifications nor additional libraries are necessary if new processing elements are inserted.
- KAHRISMA, Thread Warping and ReMAPP rely in special and particular tool chains to extract thread-level parallelism and to prepare the platform for execution. CReAMS does not change the current development flow, so well-known application programming interfaces (e.g. OpenMP) can be used. This way, the programmer can extract TLP in a friendly interface, since such APIs are already coupled to a great number of compilers (e.g. gcc and icc), which makes the software development and the binary generation process easier than the aforementioned approaches.
- In contrast to ReMAPP and Thread Warping, CReAMS employs a coarse-grained reconfigurable fabric instead of a fine-grained one. Fine-grained architectures provide higher acceleration levels, but their scope is narrowed to applications that have few kernels responsible for a large part of the execution time. Coarse-grained reconfigurable architectures reduce the reconfiguration time and memory footprint due to the small context size, which increases its field of applications, because they are capable of accelerating the entire application.

III. CReAMS Organization

A general overview of CReAMS[15] is given in Figure 1(a). The thread level parallelism is explored by replicating the number of Dynamic Adaptive Processors (DAPs) (in the example of the Figure 1(a), by eight DAPs). In this way, CReAMS extends the single-thread based reconfigurable architecture presented in [6] to handle multithreaded applications. A DAP is a transparent coarse-grained reconfigurable architecture coupled to the processor, and will be explained in details later. The communication among DAPs is done through a 2D-mesh Network on Chip using a XY routing strategy. CReAMS also includes an on-chip unified 512 KB 8-way set associative L2 shared memory, illustrated as SM in the Figure 1(a). We divided DAP in four blocks to better explain it, as illustrated in Figure 1(b). These blocks are discussed in the following sections.

A SparcV8-based architecture is used as the baseline processor to work together with the reconfigurable system. Its five stage pipeline reflects a traditional RISC design (instruction fetch, decode, execution, data fetch and writeback) and it is similar to other RISC processors used in well-known embedded platforms (e.g. MIPS, ARM).

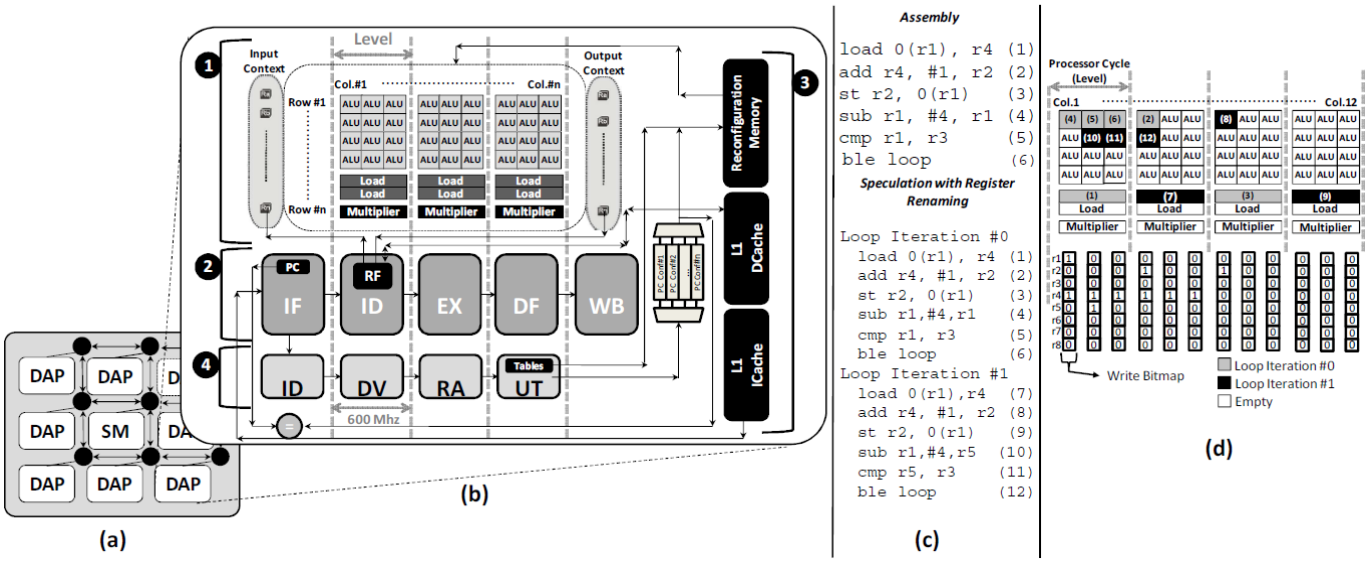

Figure 1: (a) CReAMS (b) DAP blocks (c) Assembly of a loop (d) Allocation inside of the reconfigurable data path

A. Reconfigurable Data Path Structure (Block 1)

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

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

The interconnection structure of the reconfigurable datapath is built using multiplexers to determine the dataflow into the functional units. More details are available in [14].

B. Processor pipeline (Block 2)

Block 2 is the basic processor to be used coupled with the array. It is also the reference point to determine the simulation performance of the tested configurations. In this work, the baseline processor is a SparcV8-based architecture.

C. Storage Components (Block 3)

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

D. Dynamic Detection Hardware (DDH) (Block 4)

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

For each column on the reconfiguration data path (Figure 1(b)), there is a bitmap responsible for storing in the target

operands of the already allocated instructions in the respective column, named as Write Bitmap (Figure 1(d)). Thus, for each incoming instruction, its source operands will be compared to the target operands in this bitmap to decide in which column this instruction will be allocated, according to data dependencies.

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

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

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

This hardware is capable of performing register renaming – for false dependency treatment – as well. In instruction number ten, the register r1 could be read by the incoming instruction in the second column, but could not write in this same register at this column. It 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.

IV. Homogeneous and Heterogeneous CReAMS

A. Homogeneous CReAMS

As already discussed, the homogeneous version of CReAMS is a configuration where all the DAPs are exactly the same. They all have the same size in area, the same memory size (L1 cache size, reconfiguration memory , number of input and output context, etc) and the same functional units. Figure 2(a) illustrates this concept. This configuration represents the traditional approach used in current multicore systems – in this case, DAPs would be generic cores instead. It is simple to implement, as no special scheduling is necessary (a thread is simply allocated to the next free DAP). However, this is not the most efficient approach, whereas a thread with low ILP will execute on the same environment as a thread with a high ILP.

B. Heterogeneous CReAMS

A heterogeneous version of CReAMS is a configuration where each DAP has a different number of functional units, input and output context length and memory size. This allows for some cores to be bigger than others, or in other words, more efficient to execute threads that can explore higher levels of instruction level parallelism. Similarly, smaller DAPs would be allocated to run threads with low ILP. Figure 2(b) shows a heterogeneous CReAMS of four cores where two of them are smaller than the others, one is of medium size and the last is bigger.

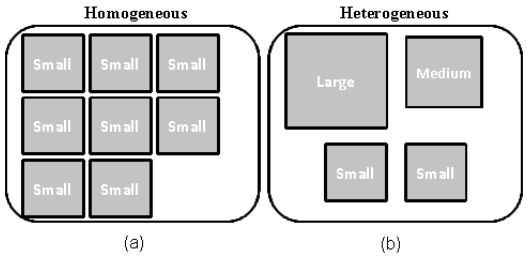

Figure 2: (a) homogeneous CReAMS composed of 8 small cores (b) heterogeneous CReAMS composed of 2 small cores, 1 medium and 1 large

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

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

V. Results

We have created a configuration of heterogeneous CReAMS and compared it with other two configurations of homogeneous CReAMS in terms of the total cycles taken to execute an application. The objective is to measure the potential gains of the heterogeneous configurations. In the homogeneous versions, a configuration with a smaller array (called SmallHomo) and one with a bigger array (called BigHomo) were created. Their resources are shown in Table 1. The heterogeneous configuration is composed of three sizes of arrays: a big, a medium and a small version (also shown in Table 1). These three sizes are distributed in 50% big arrays, 25% medium arrays and 25% small arrays. For example, a 4Hetero configuration would have 4 cores on which 2 of them have big arrays, 1 has a medium array and 1 has a small array, while an 8Hetero would have 4 cores with big arrays, 2 with medium and 2 with small.

The communication overhead is modeled as shown in [7]. In this model, the best and worst overhead cases are considered. The best case is when the data traffic is uniformly distributed among the NoC nodes, while the worst case happens when the traffic of data is concentrated from/to a specific node. Moreover, the scheduler of threads used in this

experiment is static, meaning that once a thread starts executing on a certain DAP it will keep executing on that DAP. This scheduler also does not try to allocate the thread to the best fitting core.

This work uses the Simics simulator [10] to generate the instruction trace from a set of applications. These instructions will then be split according to their threads and each of these threads will be allocated to a DAP simulator. Benchmarks of different suits were chosen to cover a wide range of applications in terms of parallelism exploitation (i.e. TLP and ILP). From parallel suits [11], we have selected the applications *lu* and *fft*. From SPEC OMPM2001 suit, we have selected *equake*, which was originally a single-threaded application that was ported to take advantage of multi-threaded environments. Finally, *susan_c* and *susan_e* were selected from the MiBench suit [12] and they represent a typical embedded scenario. *susan_c* and *fft* are benchmarks with great load balancing between the threads, making them good examples for TLP exploitation. *lu* has good load balancing for up to 8 threads. For more than that, however, the instructions are poorly distributed. Additionally, *susan_e*, *susan_c* and *lu* have big mean basic block sizes, which means that they can take advantage of ILP exploitation on bigger arrays, as they can occupy more resources on bigger configurations. Finally, *equake* has neither good load balancing nor big mean basic blocks, so it will not be much influenced by any kind of parallelism.

Figure 3 shows the results. We have simulated the homogeneous versions with up to 64 cores. Our goal here is to compare configurations of same area. As the SmallHomo configuration is about 4 times smaller than the Hetero, we have compared three multicore versions of them, whilst the BigHomo is only half the size of the Hetero configuration, so we have compared four multicore versions. This way, we guarantee that all the compared heterogeneous versions have the same area than their counterpart homogeneous configuration. Figure 3 shows the speedup of the heterogeneous configuration over the homogeneous in percentage. If the percentage is positive, the heterogeneous version is superior, while if it is negative, the homogeneous has better performance.

The results illustrate that, for some of the simulated applications, the Hetero configuration shows performance improvements over the homogeneous, especially in *lu* and *equake*. These benchmarks are not very influenced by the extra number of cores of the homogeneous configurations, but take advantage of the bigger arrays of the Hetero version. Considering the worst case communication scenario and the 16Hetero vs 64Homo, *lu* reaches performance gains up to 90%, while *equake* has gains of 66%. In the same situation, although *susan_c* and *fft* show loses, they are only of 0.5% and 2.6% respectively. *susan_c* and *fft*, are very parallel applications, as already mentioned, so the homogeneous versions – which have more cores – are superior on almost every case. However, when the worst case communication scenario is considered, the extra number of cores in the BigHomo version is actually harmful to the system and the 32Hetero configuration performs 23% and 21% (respectively) better than the 64BigHomo.

It is noticeable that in all scenarios where the homogeneous configuration performs better, this advantage decreases as the number of cores increases. This effect is also due to the communication overhead that is inserted in the system, as the data will need more hops on the NoC to reach the allocated core. Another reason for this performance loss is the load balancing of the applications. The TLP exploitation has limits for all the benchmarks tested and it is expected that, for an even higher number of cores, at some point the heterogeneous configurations are going to outperform the homogenous ones in every tested application.

TABLE I. CONFIGURATIONS

Config	Lines	Multipliers	Load/ Store	ALU	Cache L1	Input Context
Homogeneous						
Small Homo	9	2	1	3	32Kb	8
Big Homo	15	2	4	4	128Kb	16
Heterogeneous						
Small	9	1	2	3	32Kb	8
Medium	15	1	3	4	64Kb	12
Big	24	1	3	4	128Kb	24

VI. CONCLUSIONS AND FUTURE WORK

In this work we have shown the potential of using a heterogeneous CReAMS, as for some of the benchmarks, this configuration has shown performance improvements. However, we still have not reached the desired gains on most of the applications. For the next steps of this work, we will keep testing new combinations of heterogeneous configurations. There are many parameters that can be varied to reach better performance, for instance the number of functional units, the cache size and the input context size.

Furthermore, other actions can be taken to increase the heterogeneous performance. As discussed previously, a dynamic scheduler can be used to allocate threads with low ILP on smaller cores and threads with high ILP on bigger ones. Also, as the heterogeneous systems use the resources of the array more efficiently, it is expected that the power usage on this systems is lowered. In the next simulations we will consider a power budget as comparison scenario as well.

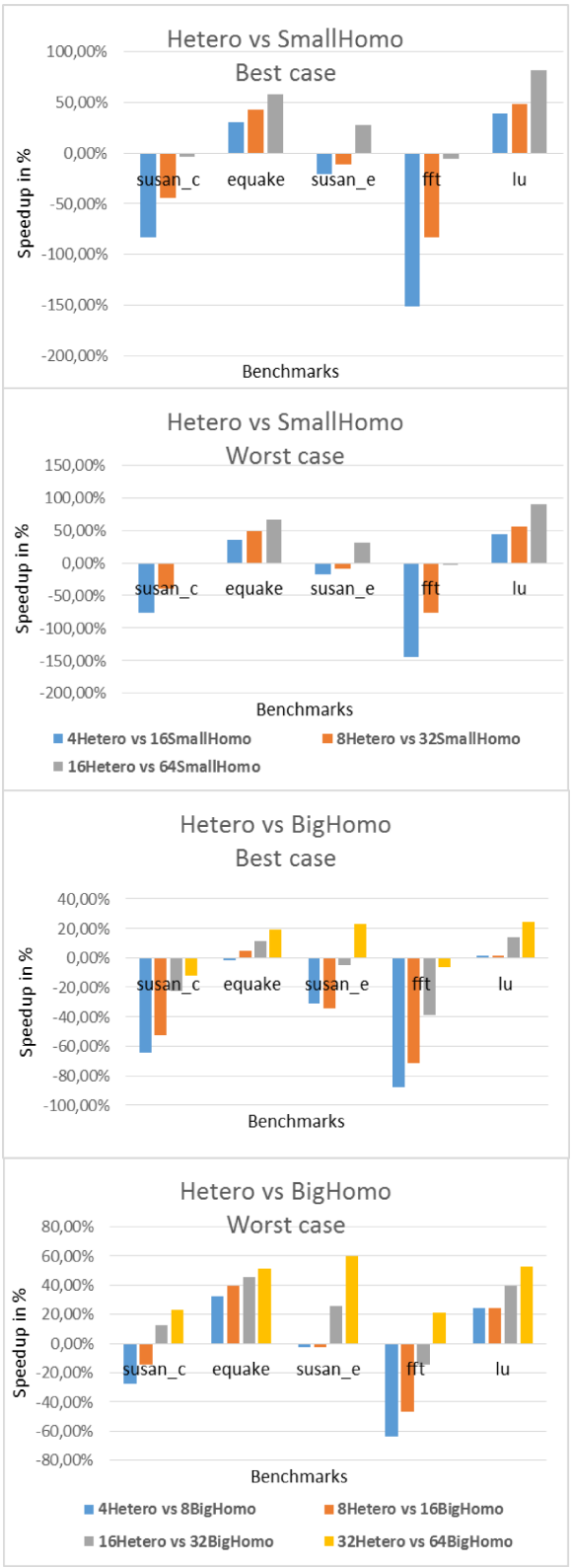

Figure 3: Performance gain in percentage of the heterogeneous configuration over the homogeneous.

REFERENCES

[1] Beck A.C.S., Lisbôa C.A.L., Carro L., *Adaptable Embedded Systems*, New York, Springer, 2013

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

[3] Lee, J., Wu, H., Ravichandran, M., and Clark, N. 2010. "Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications," ISCA '10. pp. 270-279.

[4] Roman Lysecky, Greg Stitt, and Frank Vahid. 2004. "Warp Processors," In Proceedings of the 41st annual Design Automation Conference (DAC '04). ACM, New York, NY, USA, 659-681.

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

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

[7] Rutzig, M.B.. "A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Efficient ILP and TLP Exploitation," Ph.D dissertation of Computer Science. Universidade Federal do Rio Grande do Sul (UFRGS) Porto Alegre, RS, Brazil 2012.

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

[9] Beck, A.C.S, Rutzig,M.B., Gaydadjiev,G., and Carro,L.."Run-Time Adaptable Architectures for Heterogeneous Behavior Embedded Systems," In Proceedings of the 4th international workshop on Reconfigurable Computing: Architectures, Tools and Applications (ARC '08).

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

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

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

[13] Rutzig,M.B., Beck, A.C.S and Carro,L.. "A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Simultaneous ILP and TLP Exploitation," In Proceedings of the conference on Design, Automation and Test in Europe (DATE'13). Grenoble, France. 1530-1591.

[14] Beck, A.C.S, Rutzig,M.B., and Carro L.. 2014. "A transparent and adaptive reconfigurable system," *Microprocess. Microsyst.* 38, 5 (July 2014), 509-524.

[15] Rutzig, M.B., Beck A.C.S., Carro L. "Creams: An embedded multiprocessor platform," in *Reconfigurable Computing: Architectures, Tools and Applications*, New York, Springer, 2011 pp. 118-124.