Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 546962, 13 pages
doi:10.1155/2011/546962

Research Article

Boosting Parallel Applications Performance on Applying DIM Technique in a Multiprocessing Environment

Mateus B. Rutzig,[1] Antonio C. S. Beck,[1] Felipe Madruga,[1] Marco A. Alves,[1] Henrique C. Freitas,[2] Nicolas Maillard,[1] Philippe O. A. Navaux,[1] and Luigi Carro[1]

[1] *Instituto de Informática, Universidade Federal do Rio Grande do Sul, 91501-970 Porto Alegre, RS, Brazil*
[2] *Instituto de Informática, Pontifícia Universidade Católica de Minas Gerais, 30535-901 Belo Horizonte, MG, Brazil*

Correspondence should be addressed to Mateus B. Rutzig, mbrutzig@inf.ufrgs.br

Received 11 August 2010; Revised 14 January 2011; Accepted 14 February 2011

Academic Editor: Aravind Dasu

Limits of instruction-level parallelism and higher transistor density sustain the increasing need for multiprocessor systems: they are rapidly taking over both general-purpose and embedded processor domains. Current multiprocessing systems are composed either of many homogeneous and simple cores or of complex superscalar, simultaneous multithread processing elements. As parallel applications are becoming increasingly present in embedded and general-purpose domains and multiprocessing systems must handle a wide range of different application classes, there is no consensus over which are the best hardware solutions to better exploit instruction-level parallelism (TLP) and thread-level parallelism (TLP) together. Therefore, in this work, we have expanded the DIM (dynamic instruction merging) technique to be used in a multiprocessing scenario, proving the need for an adaptable ILP exploitation even in TLP architectures. We have successfully coupled a dynamic reconfigurable system to an SPARC-based multiprocessor and obtained performance gains of up to 40%, even for applications that show a great level of parallelism at thread level.

1. 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. To worsen such scenario, the designers must take into account tighter design constraints as power budget and manufacturing process costs, all mixed up in the difficult task of increasing the processing capability.

Because of that, the instruction-level parallelism (ILP) exploitation strategy is no longer enough to improve the overall performance of general and embedded applications. The 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 instruction set architecture (ISA) extensions, like the employment of single instruction multiple data (SIMD) instructions, such approaches rely on long design and validation times, which goes against the need for a fast time-to-market for present day systems. On the other hand, application-specific integrated circuits (ASICs) provide high-performance and small chip area. However, such an approach attacks only a very specific application class, failing to deliver the required performance when executing applications in which behaviors were not considered at design time, being 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 stand-alone applications in single core environments [3–6]. Adaptable ILP exploitation is the major advantage of this technique, since the reconfigurable fabric can adapt 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 [7]. However, in contrast to architectures that make use of fine-grained parallelism (at instruction level) exploitation, such as the superscalar processors, the usage of the multiprocessor approach leaves all the responsibility of parallelism detection and allocation to the programmers. They must split and distribute the parallelized code among processing elements, handling all the communication issues. The 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 will run way below their full potential.

Thus, to cover all possible types of applications, the system must be conceived to provide a good performance at any parallelism level and to 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 based on the dynamic instruction merging (DIM) technique [8]. This system is capable of transparently exploring (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 the 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.

Therefore, the primary contributions of this work are

(i) to reinforce, by the use of an analytical model, the need for heterogeneous parallelism exploitation in multiprocessor environments,

(ii) to propose a multiprocessor architecture provided with an adaptable reconfigurable system (DIM technique), so it is possible to balance the best of both thread/process and ILP exploitations. This way, any kind of code, those that present high TLP and low ILP, or those that are exactly the opposite, will be accelerated.

2. Related Work

The usage of reconfigurable architectures in a multiprocessor chip is not a novel approach. In [9] the thread warping system is proposed. It is composed of an FPGA coupled to an ARM11-based multiprocessor system. Thread warping uses complex computer-aided detection (CAD) tools to detect, at execution time, critical regions of the running application and to map them to custom accelerators implemented in a simplified FPGA. A greedy knapsack heuristic is used to find the best possible allocation of the custom accelerators onto the FPGA, considering the possibility of partial reconfiguration. In this system, one processor is totally dedicated to run the operating system tasks needed to synchronize threads and to schedule their kernels to be executed in the accelerators. However, this processor may become overloaded if several threads are running on tens or hundreds of processors, affecting system scalability. Another drawback is that, due to the high time overhead imposed by the CAD and greedy knapsack algorithms, only critical code regions are optimized. Consequently, only applications with few and very defined kernels (e.g., filters and image processing algorithms) are accelerated, narrowing the field of application of this approach.

In [10], the Annabelle SoC is presented. It comprises an ARM core and four domain-specific coarse-grain reconfigurable architectures, named Montium cores. Each Montium core is composed of five 16-bit arithmetic and logic units (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 possibility of heterogeneous parallelism exploitation in a multiprocessor environment, this work focuses only on speeding up DSP applications (e.g., FFT, FIR, and SISO algorithms).

In [11], the authors propose the employment of a shared reconfigurable logic, claiming that area and energy overhead are barriers when reconfigurable fabric is used as a private accelerator for each processing element of a multiprocessor design. Results of area and power reduction are demonstrated when sharing temporally and spatially the reconfigurable fabric. However, such approach relies on compiler support, precluding binary compatibility and affecting time-to-market due to larger design times.

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

(i) unlike [9, 10], provides lower reconfiguration time, thus allowing ILP investigation/acceleration of the entire application code, including highly thread-parallel algorithms,

(ii) unlike [11], maintains binary compatibility through the application of a lightweight dynamic detection hardware that, at run-time, recognizes parts of code to be executed on the reconfigurable data path.

3. Analytical Model

In this section, we try to define the design space for multiprocessor-based architectures. First, we model a multiprocessing architecture (*MP—multiprocessor*) composed of many simple and homogeneous cores to elucidate the advantages of thread-level parallelism and compare its execution time (ET) to the modeling of a high-end single processor (*SHE—single high-end*) model with a great instruction-level parallelism exploration capability.

In the software point of view, we have used the amount of fine- (instruction) and coarse- (thread) level parallelism available in the application to investigate the performance potentials of both architectures. Considering a portion of code of a certain application, these software characteristics are denoted as

(i) α—can be executed in parallel in a single core,

(ii) β—cannot be executed in parallel in a single core,

(iii) δ—can be split among the cores of the multiprocessor environment,

(iv) γ—cannot be split among the cores of the multiprocessor environment.

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

$$\text{ET} = \text{Instructions} * \text{CPI} * \text{CycleTime}, \tag{1}$$

where CPI is the mean number of cycles necessary to execute an instruction and Cycletime the operating frequency of the processor.

In this model, no information about cache accesses is considered, nor the performance of the disk or I/O is taken into account. Nevertheless, although simple, this model can provide interesting clues on the potential of multiprocessing architectures for a wide range of applications classes.

3.1. Low-End Single Processor. Based on (1), for a low-end single (*SLE—single low-end*) processor, the execution time can be written as

$$\text{ET}_{\text{SLE}} = \text{Instructions}\left(\propto \text{CPI}_{\text{SLE}} + \beta\text{CPI}_{\text{SLE}}\right)\text{CycleTime}_{\text{SLE}}. \tag{2}$$

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

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

$$\text{ET}_{\text{SHE}} = \text{Instructions}\left(\propto \text{CPI}_{\text{SHE}} + \beta\text{CPI}_{\text{SLE}}\right)\text{CycleTime}_{\text{SHE}}, \tag{3}$$

CPI_{SHE}, that also could be written as $\propto \text{CPI}_{\text{SLE}}/\text{issue}$ (i.e., a high-end single processor would have the same CPI as the CPI of a low-end processor divided by the mean number of instructions issued per cycle), is usually smaller than 1, because a high-end single processor can exploit high levels of ILP, thanks to replication of functional units, branch prediction, speculative execution and mechanisms to handle false data dependencies, and so on. A typical value of CPI_{SHE} for a current high-end single processor is 0.62 [12], showing that more than one instruction can be executed per cycle. Thus, based on (3) one gets

$$\text{ET}_{\text{SHE}} = \text{Instructions}\left(\frac{\propto \text{CPI}_{\text{SLE}}}{\text{issue}} + \beta\text{CPI}_{\text{SLE}}\right)\text{CycleTime}_{\text{SHE}}. \tag{4}$$

Issue represents the maximum number of instructions that can be issued in parallel to the functional units, considering the best-case situation: there are no data or control dependencies in the code. As already explained, coefficients α and β refer to the percentage of instructions that can be executed in parallel or not (this way, $\alpha + \beta = 1$), respectively. Finally, CycleTime_{SHE} represents the clock cycle time of the high-end single processor.

3.3. Homogeneous Multiprocessor Chip. Having stated the equation to calculate the performance of the high-end and low-end single processor, now the potential use of a homogeneous multiprocessing architecture, built by the replication of low-end processors, is studied. Such architecture does not heavily exploit the available ILP, but mostly the thread-level parallelism (TLP). Some works [13] propose an automatic translation of code with enough ILP into TLP, so that more than one core will execute the code. A multiprocessor environment is usually composed of low-end processor units, so that a large number of them can be integrated within the same die. Considering that each application has a certain number of instructions that can be split into several processors, one could write the following equation, based on (1) and (2):

$$\begin{aligned}\text{ET}_{\text{MP}} &= \text{Instructions}\left(\frac{\delta}{P} + \gamma\right)\\ &\quad\times\left(\alpha\text{CPI}_{\text{SLE}} + \beta\text{CPI}_{\text{SLE}}\right)\text{CycleTime}_{\text{MP}},\end{aligned} \tag{5}$$

where δ is the amount of code that can be transformed into multithreaded code, while γ is the part of the code that must be executed sequentially (no TLP is available). P is the number of low-end processors that is available in the chip. Hence, the second term of (5) reflects the fact that in a multiprocessor environment one could benefit from thread-level parallelism, since increasing the number of processors will only accelerate parts of the code that can be parallelized at thread level.

3.4. High-End Single Processor versus Homogeneous Multiprocessor Chip. Based on the above reasoning, one can compare the performance of the high-end single processor to the multiprocessor environment. However, one important aspect is that the several low-end processors that compose the homogeneous multiprocessor design could also run at much higher frequencies than high-end processors, since their simple organizations reflect smaller area and power consumption. However, the total power budget will probably be the limiting performance factor for both designs. For the sake of the model, we will assume that

$$\text{CycleTime}_{\text{MP}} = K * \text{CycleTime}_{\text{SHE}}, \tag{6}$$

where K is the frequency adjustment factor to normalize the power consumption of both homogeneous multiprocessor and the high-end single processor.

Thus, the comparison of both architectures, based on (3) and (5), is given by

$$\frac{\text{ET}_{\text{SHE}}}{\text{ET}_{\text{MP}}} = \frac{\left[\text{Instructions}\left(\propto(\text{CPI}_{\text{SLE}}/\text{issue})+\beta\text{CPI}_{\text{SLE}}\right)\text{CycleTime}_{\text{SHE}}\right]}{\left[\text{Instructions}\left(\delta/P+\gamma\right)\left(\propto\text{CPI}_{\text{SLE}}+\beta\text{CPI}_{\text{SLE}}\right)\text{CycleTime}_{\text{MP}}\right]}. \tag{7}$$

By simplifying and merging (6) and (7), one gets

$$\frac{\text{ET}_{\text{SHE}}}{\text{ET}_{\text{MP}}} = \left[\frac{1}{\delta/P}+\gamma\right]\left[\frac{\propto(\text{CPI}_{\text{SLE}}/\text{issue})+\beta\text{CPI}_{\text{SLE}}}{\propto\text{CPI}_{\text{SLE}}+\beta\text{CPI}_{\text{SLE}}}\right]\left[\frac{1}{K}\right]. \tag{8}$$

From (8) one can notice that the high-end processor is faster than multiprocessor architecture when $(\text{ET}_{\text{SHE}}/\text{ET}_{\text{MP}}) < 1$. In addition, this equation shows that, although the multiprocessor architecture with low-end simple processors could have a faster cycle time (by a factor of K), that factor alone is not enough to define performance. Regarding the second term between brackets in (8), the fact that the high-end processor can execute many instructions in parallel could give a better performance. Since there is no instruction-level parallelism exploration in a low-end single processor, it means that the term $\propto \text{CPI}_{\text{SLE}}$ is always zero.

In the extreme case, let us imagine that issue $= P = \infty$, meaning that we have infinite resources, either in the form of arithmetic operators or in the form of processors. This would reduce (8) to

$$\frac{\text{ET}_{\text{SHE}\infty}}{\text{ET}_{\text{MP}\infty}} = \left[\frac{1}{\gamma}\right]\left[\frac{\beta\text{CPI}_{\text{SLE}}}{\propto\text{CPI}_{\text{SLE}}+\beta\text{CPI}_{\text{SLE}}}\right]\left[\frac{1}{K}\right]. \tag{9}$$

Equation (9) 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.

Another interesting experiment is to try to equal the performance of the high-end single core and the performance of the multiprocessor core. This way, let us consider that $T_{\text{SHE}} = T_{\text{MP}}$; hence,

$$\left[\left(\propto\text{CPI}_{\text{SHE}}+\beta\text{CPI}_{\text{SLE}}\right)\frac{1}{K}\right] = \left[\left(\frac{\delta}{P}+\gamma\right)\left(\alpha\text{CPI}_{\text{SLE}}+\beta\text{CPI}_{\text{SLE}}\right)\right]. \tag{10}$$

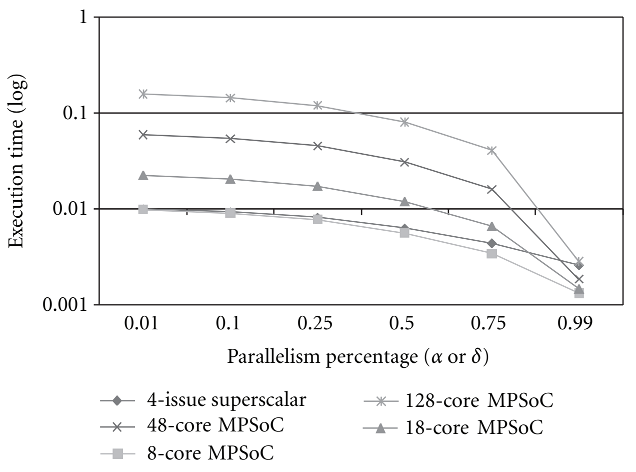

Figure 1: Multiprocessor system and superscalar performance regarding a power budget using different ILP and TLP; $\alpha = \delta$ is assumed.

From (10), one can see that one must have enough low-end processors combined to a highly parallel code (greater δ) to overcome the high-end processors advantage. This statement is clarified by the fact that the term $\propto \text{CPI}_{\text{SLE}}$ is always zero, imposing that β is equal to one and CPI_{SLE} is much higher than CPI_{SHE}.

3.5. Applying the Analytical Model in Real Processors. Given the theoretical model, one can briefly test it with some numbers based on real data. Let us consider a high-end single core: a 4-issue MIPS R10000 superscalar processor, with CPI equal to 0.6 [14] and a multiprocessor design composed of low-end MIPS R3000 processors, with CPI equal to 1.3 each [15]. A comparison between both architectures is done using the equations of the aforementioned analytical model. Figure 1 shows, in a logarithmic scale, the performance of the superscalar processor when parameters α and β change. In addition, in Figure 1 we also show the performance of the multiprocessor design, varying the δ and γ parameters and the number of processors from 8 to 128. To provide a better view of the performance considering both approaches, the x-axis of Figure 1 represents the amount of the instruction- (α) and thread- (δ) level parallelism in the application, where α is only valid for the 4-issue superscalar, while δ is valid for all the MPSoC's setups.

The goal of this comparison is to demonstrate which technique better explores its particular parallelism at different levels, considering six values for both ILP and TLP. For instance, $\delta = 0.01$ means that an application only shows 1% of thread-level parallelism within its code (valid only for the MPSoC's examples). In the same way, when $\alpha = 0.01$, it is assumed that 1% of instruction-level parallelism (ILP) is available. That is, only 1% of its instructions can be executed in parallel in the 4-issue superscalar processor. Following the same strategy found in current processor designs, for a fair performance comparison, we considered the same power budget for the high-end single core and the multiprocessor approaches. In order to normalize their power budget, we have tuned the frequency adjustment factor K of (5). For that, we fixed the 4-issue superscalar frequency to use it as the power reference, changing the K factor of the remaining approaches to achieve the same consumption as the reference. Thus, the frequency of the 8-core MPSoC must be 3 times higher than 4-issue superscalar processor. For the 18-core, such value must be a quarter higher than the reference value. Since a considerable number of cores employed in the 48-core MPSoC setup, this approach should execute 2 times slower than the 4-issue superscalar processor to achieve the same power consumption. Finally, the frequency of the 128-core MPSoC must be 5.3 times smaller than the 4-issue superscalar to respect the same power budget.

The leftmost side of Figure 1 considers any given application that has a minimum amount of instruction- ($\alpha = 0.01$) and thread- ($\delta = 0.01$) level parallelism available. In this case, the superscalar processor and the 8-core design present almost the same performance. However, considering the same power budget for all approaches by using different operating frequencies shown before, when applications show greater parallelism percentage ($\alpha > 0.25$ and $\delta > 0.25$), the 8-core design achieves better performance with TLP exploitation than the 4-issue superscalar processor with ILP exploitation.

When more cores are added in a multiprocessor design, the overall clock frequency tends to decrease, since the adjustment factor of (5) should be smaller to obey the power budget. In this way, the performance of applications that present low-thread-level parallelism (small δ) worsens when increasing the number of cores. Regarding the applications with $\delta = 0.01$ in Figure 1, performance is significantly decreased as the number of cores increases. Nevertheless, as the application thread-level parallelism increases (i.e., $\delta > 0.01$), the negative impact on performance is softened, since the additional cores will have better use.

Aiming to make a fairer performance comparison among high-end single core and multiprocessor approaches, we have devised an 18-core design composed of low-end processors that, besides presenting the same power consumption due to the power budget assumed, also has the same area of the 4-issue superscalar processor. For that, we considered that the MIPS R3000 takes only 75.000 transistors [16], almost 29 times less than the 2.2 millions of transistors spent on the MIPS R10000 design [17]. Furthermore, for a reasonable comparison, we also considered that the intercommunication mechanism would take nearly 37% of the chip area, as reported in [18]. The performance of both approaches shows the powerful capabilities of the superscalar processor. Regarding the same area and power for both designs, as shown in Figure 1, the multiprocessor approach (18-core MPSoC) only surpasses the superscalar (4-issue superscalar) performance when the TLP level is greater than 85% ($\delta > 0.85$).

Summarizing the comparison with the same power budget, the superscalar machine shows better performance over applications with low-thread-level parallelism. On the other hand, there is an additional tradeoff that must be considered regarding multiprocessor designs, since, when more cores are included in the chip, the multiprocessor performance tends to worsen, since the operating frequency must be decreased to respect the power budget limits. When almost the whole application presents high TLP ($\delta > 0.99$), the 128-core design takes longer execution time than the other multiprocessor designs since its operating frequency is very low.

Considering real applications, thread-level parallelism exploitation is widespread employed to accelerate most multimedia and DSP applications thanks to their data independent iteration loops. However, even applications with high TLP could still obtain some performance improvement by also exploiting ILP. Hence, in a multiprocessor design, ILP techniques also should be investigated to conclude what is the best fit concerning the design requirements. Hence, the analytical model indicates that heterogeneous multiprocessor system is necessary 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 named DIM technique.

4. Reconfigurable Multiprocessing System

Section 3 demonstrated that in a heterogeneous application environment, TLP and ILP exploitation are complementary. This way, it is necessary to explore different grains of parallelism to balance performance. Aiming to support this statement, we have built a multiprocessor structure shown in Figure 2(a) to reproduce the analytical model shown in Section 3 by executing well-known applications. The architecture in the example is composed of four cores, so TLP exploitation is guaranteed. However, as ILP exploitation is also mandatory, we have coupled a coarse-grain reconfigurable data path to each one of the cores, since the use of reconfigurable fabric has already shown great speedups with low-energy consumption [6, 8] concerning single thread applications.

Figure 2(b) shows in detail the microarchitecture of the processor, named as reconfigurable core (RC), used as the base processing element of the reconfigurable multiprocessing system. To better explain the RC processor, we divided such architecture in 4 blocks. Block 1 depicts the reconfigurable data path that aggregates the input context, output context, and the functional units. Block 2 presents the basic SparcV8 like five-stage pipelined processor. Block 3 illustrates the pipeline stages of the dynamic instruction merging (DIM) [8] technique that works in parallel to the processor pipeline. It is responsible for transforming instruction blocks into configurations of the reconfigurable data

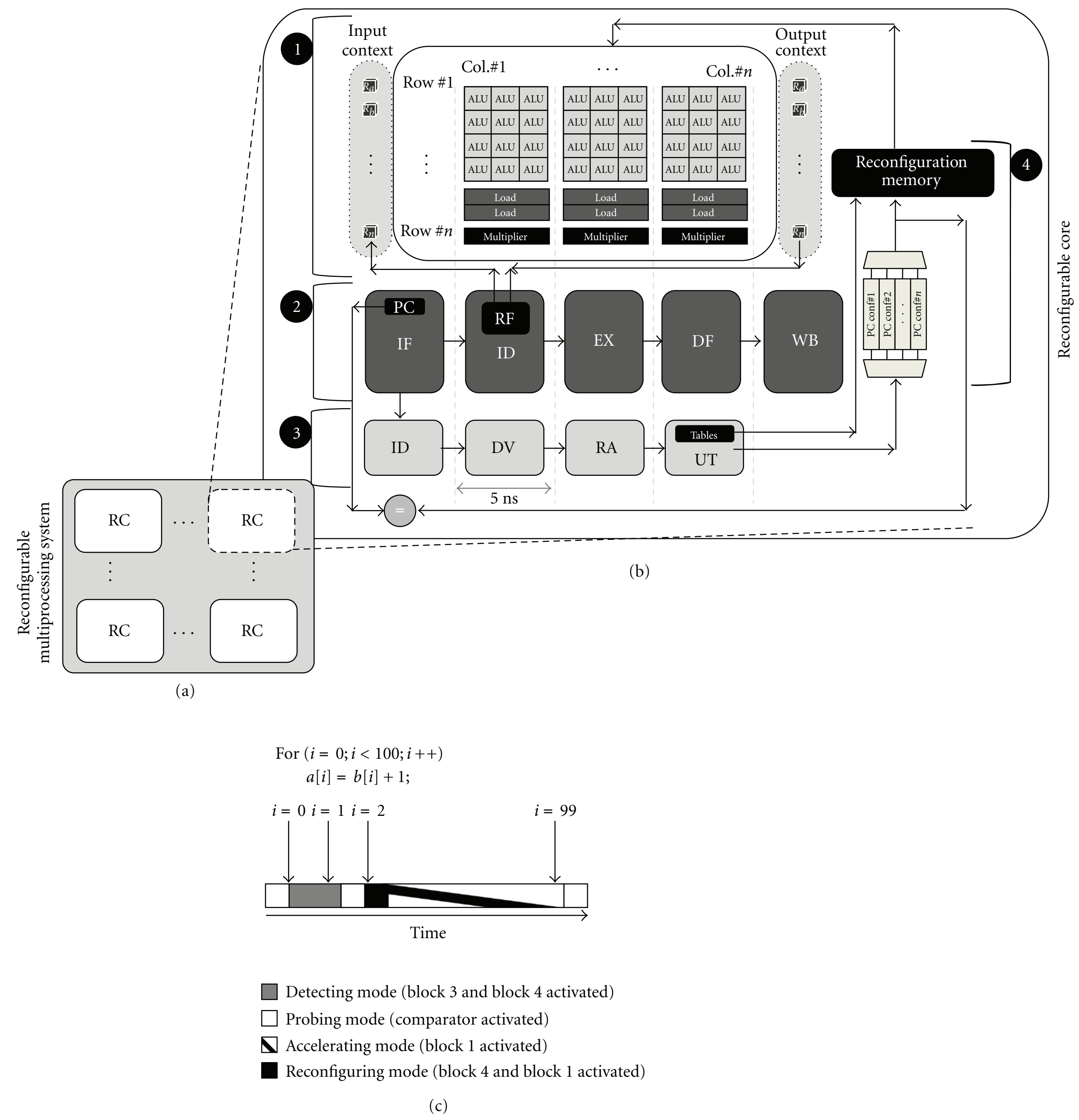
Input context
Output context
Col.#1
Col.#n
Row #1
Row #n
ALU
Load
Multiplier
Reconfiguration memory
PC
RF
IF
ID
EX
DF
WB
PC conf#1
PC conf#2
PC conf#n
ID
DV
RA
Tables
UT
5 ns
=
1
2
3
4
Reconfigurable core
RC
Reconfigurable multiprocessing system
(a)
(b)
For (i = 0; i < 100; i++)
$a[i] = b[i] + 1$;
i = 0 i = 1 i = 2 i = 99
Time
Detecting mode (block 3 and block 4 activated)
Probing mode (comparator activated)
Accelerating mode (block 1 activated)
Reconfiguring mode (block 4 and block 1 activated)
(c)

Figure 2: (a) Multiprocessing system. (b) Reconfigurable core. (c) Example of a loop optimization.

path at run time. Block 4 demonstrates the reconfiguration memory and the address cache. The reconfiguration memory holds the configuration bits previously generated by the DIM, so next time when the same translated sequence is found, the configuration bits are reused. The address cache (4-way associative) is responsible for keeping the first PC address of each translated sequence. More details about these components will be presented in the next sections.

Figure 2(c) shows an example of how a loop would be accelerated using the proposed process. The reconfigurable core works in four modes: probing, detecting, reconfiguring, and accelerating. At the beginning of the time bar shown in Figure 2(c), the RC is searching for an already translated configuration to accelerate through execution in the reconfigurable data path.

However, when the first loop iteration appears ($i = 0$), the DIM detects that there is a new code to translate and it changes to detecting mode. In that mode, while the instructions are executed in the processor pipeline, they are also translated to a configuration by the DIM. When the second loop iteration is found ($i = 1$), the DIM is still finishing building the current configuration (that started when $i = 0$) and storing it into the reconfiguration memory.

Then, when the first instruction of the third loop iteration comes to the fetch stage of the processor pipeline ($i = 2$), the probing mode detects a valid configuration in the reconfiguration memory, since the previously started detection process is now finished and the memory address of the first instruction of the translated sequence was found in the address cache.

Therefore, the RC enters in reconfiguring mode to feed the reconfigurable data path with the operands and the reconfiguration bits. Finally, the accelerating mode is activated and the next loop iterations (until the 99th) are efficiently executed, taking advantage of the reconfigurable logic.

4.1. Reconfigurable Data Path Structure (Block 1). Following the classifications shown in [19, 20], 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 each configuration, since the basic processing elements are functional units that work at the word level (arithmetic and logic, memory access, and multiplier). The data path is organized as a matrix of rows and columns, composed of functional units. Three columns of arithmetic and logic units (ALUs) compose a level. A level does not affect the SparcV8 critical path (which, in this case, it is given by the register file). 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 example of the data path shown in Figure 2(b) could execute up to four arithmetic and logic operations, two memory accesses (two memory ports are available), and one multiplication in parallel. The number of rows, in turn, determines the maximum number of dependent instructions placed into one configuration. Both the number of rows and the number of parallel components can be modified according to the application requirements and the design constraints. 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, 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 purely 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 keep the input context, which is connected to the processor register file. The fetching of the operands 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. The values stored in the output context are sent to the processor register file on demand. It means that if a given result is produced at any level and it will not be changed in the subsequent levels, its value is written back at the same level that it was produced. In the current implementation, the reconfigurable system provides two write-backs per level.

We have coupled sleep transistors [18] to switch power on/off of each functional unit in the reconfigurable data path. The dynamic reconfiguration process is responsible for the sleep transistors management. Their states are stored in the reconfiguration memory, together with the reconfiguration data. Thus, for a certain configuration, idle functional units are set to the off state, avoiding leakage or dynamic power dissipation, since the incoming bits do not produce switching activity in the disconnected circuit. Although the sleep transistors are bigger and in series to the regular transistors used in the implemented circuit, they have been designed so that their delays do not significantly impact the critical path or the reconfiguration time.

4.2. Processor Pipeline (Block 2). A SPARC-based architecture is used as the baseline processor to work together with the reconfigurable system. Its five-stage pipeline reflects a traditional RISC architecture (instruction fetch, decode, execution, data fetch, and write-back). The microarchitecture and the performance of such processor are very similar to the MIPS R3000, considered as the low-end processor in the analytical model shown in Section 3.

4.3. Dynamic Instruction Merging (Block 3). As explained before, the dynamic instruction merging (DIM) can work in four modes: detecting, probing, accelerating, and reconfiguring. As can be observed in Figure 2(b), the hardware responsible for the detecting mode contains four pipeline stages.

(i) Instruction decode (ID): the instruction is broken into operation, source operands, and target operand.

(ii) 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 column that the current instruction should be placed.

(iii) Resource allocation (RA): in this stage, the data dependence is already solved and the correct data path column is known. Hence, the RA stage is responsible for verifying the resources availability in that column, linking the instruction operation to the correct type of functional unit. If there is no functional unit available at this column, the next column at the right side will be checked. This process is repeated until finding a free functional unit.

(iv) Update tables (UT): this stage configures the routing to feed that functional unit with the correct source operands from the input context and to write the result in the correct register of the output context. After that, the bitmaps and tables are updated and

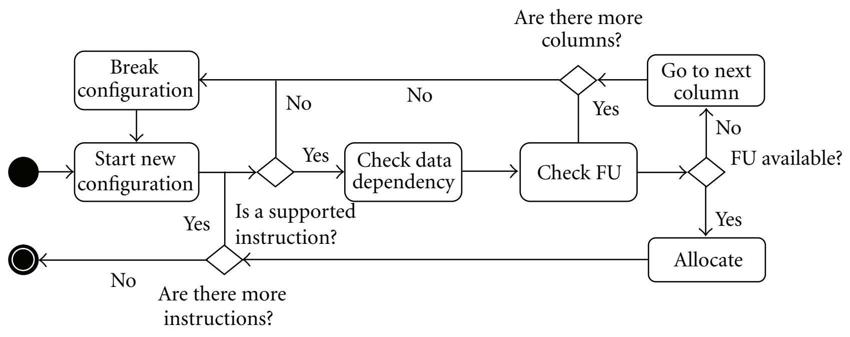

Figure 3: DIM activity diagram.

the configuration is finished: their configuration bits are sent to the reconfiguration memory and the address cache is updated.

Figure 3 illustrates, by an activity diagram, the whole DIM process to create a configuration. The first step is the execution support verification. If there is no compatible functional unit to execute such an operation (e.g., division), the configuration is finished and the next instruction is a candidate to start a new configuration. On the other hand, if there is support, the data dependency among previously allocated instructions is verified (DV stage) and the correct functional unit within that column is defined. Then, the current configuration is sent to the reconfiguration memory.

4.4. Storage Components (Block 4). Two storage components are part of the reconfigurable system: address cache and reconfiguration memory. The configurations are indexed by the address of the first instruction of the translated sequence and kept in the address cache, a 128-entry 4-way associative cache. The Address Cache is only accessed when the DIM is working in the probing mode. An address cache hit indicates that a configuration was found, changing the system to the reconfiguring mode. In this mode, using the pointer given by the address cache, the reconfiguration memory is accessed to feed the data path routing. The reconfiguration memory stores the routing bits and the necessary information (such as the input and output contexts and immediate values) to fire a configuration. Finally, the DIM hardware changes to accelerating mode, beginning the execution process in the reconfigurable data path.

5. Simulation Environment

5.1. Workload. A workload of only high parallel applications with distinct behaviors was chosen, using benchmarks from the well-known SPLASH2 [21] and PARSEC [22] suites. In addition, two numerical applications written in OpenMP were used [23].

The list below briefly describes each of them.

(i) *FFT* [21]. It is a complex 1D version of a six-step FFT algorithm.

(ii) *LU* [21]. It factors a dense matrix in the equivalent lower-upper matrix multiplication.

(iii) *Blackscholes* [22]. It solves the partial differential equation of *Blackscholes* in order to compute prices for a portfolio of European options.

(iv) *Swaptions* [22]. Monte Carlo simulation is used to price a portfolio of *swaptions*.

(v) *Molecular Dynamics* (*MD*) [4]. It implements the velocity Verlet algorithm for molecular dynamics simulation.

(vi) *Jacobi* [23]. It utilizes the Jacobi iterative method to solve a finite difference discretization of Helmholtz.

5.2. TLP and ILP Exploration Opportunities. In this section we show the opportunities for coarse- and fine-grain parallelism exploration in the selected benchmarks shown in Section 5.1. The experiments addressing these applications were done in a SparcV8 architecture varying the number of threads from 1 to 64.

The mean size of the basic blocks (BB) of an application is an important aspect to define its fine grain parallelism level since the room for most ILP exploration techniques relies on this characteristic. The second column of Table 1 presents the mean BBs size of the selected applications. As it can be noticed, even parallel applications provide great room for instruction-level parallelism exploration. The remaining columns in Table 1 show, in percentage, the load balancing between threads of the selected applications. As expected, most applications provide a perfect load balancing up to 64 threads. *FFT* and *LU* do not follow the trend of the other applications, since the load balancing decreases as the number of threads increases.

Therefore, even applications with perfect load balancing (e.g., *swaptions*) provide a great room for instruction-level parallelism since their basic blocks have enough instructions to be parallelized. In the same way, applications with poor load balancing, where probably thread-level parallelism exploration will not be enough to achieve satisfactory performance improvements (e.g., *lu* with a great number of threads), can benefit even more from instruction-level parallelism exploitation. In this way, one can conclude that

Table 1: ILP and TLP opportunities for the selected benchmarks.

Benchmark	Mean BB size (#instr)	Load balancing (%)			
		4 threads	8 threads	16 threads	64 threads
Swaptions	5.92	99.00	99.00	99.00	98.00
Blackscholes	4.83	99.00	99.00	99.00	98.00
MD	6.51	95.04	83.24	88.92	89.87
Jacobi	6.94	97.02	97.02	92.07	93.12
FFT	8.10	69.39	49.50	31.96	24.58
LU	8.32	81.20	56.77	29.35	7.03

Table 2: 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

Table 3: Average speedup on different number of cores.

	4 cores	8 cores	16 cores	64 cores
TLP	3.74	6.86	12.47	44.3
ILP + TLP	6.46	11.85	21.71	51

the mixed parallelism exploitation is mandatory even for applications where the thread-level parallelism is dominant.

5.3. Methodology. To simulate the reconfigurable multiprocessor system, we have used the scheme presented in [24]. It consists of a functional full system [25] that models the SparcV8 architecture and cycle accurate timing simulators [26] that reproduce the behavior of the individual reconfigurable cores depicted in Figure 2(b). Since the applications are split automatically by the OpenMP and the Posix Threads API, the cycle accurate simulator gives special attention to synchronization mechanisms, such as locks and barriers. Therefore, the elapsed time regarding blocking synchronization and memory transfers are precisely measured.

For all experiments, we have tuned the number of reconfigurable cores based on the number of threads used to run the applications presented in Section 5.1.

To demonstrate the impact of ILP exploitation in the performance, we have used two different configurations for the reconfigurable data path (block 1 of Figure 2(b)), changing its number of basic functional units. The setups, shown in the Table 2, have already presented the best tradeoff considering area and performance executing single-thread applications [8].

6. Results

6.1. Performance Results. This section demonstrates the performance evaluation of the reconfigurable multiprocessing system over three different aspects:

(i) TLP exploitation by changing the number of cores from four up to 64 (in these experiments, stand-alone SPARC cores are used: they are not coupled to the reconfigurable architecture);

(ii) 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);

(iii) the influence of changing the applications' data set sizes on performance.

Figure 4 explores the first two aspects discussed above: TLP exploitation only, varying the number of stand-alone SPARC cores (solid bars) and TLP + ILP exploitation, by coupling the reconfigurable architecture (RC#1) to each one of the cores (striped bars). Regarding the former, performance scales linearly as the number of cores increases. *FFT* and *LU* do not follow this behavior, since their codes, as shown in Table 1, do not present perfect load balancing as other applications.

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 for finer-grain parallelism exploitation to complement the TLP gains. Table 3 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 the eight standalone cores exploiting only TLP parallelism. The same occurs when comparing a system with 8 cores and the RC#1 setup to 16 stand-alone cores.

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 loss when using the RC#2

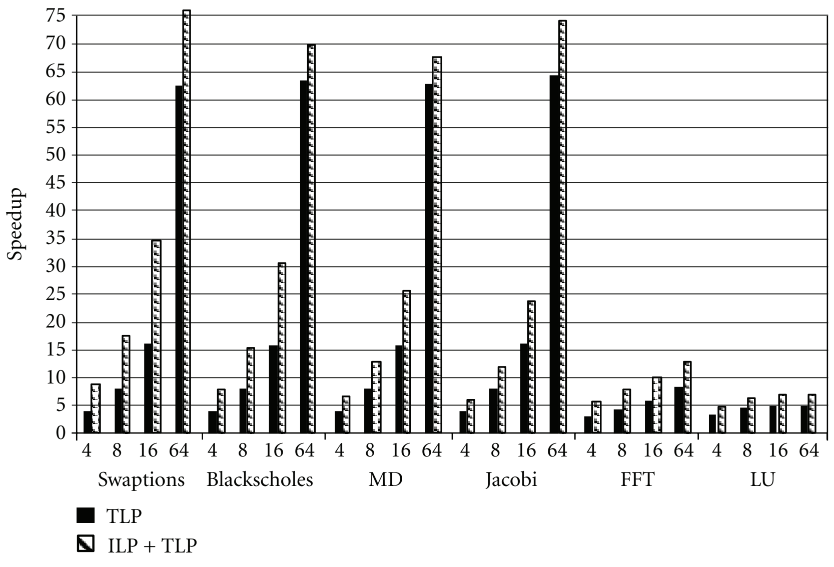

Figure 4: Performance of TLP and ILP + TLP exploration using RC#1 setup.

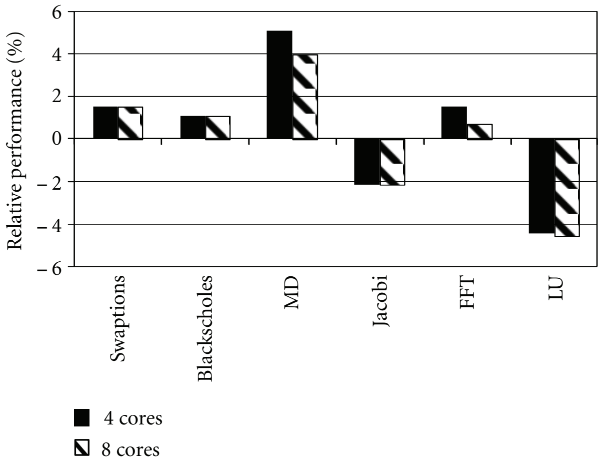

Figure 5: Performance comparison among RC#1 and RC#2 setups.

setup. It happens because the DIM produces a different amount of configurations for both setups. Due to the dynamic behavior of DIM technique, it affects the address cache storage producing more configuration misses in the RC#2 setup. Therefore, since RC#2 does not show any significant advantage over the RC#1, the RC#1 will be used for the remaining experiments.

Figure 6 shows the performance evaluation when running the same application workload with two different data set sizes, so it is possible to demonstrate that changing the data set size does not affect the performance results shown in Figure 4.

However, *FFT* and *LU* present a significant impact when changing the data set size. Figure 7 shows this data in more detail. *FFT* has a significant amount of sequential code responsible for data initialization. Thus, when we increase the data set size, the initialization becomes more significant over the whole application execution time. This behavior is more evident in the multiprocessing system composed of 16 cores.

Regarding *LU*, the larger data set size provides a perfect load balance with a great number of processors [21], as observed in Figure 7. On the other hand, smaller data set sizes increase the imbalance by splitting less blocks per processor in each step of the factorization.

6.2. Energy Results Considering the Same Power Budget. In Section 3.4, we have created a power budget to clarify the advantages/disadvantages of instruction- and thread-level parallelism exploitation. This way, we have also evaluated the energy consumption of the selected benchmarks considering the same power budget for both parallelism exploitations (ILP and TLP+ILP). The power dissipation of the stand-alone SparcV8 is 385.14 mWatts and the reconfigurable core consumes 699.33 mWatts. Therefore, we have compared the 8-core SparcV8 with the reconfigurable multiprocessing system composed of 4 reconfigurable cores, since both reach nearly 3 Watts of power dissipation. In addition, these setups will help us to measure the contribution of the proposed approach in reducing energy consumption, since, as shown in Table 3, both setups provide almost the same performance. Due to the high simulation time, we choose three benchmarks from the application workload to show the energy consumption of both approaches. This application subset contains massive thread-level parallelism applications (MD and Jacobi) as well as application that shows considerable load unbalancing (LU).

Despite the fact that all applications provide massive thread-level parallelism, since their performance scales linearly as the number of cores increases, the proposed approach consumes less Power than the multiprocessing system composed of stand-alone SparcV8 in all benchmarks evaluated. Energy savings are possible because, although the power consumption of reconfigurable core is the same as

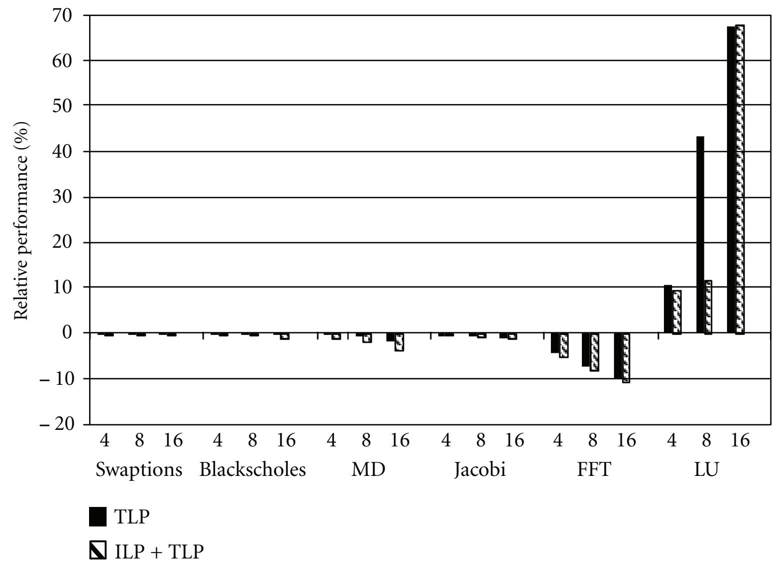

Figure 6: Performance comparison regarding different application data set sizes.

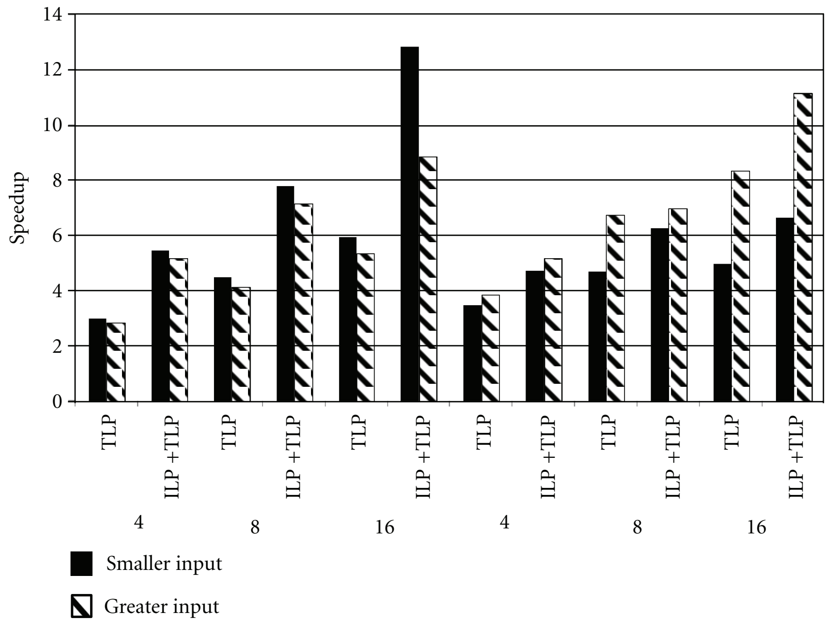

Figure 7: FFT and LU performance regarding different application data set sizes.

the power presented in the multiprocessing system composed of stand-alone SparcV8, the average power is lower, mainly thanks to the use of sleep transistors to turn off idle functional units of the reconfigurable data path. In addition, although more power is spent because of the DIM hardware and reconfigurable data path, total average power is reduced since there are fewer memory accesses for instructions: once they were translated to a data path configuration, they will reside in the reconfiguration memory.

Disregarding the power budget proposed in this section, we can compare the energy consumption of the 8-core SparcV8 with the multiprocessing system composed of eight reconfigurable cores. As can be seen in Table 4, the proposed approach outperforms the 8-core SparcV8, on average, by 72% and still consumes 42% less energy. The main source, besides the already mentioned, is the shorter execution time of the mixed parallelism exploration.

7. Conclusions

This paper demonstrated that, although the instruction-level parallelism (ILP) exploitation is reaching its limits and multiprocessing system appears as a solution to accelerate applications by exploring coarse grains of parallelism, there are significant sequential parts of code that still must be handled by ILP exploitation mechanisms. Therefore,

Table 4: Energy consumption, in mJ, of both multiprocessing system and reconfigurable system on running MD, Jacobi and LU.

	TLP				ILP + TLP			
	4 cores	8 cores	16 cores	64 cores	4 cores	8 cores	16 cores	64 cores
MD	0.282	0.329	0.353	0.376	0.185	0.216	0.232	0.248
Jacobi	99.1	115.7	124.3	132.3	69.6	81.3	87.3	93.1
LU	0.167	0.194	0.232	0.330	0.113	0.132	0.157	0.227
Average	33.2	38.8	41.6	44.3	23.3	27.2	29.2	31.2

there is the need of mixed-grain parallelism exploitation to achieve balanced performance improvements even for applications that present dominant thread-level parallelism. This paper presented such system: an adaptable ILP exploitation mechanism, using reconfigurable logic, coupled to a multiprocessing environment.

References

[1] D. W. Wall, "Limits of instruction-level parallelism," *ACM SIGPLAN Notices*, vol. 26, no. 4, pp. 176–188, 1991.

[2] J. Mak and A. Mycroft, "Limits of instruction data dependence graphs," in *Proceedings of the 7th International Workshop on Dynamic Analysis (WODA '09)*, Chicago, Ill, USA, July 2009.

[3] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—a self-reconfigurable data processing architecture," *Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[4] S. J. Patel and S. S. Lumetta, "rePLay: a hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 590–608, 2001.

[5] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pp. 12–21, IEEE Computer Society, Napa Valley, Calif, USA, 1997.

[6] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," in *Proceedings of the 41st Annual Conference on Design Automation (DAC '04)*, pp. 659–681, ACM, New York, NY, USA, 2004.

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

[8] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 1208–1213, March 2008.

[9] G. Stitt and F. Vahid, "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 (CODES+ISSS '07)*, ACM, Salzburg, Austria, September-October 2007.

[10] G. J. Smit, A. B. Kokkeler, P. T. Wolkotte, and M. D. van de Burgwal, "Multi-core architectures and streaming applications," in *Proceedings of the International Workshop on System Level Interconnect Prediction (SLIP '08)*, pp. 35–42, ACM, Newcastle, UK, April 2008.

[11] M. A. Watkins, M. J. Cianchetti, and D. H. Albonesi, "Shared reconfigurable architectures for CMPS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 299–304, September 2008.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE International Workshop Workload Characterization (WWC '01)*, pp. 3–14, Washington, DC, USA, December 2001.

[13] Y. Song, S. Kalogeropulos, and P. Tirumalai, "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 (PACT '05)*, pp. 99–109, IEEE Computer Society, Washington, DC, USA, September 2005.

[14] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance analysis using the MIPS R10000 performance counters," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, p. 16, IEEE Computer Society, Pittsburgh, Pa, USA, January 1996.

[15] T. Roirdan, G. P. Grewal, S. Hsu et al., "System design using the MIPS R3000/3010 RISC chipset," in *Proceedings of the 34th IEEE Computer Society International Conference on Intellectual Leverage, Digest of Papers (COMPCON '89)*, pp. 494–498, San Francisco, Calif, USA, 1989.

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

[17] K. C. Yeager, "The Mips R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, 1996.

[18] http://blogs.intel.com/research/2007/07/inside_the_terascale_many_core.php.

[19] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[20] A. C. Beck and L. Carro, *Dynamic Reconfigurable Architectures and Transparent Optimization Techniques*, Springer, New York, NY, USA, 2009.

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

[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 72–81, ACM, Toronto, Canada, October 2008.

[23] A. J. Dorta, C. Rodriguez, F. D. Sande, and A. Gonzalez-Escribano, "The OpenMP source code repository," in *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '05)*, pp. 244–250, IEEE Computer Society, Washington, DC, USA, February 2005.

[24] M. Monchiero, J. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," *ACM SIGARCH Computer Architecture*, vol. 37, no. 2, pp. 10–19, 2009.

[25] P. S. Magnusson, M. Christensson, J. Eskilson et al., “Simics: a full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 12–58, 2002.

[26] M. B. Rutzig, A. C. Beck, and L. Carro, “Dynamically adapted low power ASIPs,” in *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, J. Becker, R. Woods, P. Athanas, and F. Morgan, Eds., vol. 5453 of *Lecture Notes In Computer Science*, pp. 110–122, Springer, Karlsruhe, Germany, March 2009.