TLP and ILP exploitation through a Reconfigurable Multiprocessor System

Mateus B. Rutzig[1], Felipe Madruga[1], Marco A. Alves[1], Henrique Cota[1], Antonio C.S.Beck[2], Nicolas Maillard[1], Philippe O. A. Navaux[1], Luigi Carro[1]
[1]Universidade Federal do Rio Grande do Sul, Instituto de Informática – Porto Alegre/Brazil
[2]Universidade Federal de Santa Maria, Departamento de Eletrônica e Computação – Santa Maria/Brazil
{mbrutzig, flmadruga, mazalves, hcfreitas, caco, nicolas, navaux, carro}@inf.ufrgs.br

Abstract— **Limits of instruction level parallelism and the higher transistor density sustain the increasing need for multiprocessor systems: they are rapidly taking over both general purpose and embedded processor domains. Nowadays, since these processors must handle a wide range of different application classes, there is no consensus over which are the best hardware solutions to exploit the best of ILP and TLP together. Current multiprocessing systems are composed either of many homogeneous and simple cores, or of complex superscalar SMT processing elements. In this work, we have expanded a reconfigurable architecture to be used in a multiprocessing scenario, showing the need for an adaptable ILP exploitation even in TLP architectures. We have successfully coupled a dynamic reconfigurable system to a SPARC-based multiprocessor, and obtained performance gains of up to 40% even for applications that show a great level of parallelism at thread level, demonstrating the need for an adaptable ILP exploration.**

Keywords- reconfigurable architecture, SPARC, multicore.

I. INTRODUCTION

Industry competition in the current electronics market makes the design of a device increasingly complex. New marketing strategies have been focusing on increasing the product functionalities to attract consumer´s interest: they desire the equivalent of a supercomputer at the size of a portable device. However, the convergence of different functions in a single device produces new design challenges by enlarging the range of heterogeneous code that the system must handle. Nowadays, designers must take into account tighter design constraints as power budget and manufacturing process costs, all mixed up in the difficult task to increase the processing capability.

Because of that, instruction level parallelism (ILP) exploitation is no longer enough to improve performance of general and embedded applications. Newest ILP exploitation techniques do not provide an advantageous tradeoff between the amount of transistors added and the extra speedup obtained [1][2]. Despite the great advantages shown in the employment of some ISA extensions, like new SIMD instructions, such approaches rely on long design and validation times, which go against the need of a fast time-to-market for nowadays systems. On the other hand, ASIC provides high-performance and small chip area. However, such approach attacks only a very specific application class, failing to deliver the required performance when executing applications which behaviors were not considered at design time, making this approach not suitable for executing general-purpose applications.

Reconfigurable systems appear as a mid-term between general purpose processors and ASICs, solving somehow the ILP issues discussed before. They have already shown good performance improvements and energy savings for standalone applications in a single core environment [4][5][6][7]. Adaptable ILP exploitation is the major advantage of this technique, since the reconfigurable fabric can be dynamic restructured to fit the required application parallelism degree at a given time, enabling acceleration over a wide range of different application classes.

However, as already discussed, general-purpose and embedded systems are composed of a wide range of applications with different behaviors, in which the parallelism grain available varies from the finest to the coarsest. To accelerate applications that present high level of coarse-grained parallelism (at thread/process level), multiprocessor systems are widely employed, providing high performance and short validation time [3]. However, in contrast to architectures that make use of fine-grained parallelism (e.g. at instruction level) exploitation, such as the superscalar processors, the usage of the multiprocessor approach leaves all the responsibility of parallelism detection and allocation for the programmers. They must split and distribute the parallelized code among processing elements, handling all the communication issues. Software partitioning is a key feature in a multiprocessor system: if it is poorly performed, or if the application does not provide a minimum parallelism at process/thread levels, even the most computational powerful system becomes useless.

Thus, to cover all possible types of applications, the system must be conceived to have a good performance at any parallelism level and be adaptable to the running applications. Nowadays, at one side of the spectrum, there are the multiprocessing systems composed of many homogeneous and simple cores to better explore the coarse-grained parallelism of highly thread-based applications. At the other side, there are multiprocessor chips assembled with few complex superscalar/SMT processing elements, to explore applications where ILP exploration is mandatory. As can be noticed, there is no consensus on the hardware logic distribution to explore the best of ILP and TLP together regarding a wide range of application classes.

In this scenario, we merge different concepts by proposing a novel dynamic reconfigurable multiprocessor system. This system is capable of transparently explore (no changes in the binary code are necessary at all) the fine-grained parallelism of the individual threads, adapting to the available ILP degree, while at the same time taking advantage of available thread/process parallelism. This way, it is possible to have a system that adapts itself to any kind of available parallelism, handling a wide range of application classes.

The primary contributions of this work are:

- to reinforce, by the use of an analytical model, the need of heterogeneous parallelism exploitation in multiprocessor environments.

978-1-4244-6534-7/10/$26.00 ©2010 IEEE

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on April 13,2021 at 14:47:33 UTC from IEEE Xplore. Restrictions apply.

- to propose a multiprocessor architecture provided with an adaptable reconfigurable system, so it is possible to balance the best of both thread/process and instruction parallelism levels. This way, any kind of code, from those that present high TLP and low ILP to those that are exactly the opposite are accelerated.

II. RELATED WORK

The usage of reconfigurable architectures in a multiprocessor chip is not a novel approach. In [8] the Thread Warping system is proposed, which is composed of an FPGA coupled to an ARM11-based multiprocessor system. Thread warping uses complex CAD tools to detect, at execution time, critical regions and map them to custom accelerators implemented in a simplified FPGA. After CAD synthesis, a greedy knapsack heuristic is used to find the best possible allocation of custom accelerators onto FPGA, taking into account the possibility of partial reconfiguration. One processor is totally dedicated to run the operating system tasks needed to synchronize threads and schedule their kernels to be executed on the accelerators. However, the processor responsible for the scheduling may become overloaded if several threads are running on tens or hundreds of processors, affecting system scalability. Besides, due to the high overhead time imposed by the CAD and greedy knapsack algorithm, only critical code regions are optimized, so only applications with few and very defined kernels (e.g. filters and image processing algorithms) are accelerated, narrowing its field of application.

Another study on sharing reconfigurable fabric in a multiprocessor design is shown in [10]. The authors claim that area and energy overhead are barriers when reconfigurable fabrics are used as a private accelerator for all processing elements of a multiprocessor designs. Thus, this work shows the need of reconfigurable fabric being shared among processing elements. Results of area and power reduction are demonstrated on sharing temporally and spatially the reconfigurable fabric. However, both private and shared approach relies on compiler support, breaking the binary compatibility feature and affecting time-to-market due to larger design times.

In [9] it is presented the Annabelle SoC that is comprises an ARM core and four domain specific coarse-grain reconfigurable architectures, called Montium cores. Each Motium core is composed of five 16-bit ALUs structured to accelerate DSP applications. The ARM926 is responsible for the dynamic reconfiguration processes by executing the run-time mapping algorithm, which is used to determine a near-optimal mapping of the applications to the Montium cores. Although the authors discuss the need of heterogeneous parallelism exploitation on a multiprocessor environment to achieve speedups over a wide range of applications classes, in that specific work the Annabelle SoC was specifically designed to speed up only DSP applications (e.g. FFT, FIR and SISO algorithm).

In this work, we address the particular drawbacks of the above approaches by creating an adaptable reconfigurable multiprocessing system that:

- unlike [8][9], provides lower reconfiguration times, thus allowing ILP investigation/acceleration of the entire application code, including highly thread-parallel algorithms;
- unlike [10], maintains binary compatibility applying a lightweight dynamic detection hardware that, at run-time, recognize potential parts of code to be executed on the reconfigurable data path;
- because of the two aforementioned features, in opposite to the previous works, it can speedup even applications that present the worst scenario for ILP exploitation: highly parallel application containing small parts of sequential code.

Furthermore, we demonstrate through an analytical model the need of mixed parallelism exploitation (ILP and TLP) regarding a multiprocessing environment.

III. ANALYTICAL MODEL

In this section, we try to define the design space for multiprocessor-based architectures. First, we model a multiprocessing architecture (MultiProcessor) made of many simple and homogeneous cores, and compare it to the modeling of a high-end single-core processor with a great ILP exploration capacity (Single High-End) in terms of performance.

Let us start with the basic equation relating time with instructions,

$$ExecutionTime = Instructions * CPI * CycleTime\ , \ (1)$$

where *CPI* is the mean amount of cycles taken to execute a instruction in a certain benchmark. In case of a high-end ILP exploitation architecture, one can write

$$T_{SHE} = Instructions\left(\frac{\propto CPI_{SHE}}{issue} + \beta\ CPI_{SHE}\right) CycleTime_{SHE}, (2)$$

Where CPI_{SHE} is the mean amount of cycles that ILP to execute instructions for a certain benchmark. Variable *issue* is the amount of parallelism that can be obtained in the high-end single processor in the best-case situation, without data or control dependencies in the code. Theoretically, this part of the code could be accelerated by the amount of parallelism available in the high-end processor. CPI_{SHE} can be obtained from benchmarks, and is usually smaller than 1, because the high-end single processor can accelerate code with lots of control dependencies, thanks to branch prediction and false data dependencies. A typical value for a current high-end single processor CPI_{SHE} would be 0.62 [11], showing that more than one instruction can be executed at any cycle.

Coefficients α and β refer to the percentage of instructions that has ILP or not ($\alpha + \beta = 1$). Typical values are α=0.62 and β=0.38 for the MiBench [11] which is a benchmark composed of a wide range of embedded application classes. This division of instructions on code that can be parallelized or code that can only be accelerated by speculation is important, because it states one of the major differences

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on April 13,2021 at 14:47:33 UTC from IEEE Xplore. Restrictions apply.

between superscalar and in-order-processors. $CycleTime_{SHE}$ represents the clock cycle of the single high-end processor.

It is clear that in this modeling no information about cache misses is used, nor the performance of the disk or I/O is taken into account. Nevertheless, although simple, this modeling will give some interesting clues on the potential of multiprocessing architectures for a wide range of applications classes.

Having stated the equation for high-end single core performance, one must look now for the potential use of a multiprocessing architecture. A homogeneous multiprocessor chip will have to be built by replication of simple (low-end) processors. Moreover, the advantage of using multiprocessing is based not only on the available ILP, but mostly on the potential thread level parallelism (TLP). Some works [12] can even translate code with enough ILP into TLP, so that more than one core can execute the code, exploiting ILP also at the single-issue core level, when embedded in a multiprocessing device.

For a single low-end core processor (SLE), the performance equation can be written as

$$T_{SLE} = Instructions(\propto CPI_{SLE} + \ \beta CPI_{SLE}) CycleTime_{SLE}, (3)$$

and since in an multiprocessor environment must be composed of simple, so that a large number of them can be integrated, the CPI_{SLE} of the low-end single core is bigger than 1, and reach 3 or more for a processor without branch prediction and a 5 stage pipeline. Since there is no available parallelism at the instruction level that can be used in the low-end simple core with only a single execution flow, the separation in α and β instructions types does not makes much sense (but we will keep the notation for comparison purposes). On the other hand, using the techniques proposed in [10], there are a certain number of instructions that can be split into several processors, and hence one could write

$$T_{MP} = Instructions\left(\frac{\delta}{P} + \gamma\right)(\alpha CPI_{SLE} + \beta CPI_{SLE})\ CycleTime_{MP}, (4)$$

where δ is the amount of code that has TLP and can be transformed into multithreaded code, while γ is the part of the code that must be executed sequentially (no TLP can be obtained). P is the number of homogeneous processor that is available in the chip. Hence, equation (4) reflects the fact that in a multiprocessor environment one could benefit from thread level parallelism, but increasing the number of processors can only accelerate parts of the code that can be parallelized at thread level.

One important aspect also is that simpler processors could also run at much higher frequencies than high-end single core processors, since their organizations reflect smaller area and power consumption. However, the total power budget will probably be a limiting factor. For the sake of the modeling, we will assume that

$$CycleTime_{MP} = K * CycleTime_{SHE}, (5)$$

Based on the above reasoning, one can compare the performance of the single high-end processor with the one of the multiprocessor system:

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

and by simplifying (6) one gets

$$= \left[1/\frac{\delta}{P} + \gamma\right]\left[\frac{\propto \frac{CPI_{SHE}}{issue} + \beta CPI_{SHE}}{\propto CPI_{SLE} + \beta CPI_{SLE}}\right]\left[\frac{1}{K}\right], (7)$$

It is clear from equation (7) that although the low-end cores have a faster cycle (by a factor of K), that factor is not enough to define performance. Regarding the second term on equation (7), the fact that the high-end processors can execute many instructions in parallel could give a better performance. In the extreme case, let us imagine that *issue*=*P*=∞, meaning that we have infinite resources, either in the form of arithmetic operators or in the form of processors. This would reduce equation (7) to

$$\frac{T_{SHE\infty}}{T_{MP\infty}} = \left[\frac{1}{\gamma}\right]\left[\frac{\beta CPI_{SHE}}{\propto CPI_{SLE} + \beta CPI_{SLE}}\right]\left[\frac{1}{K}\right], (8)$$

Equation (8) clearly shows that, as long as one has code which carries control or data dependencies, and cannot be parallelized (at the instruction or thread level), a machine based on a high-end single core will always be faster than a multiprocessor-based machine, regardless of the amount of available resources.

An interesting question is: when the performance of the high-end single core is equal to the performance of the multiprocessor core. One can write $T_{SHE} = T_{MP}$, hence

$$\left[\left(\propto \frac{CPI_{SHE}}{issue} + \beta CPI_{SHE}\right)\frac{1}{K}\right] = \left[\left(\frac{\delta}{P} + \gamma\right)(\alpha CPI_{SLE} + \beta CPI_{SLE})\right], (9)$$

From equation (9) one can see that one must have enough low-end single processors combined to a highly parallel code (greater δ) to overcome the high-end processors advantage.

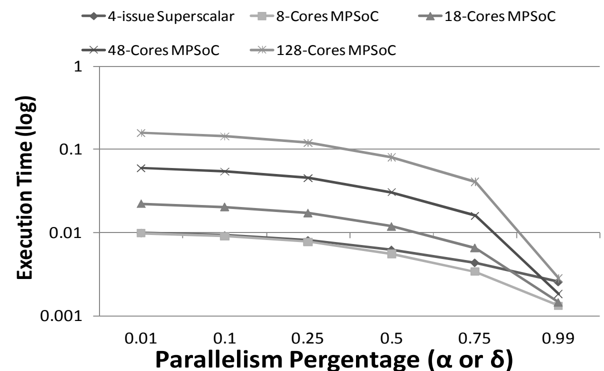

Figure 1. Multiprocessor system and Superscalar performance regarding a power budged using different ILP and TLP; α = δ is assumed.

Given the theoretical model, one can briefly test it with some numbers based on real data. Let us consider a high-end

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on April 13,2021 at 14:47:33 UTC from IEEE Xplore. Restrictions apply.

single core the 4-issue MIPS R10000 superscalar processor, and a multiprocessor design composed of replications of the low-end MIPS R3000 processor. A comparison between both architectures is done using the equations of the proposed analytical model. Figure 1 shows, in a logarithmic scale, the performance of the superscalar processor when parameters and change, assuming the 4-issue core, with CPI 0.6. In addition, also in Figure 1 we show the performance of many cores within the multiprocessor design, varying the and parameters and the number of in-order processors from 8 to 128. The goal of this comparison is to demonstrate which technique better explores parallelism at different levels, considering six values for both ILP and TLP. For instance, = 0.01 means that an application only shows 1% thread level parallelism (TLP) within its code. When =0.01 it is assumed that 1% of instruction level parallelism (ILP) is provided in the application code, that is, only 1% of its instructions can be executed in parallel. The MIPS R3000 core is a simple in-order core that has CPI=1.8. Its hardware takes only 75,000 transistors [13], almost 29 times less than the 2.2 millions of transistors used on the MIPS R10000 [14]. Following the same strategy found in current processor designs, for a fair performance comparison we considered the same power budged for the high-end single core and the multiprocessor approaches. In order to normalize the power budged of both approaches we have tuned the multiprocessor design (replication of low-end processors) clock frequency to achieve the same power of the superscalar processor.

Regarding applications that reflect minimum ILP (α=0.01) or TLP (δ =0.01) factors, the superscalar processor and an 8-Cores design present almost the same performance. However, when applications show greater parallelism percentage (α>0.25 and δ>0.25), the 8-Cores design well handles the TLP parallelism, surpassing the ILP exploitation in the 4-issue superscalar processor. Due to the power budged assumed, when more cores are inserted in a multiprocessor design the overall clock frequency tends to decrease. The performance of applications with low thread level parallelism (small δ) worsens with the increase in the number of cores. Regarding the applications with δ =0.01 in Figure 1, lots of performance is wasted as the number of cores increases. Nevertheless, as the application thread level parallelism increases, (δ > 0.01) the negative impact on performance is softened, since the additional cores have better use.

Aiming to make a fair performance comparison among high-end single core and multiprocessor approaches, we have devised an 18-Core design composed of low-end processors that presents the same area of 4-issue superscalar processor. Since MIPS R10000 is 29 times bigger than MIPS R3000, for a fair comparison we considered that the intercommunication mechanism takes an area of almost 37% of chip, since this is also the number presented in [15]. The performance of both approaches shows the powerful capabilities of the superscalar processor, since it can accelerate code with different levels of available parallelism, thanks to speculation and register renaming. As shown in Figure 1, the multiprocessor approach only surpasses the superscalar performance when the TLP level is greater than 85% regarding the same area for both designs.

Summarizing the comparison with the same power budged, the superscalar machine shows better performance over application with low thread level parallelism. Regarding multiprocessor designs there is an additional tradeoff, since when more cores are included in the chip, the MPSoC performance tends to worsen, since more core means more power, and hence the cores must be run at lower frequency to obey the power budged. When almost the whole application supports TLP exploitation (δ > 0.99), the 128-Cores design takes a bigger execution time than the other multiprocessor designs.

Regarding real applications, thread level parallelism exploitation is widespread employed to accelerate most multimedia and DSP applications thanks of their data independent iteration loops. However, even applications with high TLP could still obtain some performance improvement by also exploiting ILP. Hence, in a multiprocessor design ILP techniques also should be investigated to conclude what is the best fit concerning the design requirements. On the other hand, tightly code dependent applications, as database applications, turn most processing elements into idle mode, and need ILP exploitation to supply its poor coarse-grain parallelism. Hence, the analytical modeling indicates that heterogeneous multiprocessor system is primary to balance the performance of a wide range of application classes. Section 6 reinforces this trend running real applications over a multiprocessor design coupled to an adaptable ILP exploitation approach.

IV. RECONFIGURABLE MULTIPROCESSING SYSTEM

Section 3 demonstrated the need to explore different grains of parallelism to balance performance, showing that TLP and ILP exploitation are complementary, regarding a heterogeneous application environment. Aiming to reproduce the analytical model on a real application workload, we have replicated a reconfigurable system core, building a reconfigurable multiprocessor structure shown in the bottom of Figure 2(a).

In the right side of this Figure is shown one reconfigurable core that is composed of four major blocks. Block Number 1 depicts the reconfigurable data path that aggregates the input context, output context and the functional units. Block Number 2 presents the processor pipeline composed of five stages. Block Number 3 illustrates the pipeline stages of the Dynamic Detection Hardware (DDH) that works in parallel to the processor pipeline. It is responsible for building instruction blocks, called configurations, which will be executed in the reconfigurable data path. Block Number 4 demonstrates the storage components. The reconfiguration memory holds the configuration bits previously generated by the DDH, so they can be reused next time they are found. This way, in opposite to superscalar processors, the work of extracting the ILP of a certain sequence of code is done only once. Besides, there is a 4-way associative cache called Address

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on April 13,2021 at 14:47:33 UTC from IEEE Xplore. Restrictions apply.

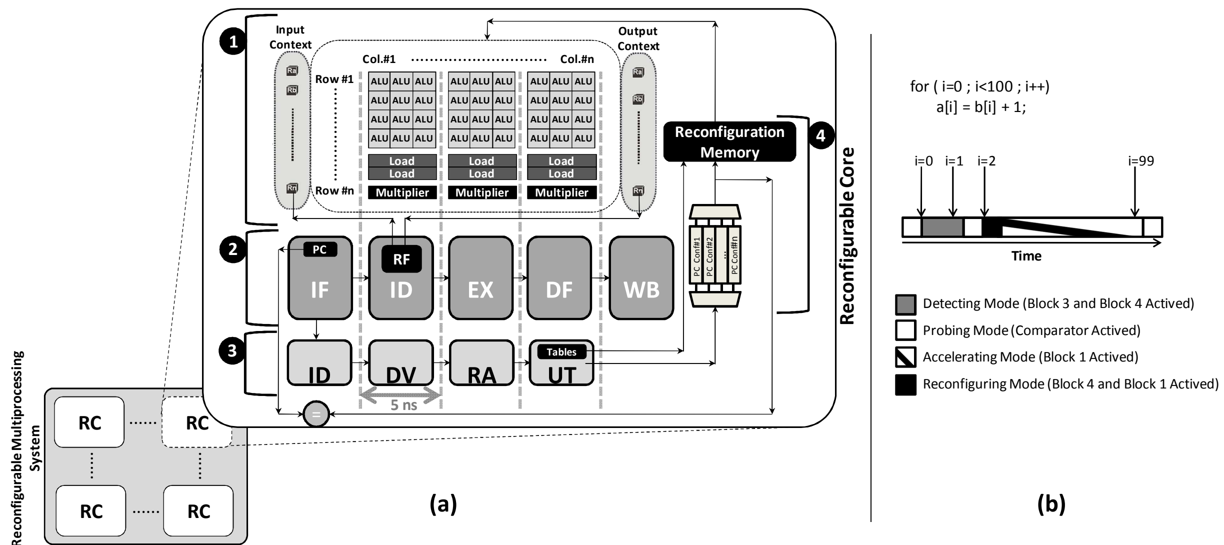

Figure 2. (a) Reconfigurable Multiprocessor System Components (b) Acceleration process example

Cache that stores, for each configuration, the instruction memory address of the first instruction of the translated sequence. More details about these components are presented in the next sections.

Figure 2(b) shows an example of how a loop would be accelerated using the proposed process. The reconfigurable core (RC) works in four modes: probing, detecting, reconfiguring and accelerating. At the beginning of the time bar shown in Figure 2(b), the RC is searching for an already built configuration to accelerate. When the first loop iteration appears (when i=0), the DDH detects a new code to translate and it changes to detecting mode. In that mode, while the instructions are being translated to a configuration in the DDH, they are executed in the processor pipeline. When the second loop iteration comes to the processor pipeline (i=1), the DDH closes the previous configuration (i=0) and stores it into the reconfiguration memory, updating the Address Cache with the instruction memory address of the first instruction of this sequence. Then, when the first instruction of the third loop iteration comes to the processor pipeline, the probing mode detects a valid configuration in the reconfiguration memory. The RC enters in reconfiguring mode to feed the reconfigurable data path with the operands and the reconfiguration bits. After that, the accelerating mode is activated and the next iterations (until the 99th) are executed efficiently into the reconfigurable data path.

4.1. Reconfigurable Data Path Structure (Block 1)

Following the classifications shown in [16], the reconfigurable data path is tightly coupled to the processor pipeline. Such coupling approach avoids external accesses to the memory, saving power and reducing the reconfiguration time. Moreover, its coarse-grained nature decreases the size of the memory necessary to keep one configuration, since the basic processing elements are functional units that work at the word level (arithmetic and logic, memory access and multiplier), which need only a few bits to select the desired operation. The data path is organized as a matrix of rows and columns, composed of functional units. The number of basic rows dictates the maximum instruction level parallelism that can be exploited, since instructions placed in the same column are executed concurrently (in parallel). The number of columns, in turn, determines the maximum number of dependent instructions placed into one configuration. For example, the data path shown in Figure 2(a) will execute up to four arithmetic and logic operations, two memory accesses (reflecting two memory ports) and one multiplication in parallel. It is important to notice that simple arithmetic and logic operations can be executed within the same processor cycle without affecting the critical path. Consequently, some data dependent instructions are also accelerated. Memory accesses and multiplications take one equivalent processor cycle to perform their operations.

The entire structure of the reconfigurable data path is totally combinational: there is no temporal barrier between the functional units. The only exception is for the entry and exit points. The entry point is used to load the input context, which is connected to the processor register file. The operand fetch from the register file is the first step to configure the data path before actual execution. After that, results are stored in the output context registers through the exit point of the data path. Finally, the values stored in the output context registers are send to the processor register file.

4.2 Processor Pipeline (Block 2)

A SPARC-Based architecture is used as the baseline processor to support the reconfigurable system. Its five stage pipeline reflects a traditional execution flow (instruction fetch, decode, execution, data fetch and write back) of a SparcV8 instruction set.

4.3 Dynamic Detection Hardware (Block 3)

As explained before, the dynamic detection hardware (DDH) can work in four modes: detecting , probing, accelerating and reconfiguring. As can be observed in Figure 2 (a), the detecting mode contains four pipeline stages:

- Instruction Decode (ID) –In this stage, the instruction is broken into operation, source operands and target operand.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on April 13,2021 at 14:47:33 UTC from IEEE Xplore. Restrictions apply.

- Dependence Verification (DV) - the source operands are compared to the target operands of previously detected instructions to verify which column the current instruction should be allocated, according to their data dependencies. The placement algorithm is very simple: the DV stage only indicates the leftmost data path column that the current instruction should be placed.
- Resource Allocation (RA) – In this stage the data dependence is already solved and the correct data path column is known. The RA stage verifies the resources availability linking the instruction operation with the correct type of functional unit. If there is no functional unit available at this column, the next column at the right side is candidate to the current instruction placement.
- Update Tables (UT) – at the beginning of UT stage, the functional unit responsible for the current instruction execution is already defined. Thus, this stage configures the routing to feed that functional unit with the correct source operands from input context and to write the result in the correct register of the output context. After that, the bitmaps and tables are updated and the configuration is closed, meaning that their configuration bits are sent to the reconfiguration memory and for the Address Cache is updated.

Figure 3 illustrates, by an activity diagram, the DDH process for creating a configuration. Summarizing the detection, the first step is the execution support verification. Thus, if there is no compatible functional unit to execute such operation (e.g. division or branch), the configuration is broken and the next instruction starts a new configuration. If there is support, the data dependency among previously allocated instructions is verified (DV stage) and the correct column is defined. Thus, the next step looks for an empty functional unit traveling from the defined up to the rightmost column. If there is no functional unit available, the current configuration is sent to the reconfiguration memory. After, a new configuration is created to allocate the instruction that caused a functional unit miss. Otherwise, if an empty functional unit is found, the instruction is allocated in the current configuration, and the next instruction, if there is any, starts the process on the first step (execution support verification).

4.4 Storage Components (Block 4)

Two storage components are part of the dynamic data path reconfiguration: Address Cache and Reconfiguration Memory. The reconfiguration memory stores the routing bits and the necessary information (e.g. input context, immediate values, and output context) to fire a configuration. The configurations are indexed by the main memory address of its first instruction detected. For this work, the Address Cache is implemented as a 128 entries 4-way associative cache to hold those addresses. The Address Cache only is accessed when the DDH is working in the probing mode. An Address Cache hit indicates a configuration found which changes the system to the reconfiguring mode. In this mode, the reconfiguration memory is accessed to feed the data path routing. In addition, the necessary data to fire that configuration fetched from the register file and stored in the input context. Thus, the DDH hardware changes to accelerating mode beginning an execution process in the reconfigurable data path.

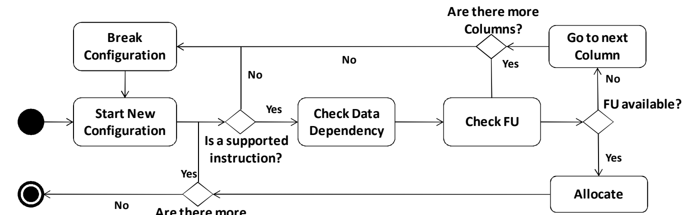

Figure 3. DDH activity diagram

V. SIMULATION ENVIRONMENT

5.1. Methodology

To simulate the reconfigurable multiprocessor system we have used the scheme presented in [17]. Basically, it consists of a functional full-system [18] and cycle-accurate timing simulators [22] for the individual reconfigurable cores. While the application runs on the functional simulator, instructions of each software thread are sent to the cycle-accurate timing simulators. Special attention is given to synchronization mechanisms, such as locks and barriers, so the elapsed time regarding blocking synchronization and memory transfers are precisely calculate.

To demonstrate the impact of ILP exploitation on the performance we have used two reconfigurable cores configurations, changing the number of basic functional units that compose their reconfigurable data path. Table 1 presents these setups. We also considered different reconfigurable multiprocessor architectures changing their number of cores.

Table 1. Number of basic functional units of setups

	RC#1	RC#2
Columns	48	150
ALU/Column	6	8
Load/Column	4	6
Mul/Column	1	2

5.2. Workload

A workload of parallel programs with very distinct behaviors was created using benchmarks from the well-known SPLASH2 [19] and PARSEC [20] suites. In addition, two numerical applications written in OpenMP were used [21].

The list below briefly describes each of them.

- *FFT* [19]: A complex 1-D version of a six-step FFT algorithm.
- *LU* [19]: Factors a dense matrix in the equivalent lower-upper matrix multiplication.
- *Blackscholes* [20]: It solves the partial differential equation of *Blackscholes* in order to compute prices for a portfolio of European options.
- *Swaptions* [20]: Monte Carlo simulation is used to price a portfolio of *swaptions*.
- *Molecular Dynamics* (*MD*) [5]: It implements the velocity Verlet algorithm for molecular dynamics simulation.
- *Jacobi* [21]: It utilizes the Jacobi iterative method to solve a finite difference discretization of Helmholtz.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on April 13,2021 at 14:47:33 UTC from IEEE Xplore. Restrictions apply.

The proposed workload, which is composed of only high parallel applications, was chosen to show the need for an adaptable ILP exploitation even for high TLP-based applications.

VI. RESULTS

This section demonstrates the performance evaluation of the reconfigurable multiprocessing system over three different aspects: TLP exploitation by changing the number of cores from four up to 64 – in these experiments, standalone SPARC cores are used: they are not coupled to the reconfigurable architecture; TLP + ILP exploitation, repeating the previous experiment but now using the SPARC cores together with the reconfigurable architecture in the two different versions (RC#1 or RC#2); and the influence of changing the applications' data input sizes on performance.

Figure 4 shows the performance speedups in two different scenarios: by exploring only TLP, varying the number of standalone SPARC cores (solid bars), and by coupling the reconfigurable architecture (RC#1) to each one of these cores, so the ILP is also exploited (striped bars). Regarding only TLP exploitation, the performance scales linearly as the number of cores increases. *FFT* and *LU* do not follow this behavior, since their codes present a significant part of sequential code, responsible for data initialization.

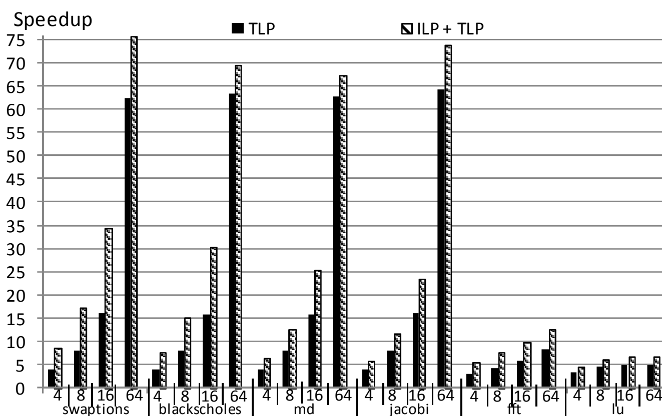

Figure 4. Performance of TLP and ILP+TLP exploration using RC#1 setup

As can be observed, the results reinforce the conclusion gathered from the analytical model in Section 3: even for high TLP-based applications, there is a need of finer grain parallelism exploitation to complement the TLP gains. Table 2 shows the average speedup of both approaches. This Table demonstrates that TLP+ILP exploitation using the RC#1 setup composed of four cores presents similar performance gains comparing to eight standalone cores exploiting only TLP parallelism. The same occurs when comparing a system with 8 cores and the RC#1 setup to 16 standalone cores.

Table 2. Average speedup on different number of cores

	4 cores	8 cores	16 cores	64 cores
TLP	3.74	6.86	12.47	44.30
ILP + TLP	6.46	11.85	21.71	51.00

Figure 5 compares the performance of a system composed of 4 or 8 cores, in which each is coupled to the RC#2 over the system in which the cores are coupled to the RC#1. The improvement is negligible, and not proportional to the additional number of basic functional units. This happens because of the high TLP degree presented in the selected workload. Their threads do not present enough instructions that can be accelerated by the additional basic functional units available in RC#2, so the amount of basic functional units of RC#1 is adequate to satisfactorily explore the ILP available in most applications of the selected workload. *Molecular Dynamics* (*MD*) is the one that best takes advantage of the extra units of the RC#2, although it presents only 5% of performance improvements than RC#1. *Jacobi* and LU executions show performance losses when using the RC#2 setup. The reason for that is because the DDH produces a different amount of configurations for both setups. This fact affects the Address Cache storage producing more configuration misses and, consequently in the RC#2 setup. Because RC#2 does not show any significant advantage over the RC#1, the RC#1 is used for the remaining experiments.

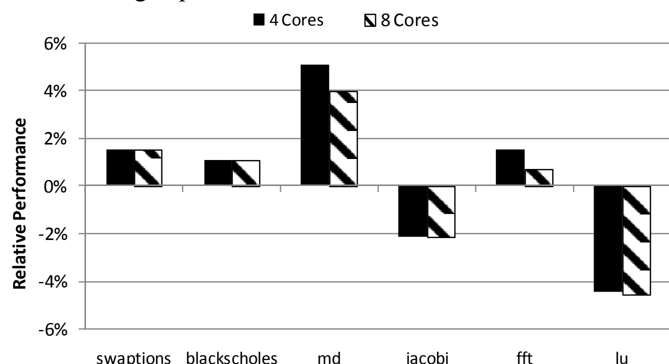

Figure 5. Performance Comparison among RC#1 and RC#2 setups

Figure 6 shows the performance evaluation when running the same application workload with different data input sizes, so it is possible to investigate if the data input size of the applications affects the performance results shown in Figure 4. Two different data input sizes were used. As can be observed, changing the data input size does not affect performance for most applications.

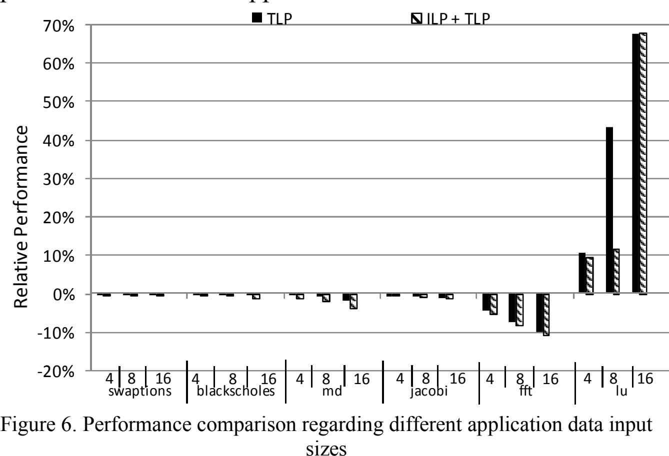

Figure 6. Performance comparison regarding different application data input sizes

On the other hand, *FFT* and *LU* present a significant impact when changing the data input size. Figure 7 shows this data in more details. As discussed before, *FFT* has a

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on April 13,2021 at 14:47:33 UTC from IEEE Xplore. Restrictions apply.

significant amount of sequential code responsible for data initialization. Thus, when we increase the data input size, the initialization becomes more significant over the whole application execution time affecting the application performance. This behavior is more evidenced in the multiprocessing system composed of 16 cores.

Regarding *LU*, the chosen data input size provides a perfect load balance with greater number of processors [19]. Small data input sizes increases the imbalance by splitting less blocks per processor in each step of the factorization. Figure 7 evidences that *LU* presents better speedups for greater data input sizes since it better handles the parallel load-balancing problem.

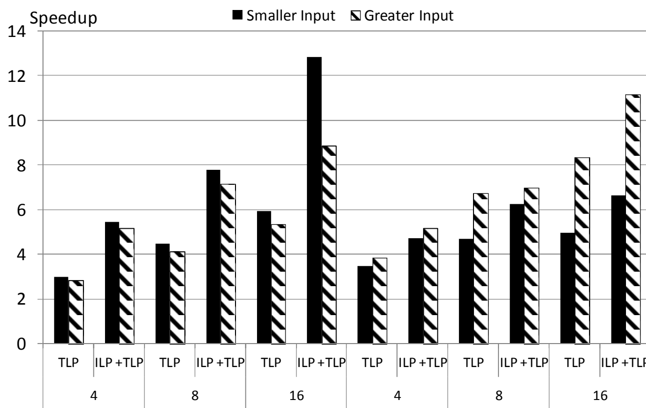

Figure 7. FFT and LU performance regarding different application data input sizes

VII. CONCLUSIONS

As the instruction level parallelism (ILP) exploitation is no longer an efficient technique to improve performance of general and embedded processors, multiprocessing system appears as a solution to accelerate applications exploring coarser grain parallelism. This paper shows the need of a mixed grain parallelism exploitation to achieve a balanced performance over applications with heterogeneous behaviors. The coupling of an adaptable ILP exploitation on a multiprocessing environment has presented suitable speedups even for TLP-based applications.

VIII. REFERENCES

[1] Wall, D. W. 1991. Limits of instruction-level parallelism. *SIGPLAN Not.* 26, 4 (Apr. 1991), 176-188.

[2] Mak, J..,Mycroft,A. Limits of instruction data dependence graphs. *WODA '09,* July 20, 2009, Chicago, Illinois, USA.

[3] Olukotun, K. 2007 *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. 1st. Morgan and Claypool Publishers.

[4] Baumgarte, V., Ehlers, G., May, F., Nückel, A., Vorbach, M., and Weinhardt, M. 2003. PACT XPP—A Self-Reconfigurable Data Processing Architecture. J. Supercomput. 26, 2 (Sep. 2003), 167-184

[5] Patel, S. J. and Lumetta, S. S. 2001. rePLay: A Hardware Framework for Dynamic Optimization. IEEE Trans. Comput. 50, 6 (Jun. 2001), 590-608.

[6] Hauser, J. R. and Wawrzynek, J. Garp: a MIPS processor with a reconfigurable coprocessor. In Proceedings of the 5th IEEE Symposium on Fpga-Based Custom Computing Machines. FCCM. IEEE Computer Society, Washington, DC, 12.

[7] Lysecky, R., Stitt, G., and Vahid, F. 2004. Warp Processors. In Proceedings of the 41st Annual Conference on Design Automation. DAC '04. ACM, New York, 659-681.

[8] Stitt, G. and Vahid, F. 2007. Thread warping: a framework for dynamic synthesis of thread accelerators. In *Proceedings of the 5th IEEE/ACM international Conference on Hardware/Software Codesign and System Synthesis* (Salzburg, Austria, September 30 - October 03, 2007). CODES+ISSS '07. ACM, New York, NY, 93-98

[9] Smit, G. J., Kokkeler, A. B., Wolkotte, P. T., and van de Burgwal, M. D. 2008. Multi-core architectures and streaming applications. In *Proceedings of the 2008 international Workshop on System Level interconnect Prediction* (Newcastle, United Kingdom, April 05 - 08, 2008). SLIP '08. ACM, New York, NY, 35-42

[10] Watkins, M.A.; Cianchetti, M.J.; Albonesi, D.H., "Shared reconfigurable architectures for CMPS," Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on , vol., no., pp.299-304, 8-10 Sept. 2008

[11] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. Wwc-4. 2001 IEEE international Workshop* (December 02 - 02, 2001). WWC.Washington, DC, 3-14.

[12] Song, Y., Kalogeropulos, S., and Tirumalai, P. 2005. Design and Implementation of a Compiler Framework for Helper Threading on Multi-core Processors. In *Proceedings of the 14th international Conference on Parallel Architectures and Compilation Techniques* (September 17 - 21, 2005). PACT. IEEE Computer Society, Washington, DC, 99-109.

[13] Rowen, C.; Johnson, M.; Ries, P., "The MIPS R3010 floating-point coprocessor," Micro, IEEE , vol.8, no.3, pp.53-62, June 1988

[14] Yeager, K.C. The Mips R10000 Superscalar Microprocessor,; IEEE Micro, pp. 28-40, Apr. 1996

[15] Available at http://blogs.intel.com/research/2007/07/inside_the_ terascale_many_ core.php

[16] Compton K., Hauck, S., "Reconfigurable computing: A survey of systems and software". In ACM Computing Surveys, vol. 34, no. 2, pp. 171-210, June 2002.

[17] Monchiero, M., Ahn, J., Falcón, A., Ortega, D., and Faraboschi, P. 2009. How to simulate 1000 cores. *SIGARCH Comput. Archit. News* 37, 2 (Jul. 2009), 10-19.

[18] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., and Werner, B. 2002. Simics: A Full System Simulation Platform. *Computer* 35, 2 (Feb. 2002), 50-58.

[19] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual international Symposium on Computer Architecture* (S. Margherita Ligure, Italy, June 22 - 24, 1995). ISCA '95. ACM, New York, NY, 24-36.

[20] Bienia, C., Kumar, S., Singh, J. P., and Li, K. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada, October 25 - 29, 2008). PACT '08. ACM, New York, NY, 72-81.

[21] Dorta, A. J., Rodriguez, C., Sande, F. d., and Gonzalez-Escribano, A. 2005. The OpenMP Source Code Repository. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing* (February 09 - 12, 2005). PDP. IEEE Computer Society, Washington, DC, 244-250.

[22] Rutzig, M. B., Beck, A. C., and Carro, L. 2009. Dynamically Adapted Low Power ASIPs. In *Proceedings of the 5th international Workshop on Reconfigurable Computing: Architectures, Tools and Applications* (Karlsruhe, Germany, March 16 - 18, 2009). J. Becker, R. Woods, P. Athanas, and F. Morgan, Eds. Lecture Notes In Computer Science, vol. 5453. Springer-Verlag, Berlerroin, Heidelberg, 110-122.

Authorized licensed use limited to: UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Downloaded on April 13,2021 at 14:47:33 UTC from IEEE Xplore. Restrictions apply.