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

ANTONIO CARLOS SCHNEIDER BECK FILHO

Transparent Reconfigurable Architecture for Heterogeneous Applications

Thesis presented in partial
fulfillment of the requirements
for the degree of Doctor
of Computer Science

Prof. Dr. Luigi Carro
Advisor

Porto Alegre, June 2008

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Beck Filho, Antonio Carlos Schneider

Transparent Reconfigurable Architecture for Heterogeneous Applications / Antonio Carlos Schneider Beck Filho. – Porto Alegre: PPGC da UFRGS, 2008.

188 f.: il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2008. Advisor: Carro, Luigi.

1. Microeletrônica. 2. Arquiteturas. 3.Reconfiguráveis. I. Carro, Luigi. II. Título.

To Sabrina and my parents.

ACKNOWLEDGMENTS

First, I would like to give special thanks to my dear Sabrina and to my parents.

I would also like to express my gratitude to the following friends and colleagues: Julius Mattos, Mateus Beck, Émerson Barbieri, Fernando Cortes, Leomar Rosa, Marcio Oyamada, Emilena Specht, Alexandre Gervini, Marco Wehrmeister, Rodrigo Araújo, Daniel Biasoli, André Brandão, Cassio Ehlers, Eduardo Funari, Marcos Pont, Rodrigo Motta, Dalton Colombo, Renato Hentschke, Lisane Brisolara, Victor Gomes, Renato Hubner, Luis Otávio Soares, Arnaldo Azevedo, Ivan Garcia and all the people in the lab.

My sincere thanks to Professors Flávio Wagner, Altamiro Susin, Sérgio Bampi and Ricardo Reis; to my advisor in The Netherlands, Professor Georgi Gaydadjiev and to my advisor here in Brazil, Professor Luigi Carro.

Finally, I thank the Federal University of Rio Grande do Sul and the research support agencies, CAPES and CNpQ.

The long version, in Portuguese:

Obrigado...

À Sabrina Moraes. Por ter sido trocada várias vezes pelo trabalho, pelo computador, por *papers* e *deadlines*. Novamente.

Aos meus pais, Antonio e Léia Beck, pelo constante incentivo e por entenderem a minha ausência.

Ao Julius Mattos, pelo enorme força que me deu quando cheguei a Porto Alegre. Sempre presente e disposto a ajudar.

Ao Mateus Beck, primo e parceria de *rejecteds*.

Ao Émerson Barbieri, grande amigo e colorado.

Aos amigos que vêm desde o tempo do mestrado: Fernando Cortes, Leomar Rosa, Marcio Oyamada, Emilena Specht, Alexandre Gervini, Marco Wehrmeister e Renato Hentschke.

E aos amigos que vêm desde a graduação: Rodrigo Araújo, Daniel "Jacaré" Biasoli, André Brandão e Cássio Ehlers.

Pro pessoal do Gigante da Beira-Rio: Émerson, Eduardo Funari e Marcos Pont. Neste período, vimos o Sport Club Internacional ser campeão de tudo: Libertadores, Mundo, Recopa, Sul Americana e muito mais. E, de quebra, goleadas nos rivais históricos.

À gurizada parceira de palas em Porto Alegre: Fernando, Rodrigo “Buka” Motta, Dalton “Paulista” Colombo e Mateus “Amagura”.

Ao *MP3 Player Team*, que fez o Femtojava em FPGA tocar 3 segundos de MP3 em *loop* infinito: Victor Gomes, Renato Hubner e Julius.

Pro Arnaldo Azevedo, pro Ivan Garcia, e pra turma de portugueses. Amigos que deixei no frio da Holanda.

À turma do futebol da informática, liderada pelo Luis Otávio Soares.

Pro pessoal do nosso laboratório, 67-213, e do laboratório ao lado, 67-211.

Ao grupo de Microeletrônica, em especial ao LSE (Laboratório de Sistemas Embarcados).

Aos professores Flávio Wagner, Altamiro Susin, Sérgio Bampi e Ricardo Reis.

Ao Professor Georgi Gaydadjiev, meu orientador durante o Sanduíche na Holanda. E também para a TUDelft, Universidade que frequentei.

Ao Professor Luigi Carro, por me orientar novamente, passar um pouco do ímpeto em querer conquistar o mundo e lembrar dos *deadlines* dos congressos.

À Universidade Federal do Rio Grande do Sul e funcionários, pelo ensino de muita qualidade. Também, pra CAPES e pro CNpQ.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic and Logic Unit
AMD	Advanced Micro Devices
AMIL	Number of Merged Instructions
ASIC	Application Specific Integrated Circuits
ASIP	Application Specific Instruction set Processor
BT	Binary Translation
CAD	Computer Aided Design
CCA	Configurable Compute Array
CDFG	Control Data Flow Graph
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chip Multiprocessor
CORDIC	Coordinate Rotation Digital Computer
CPII	Cycles per Issue Interval
DCT	Discrete Cosine Transform
DIM	Dynamic Instruction Merging
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
IMDCT	Inverse Modified Discrete Cosine Transform
ILP	Instruction Level Parallelism
IPC	Instructions per Cycle
IPII	Instructions per Cycle Interval
ITRS	International Technological Roadmap for Semiconductors
JVM	Java Virtual Machine
JIT	Just-in-time
LUT	Look-Up Table
MAC	Multiply and Accumulate

NRE	Non-Recurring Engineering
PE	Processing Element
RAM	Random Access Memory
RFU	Reconfigurable Functional Unit
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SIMD	Single Instruction – Multiple Data
SSE	Streaming SIMD Extensions
VLIW	Very Long Instruction Word
VMM	Virtual Machine Monitor
MMX	Multimedia Extension
MP3	MPEG-1 Audio Layer 3
OPI	Operations Per Instruction
OS	Operating System
RISP	Reconfigurable Instruction Set Processor
RFU	Reconfigurable Functional Unit
RPU	Reconfigurable Processor Unit
SMT	Simultaneous Multithreading
TOS	Top Of Stack
RTM	Reuse Trace Memory
WAR	Write After Read
WAW	Write After Write

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

As Moore's law is losing steam, one already sees the phenomenon of clock frequency reduction caused by the excessive power dissipation in general purpose processors. At the same time, embedded systems are getting more heterogeneous, characterized by a high diversity of computational models coexisting in a single device. Therefore, as innovative technologies that will completely or partially replace silicon are arising, new architectural alternatives are necessary. Although reconfigurable computing has already shown to be a potential solution for such problems, significant speedups are achieved just in very specific dataflow oriented software, not representing the reality of nowadays systems. Moreover, its wide spread use is still withheld by the need of special tools and compilers, which clearly preclude software portability and reuse of legacy code. Based on all these facts, this thesis presents a new technique using reconfigurable systems to optimize both control and dataflow oriented software without the need of any modification in the source or binary codes. For that, a Binary Translation algorithm has been developed, which works in parallel to the processor. The proposed mechanism is responsible for transforming sequences of instructions at run-time to be executed on a dynamic coarse-grain reconfigurable array, supporting speculative execution. This way, it is possible to take advantage of using pure combinational logic to speed up the execution, maintaining full binary compatibility in a totally transparent process. Three different case studies were evaluated: a Java Processor and a MIPS R3000 – representing the embedded systems field – and the Simplescalar Toolset, a widely used toolset that simulates a superscalar architecture based on the MIPS R10000 processor – representing the general-purpose market.

Keywords: Reconfigurable Array, Binary Translation, Stack Machines, MIPS.

Uma Arquitetura Reconfigurável Transparente para Aplicações Heterogêneas

RESUMO

Atualmente, pode-se observar que a Lei de Moore vem estagnando. A freqüência de operação já não cresce da mesma forma, e a potência consumida aumenta drasticamente em processadores de propósito geral. Ao mesmo tempo, sistemas embarcados vêm se tornando cada vez mais heterogêneos, caracterizados por uma grande quantidade de modelos computacionais diferentes, sendo executados em um mesmo dispositivo. Desta maneira, como novas tecnologias que irão substituir totalmente ou parcialmente o silício estão surgindo, novas soluções arquiteturais são necessárias. Apesar de sistemas reconfiguráveis já terem demonstrado serem candidatos em potencial para os problemas supracitados, ganhos significativos de desempenho são alcançados apenas em programas que manipulam dados massivamente, não representando a realidade dos sistemas atuais. Ademais, o seu uso em alta escala ainda está limitado à utilização de ferramentas ou compiladores que, claramente, não mantêm a compatibilidade de software e a reutilização do código binário já existente. Baseando-se nestes fatos, a presente tese propõe uma nova técnica para, utilizando um sistema reconfigurável, otimizar tanto programas orientados a dados como aqueles orientados a controle, sem a necessidade de modificação do código fonte ou binário. Para isto, um algoritmo de Tradução Binária, que trabalha em paralelo ao processador, foi desenvolvido. O mecanismo proposto é responsável pela transformação de seqüências de instruções, em tempo de execução, para serem executadas em uma unidade funcional reconfigurável de granularidade grossa, suportando execução especulativa. Desta maneira, é possível aproveitar as vantagens do uso da lógica combinacional para aumentar o desempenho e reduzir o gasto de energia, mantendo a compatibilidade binária em um processo totalmente transparente. Três diferentes estudos de caso foram feitos: os processadores Java e MIPS R3000 – representando o campo de sistemas embarcados – e o conjunto de ferramentas Simplescalar, que simula um processador superescalar baseado no MIPS R10000 – representando o mercado de processadores de propósito geral.

Palavras-chave: Sistemas Reconfiguráveis, Tradução Binária, Máquina de pilha, MIPS.

1 INTRODUCTION

The possibility of increasing the number of transistors inside an integrated circuit with the passing years, according to Moore's Law, has been pushing performance at the same level of growth. However, this law, as known today, will no longer exist in a near future. The reason is very simple: physical limits of silicon (KIM et al., 2003) (HOMPSON, 2005). Because of that, new technologies that will completely or partially replace silicon are arising. The problem is that, according to the ITRS roadmap (SEMICONDUCTOR, 2008), these technologies have a high level of density and are slower than traditional scaled CMOS, or the opposite: new devices can achieve higher speeds but with a huge area and power overhead – even when comparing to future CMOS technology.

Additionally, high performance architectures as the diffused superscalar machines are achieving their limits. As it is shown in (FLYNN; HUNG, 2005) and (SIMA; FALK, 2004), there are no novelties in such systems. The advances in ILP exploitation are stagnating: considering the Intel's family of processors, the IPC rate has not increased since the Pentium Pro in 1995, as Figure 1.1 shows. This occurs because these architectures are challenging some well-known limits of the ILP (WALL, 1991). Recent increases in performance have occurred mainly thanks to boosts in clock frequency through the employment of deeper pipelines. Even this approach, though, is reaching its limit. For example, the clock frequency of Intel's Pentium 4 processor had a modest increase from 3.06 to 3.8 GHz between 2002 and 2006 (INTEL, 2008).

Another trend is the so-called "Mobile Supercomputers" (AUSTIN et al., 2004). In the future, embedded devices will need to perform some intensive computational programs, such as real-time speech recognition, cryptography, augmented reality etc, besides the conventional ones, like word and email processing. According to the cited work, they must not exceed 75mW of power consumption. Figure 1.2 reinforces the trend demonstrated by Figure 1.1: even considering desktop computer processors, new architectures will not meet the requirements for future embedded systems. The star indicates where the expected future mobile computer requirements should be in terms of performance. The other curves represent important characteristics that will restrict performance improvements in those systems:

- Lack of ILP, as discussed before;
- Restrictions in the critical path of the pipeline stages: Intel's Pentium 4 microprocessor has only 12 fanout-of-four (FO4) gate delays per stage, leaving little logic that can be bisected to produce higher clocked rates. This becomes even worse considering that the delay of those FO4 will increase comparing against other circuitry in the system. One already can see this

trend in the newest Intel processors based on the Core and Core2 architectures, which have less pipeline stages than the Pentium 4.

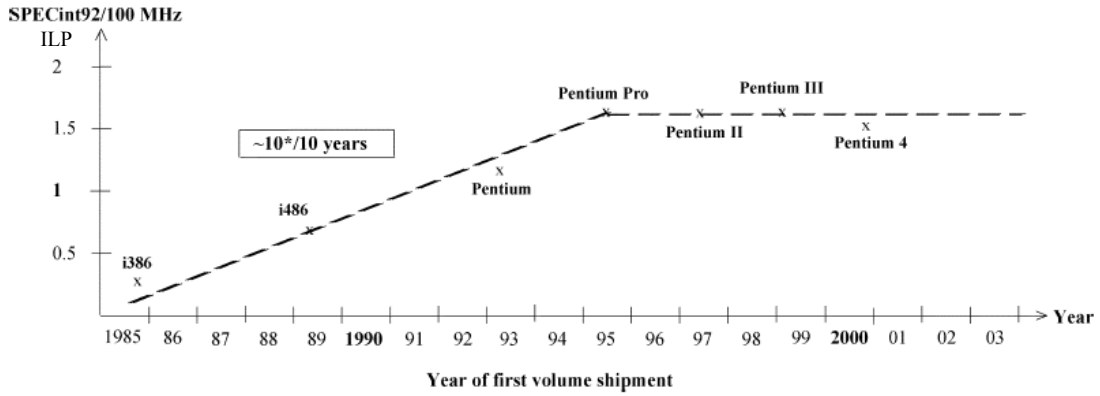


Figure 1.1: There is no improvements regarding the IPC in the Intel's Pentium Family of processors (SIMA; FALK, 2004)

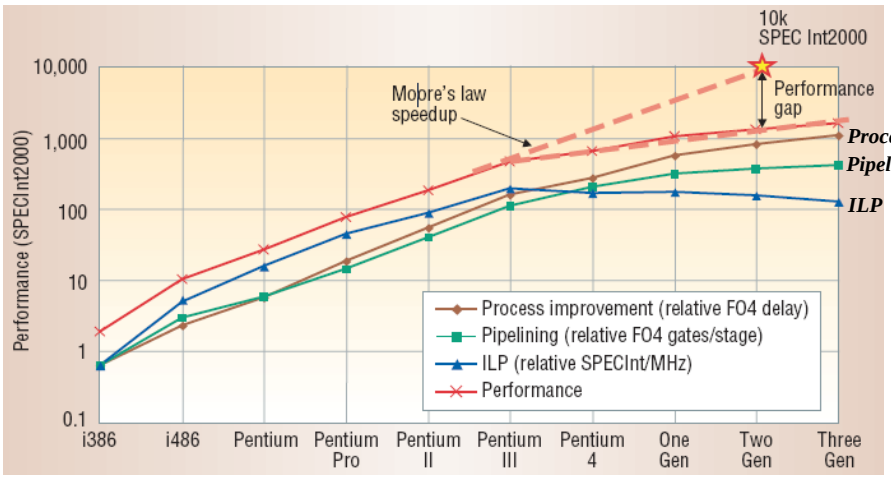


Figure 1.2: Near future limitations of performance, ILP and pipelining (AUSTIN et al., 2004)

Another concern is the excessive power consumption. As previously stated about performance, power spent by future systems is also far from the expected, as it can be observed in Figure 1.3. Another issue that must be pointed out is that leakage power is becoming more important and, while a system is in standby mode, it will be the dominant power consumed by it.

This way, one can observe that companies are migrating to chip multiprocessors to take advantage of the extra area available, even though, as this thesis will show, there is still a huge potential to speed up a single thread software. Therefore, the clock frequency increase stagnation, excessive power consumption and higher hardware costs to ILP exploitation together with the foreseen slow technologies are new architectural challenges to be dealt with. Hence, new alternatives that can take advantage of the integration possibilities and that can address the performance and power issues stated before become necessary.

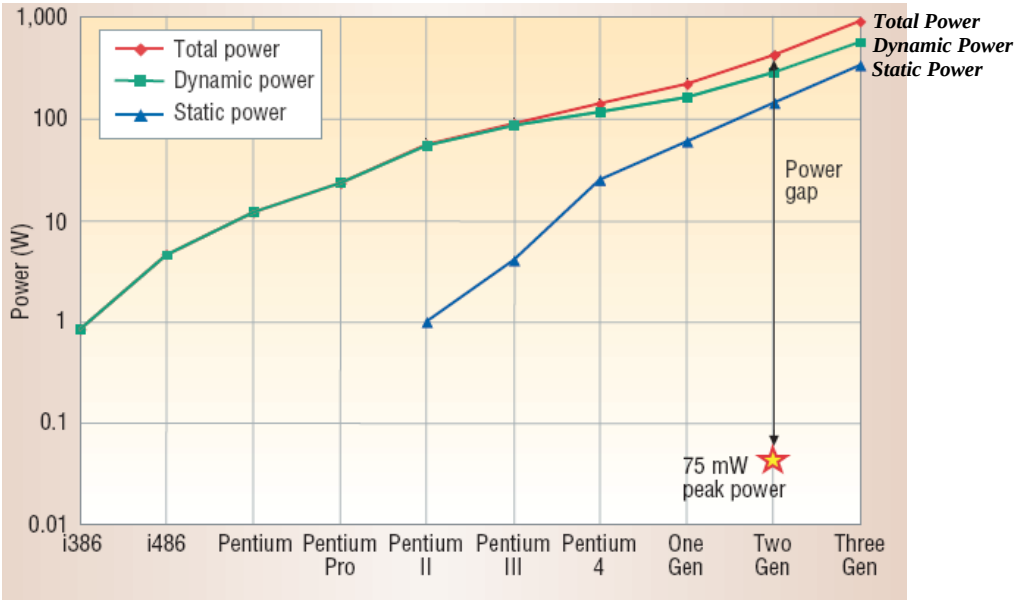


Figure 1.3: Power consumption in present and future desktop processors (AUSTIN et al., 2004)

In this scenario, reconfigurable architectures appear to be an attractive solution. By translating a sequence of code into combinational logic, one can have huge performance gains with energy savings, at the price of extra area – exactly the only resource available nowadays and in future technologies (GUPTA; MICHELI, 1993) (VENKATARAMANI et al., 2001) (STITT; VAHID, 2002). At the same time that reconfigurable computing can explore the ILP of the applications, it can also speed up sequences of data dependent instructions, which is its main advantage when comparing to traditional architectures. Furthermore, as reconfigurable architectures are highly based on regular circuits, another advantage emerges: it is common sense that as the more the technology shrinks to 65 nanometers and below, the harder it will be to print the geometries employed today, directly affecting the yield (OR-BACH, 2001). Moreover, because circuit customization is a very expensive process, regular circuits customized in the field are also considered as the new low cost solution.

However, reconfigurable systems have two main drawbacks. The first one is that they are designed to handle very data intensive or streaming workloads. This means that the main design strategy is to consider the target applications as having very few distinct kernels for optimization. By speeding up small parts of the software, huge gains would be achieved. In contrast, a desktop system usually executes a large number of applications with different behaviors at the same time; and the number of applications that a single embedded device must handle is growing, as well as the heterogeneity of their behaviors.

The second problem is that the process of mapping pieces of code to reconfigurable logic usually involves some kind of transformation, manual or using special languages or tool chains. These transformations modify somehow the source or the binary code, precluding the wide spread usage of reconfigurable systems. As the old X86 ISA has been showing, sustaining binary compatibility, allowing legacy code reuse

and traditional programming paradigms are key factors to reduce the design cycle and maintain backward compatibility.

Based on these two main concerns discussed above, this thesis proposes the use of a technique called Dynamic Instruction Merging (DIM), which is a Binary Translation (BT) method implemented in hardware. It is used to detect and transform sequences of instructions at run time to be executed in a reconfigurable array. DIM is a totally transparent process: there is no need for changing the code before its execution at all, allowing full binary code reuse. With the BT mechanism, it is possible to ensure software compatibility at any level of the design cycle, allowing the utilization of a reconfigurable hardware without requiring any tools for the hardware/software partitioning or special compilers.

The employed reconfigurable unit is a coarse-grained array, composed of simple functional units and multiplexers. Being not limited to the complexity of fine-grain configurations and using the special BT mechanism, the proposed system can also speed up control-flow oriented software, without any distinct kernel subject to optimization. Consequently, it is possible to increase the performance of any kind of software as well as reduce the energy consumption, not being limited to just DSP-like or loop centered applications, as reconfigurable systems usually do (STITT et al., 2003).

This proposal can be applied in the embedded system domain as well as in the general-purpose one, and in this work these both fields of application are analyzed. In the following sections, the motivations to implement the proposed technique are discussed in more details.

1.1 Main Motivations

In this section, we discuss in more details the main motivations that inspired our work. The first one relates to the hardware limits and costs that architectures are facing in order to increase the ILP of the running application. Since the searching for ILP is becoming more difficult, the second motivation is based on the use of pure combinational logic as a solution to speed up instructions execution. However, even a technique that could increase the performance should be passive of implementation in nowadays technology and still sustain binary compatibility. The possibilities of implementation and implications of code reuse lead to our next motivation. Finally, the last one discusses about the future and new technologies, where the reliability and yield costs will become even more important, with regularity playing a major role.

1.1.1 Overcoming some limits of the parallelism

In the future, advances in compiler technology together with significantly new and different hardware techniques may be able to overcome some limitations of the ILP exploitation. However, it is unlikely that such advances, when coupled with realistic hardware, will overcome all these limits. Instead, the development of new hardware and software techniques will continue to be one of the most important challenges in computer design.

To better understand the main issues related to ILP exploitation, in (HENNESSY; PATTERSON, 2003) assumptions are made for an ideal (or perfect) processor, as follows:

1. Register renaming – It is the process of renaming target registers in order to avoid false dependences (Write after Read and Write after Write). This way, it is

possible to better explore the parallelism of the running application. The perfect processor would have an infinite number of virtual registers available to perform this renaming and hence all false dependences could be avoided. Therefore, an unbounded number of data independent instructions could begin to be simultaneously executed.

2. Branch prediction – It is the mechanism responsible for figuring out if the branches will be taken or not taken, depending on where the execution currently is. The main objective is to diminish the number of pipeline stalls due to taken branches. It is also used as a part of the speculative mechanism to execute instructions beyond basic blocks. In an ideal processor, all conditional branches would be correctly predicted, meaning that the predictor is perfect.

3. Jump prediction – In the same manner, all jumps are perfectly predicted. When combined with the branch prediction, previously discussed, the processor could have a perfect speculation mechanism and an unbounded buffer of instructions available for execution.

4. Memory-address alias analysis – It is the comparison among references to memory encountered in instructions. Some of these references are calculated at run-time and, as different instructions can access the same address of the memory in a different order, data coherence problems can arise. In the perfect processor, all memory addresses would be exactly known before actual execution begins and a load could be moved before a store, once provided that both addresses are not identical.

While assumptions 2 and 3 would eliminate all control dependences, assumptions 1 and 4 would eliminate all but the true data dependences. Together, these assumptions mean that any instruction belonging to the program's execution could be scheduled on the cycle immediately following the execution of the predecessor on which it depends. It is even possible, under these assumptions, for the last dynamically executed instruction in the program to be scheduled on the very first cycle. Thus, this set of assumptions subsumes both control and address speculation and implements them as if they were perfect.

The analysis of the hardware costs to get as close as possible of this ideal processor is quite complicated. For example, let us consider the instruction window, which represents the set of instructions that are examined for simultaneous execution. In theory, a processor with perfect register renaming should have an instruction window of infinite size, so it could analyze all the dependencies at the same time.

To determine whether n issuing instructions have any register dependencies among them, assuming all instructions are register-register and the total number of registers is unbounded, one must perform comparisons. Thus, to detect dependences among the next 2000 instructions requires almost four million comparisons. Even issuing only 50 instructions requires 2,450 comparisons. This cost obviously limits the number of instructions that can be considered for issue at once. To date, the window size has been in the range of 32 to 126, which can require over 2,000 comparisons. The HP PA 8600 reportedly has over 7,000 comparators (HENESSY; PATTERSON, 2003).

Another good example to illustrate how much hardware a modern superscalar design needs to execute instructions in parallel is the Alpha 21264 (KESSLER, 1999). It issues up to four instructions per clock and initiates execution on up to six (with significant restrictions on the instruction type, e.g., at most two load/stores), supports a large set of renaming registers (41 integer and 41 floating point, allowing up to 80

instructions in-flight), and uses a large tournament-style branch predictor. Not surprisingly, half of the power consumed by this processor is related to the ILP exploitation (WILCOX; MANNE, 1999).

Other possible implementation constraints in a multiple issue processor, besides the ones cited before, include: issues per clock, functional units and unit latency, number of register file ports, functional unit queues, issue limits for branches, and limitations on instruction commit.

1.1.2 Using the Pure Combinational Logic Circuit Advantages

There are always potential gains when passing the execution from sequential to combinational logic. Using a combinational mechanism could be a solution to speed up the execution of sequences of instructions that must be executed in order, due to data dependencies. This concept is better explained with a simple example. Let us have an $n_x n$ bit multiplier, with input and output registers. By implementing it with a cascade of adders, one might have the execution time, in the worst case, as follows:

$$T_{mult\_combinational} = t_{ppFF} + 2*n*t_{cell} + t_{setFF} \quad (1)$$

where t_{cell} is the delay of an AND gate plus a 2-bits full-adder, t_{ppFF} the time propagation of a Flip-Flop, and t_{setFF} the set time of the Flip-Flop.

The area of this multiplier is

$$A_{combinational} = n^2 * A_{cell} + A_{registers} \quad (2)$$

considering A_{cell} and $A_{registers}$ as the area occupied by the cell and registers, respectively.

If one could do the same multiplier by the classical shift and add algorithm, and assuming a carry propagate adder, the multiplication time would be

$$T_{mult\_sequential} = n * (t_{ppFF} + n*t_{cell} + t_{setFF}) \quad (3)$$

And the area given by

$$A_{sequential} = n*A_{cell} + A_{control} + A_{registers} \quad (4)$$

with $A_{control}$ being the area overhead due to the control unit.

Comparing equations (1) with (3), and (2) with (4), it is clear that by using a sequential circuit one trades area by performance. Any circuit implemented as a combinational circuit will be faster than a sequential one, but will most certainly take much more area.

1.1.3 Software Compatibility and Reuse of Binary Code

Among thousands of products launched every day, one can observe those which become a great success and those which completely fail. The explanation perhaps is not just about their quality, but it is also about their standardization in the industry and the concern of the final user on how long the product he is acquiring will be subject to upgrades.

The x86 architecture is one of these major examples. The X86 ISA itself did not follow the last trends in processor architectures at the time of its deployment. It was developed at a time when memory was considered very expensive and developers used to compete on who would implement more and different instructions in their architectures. Its ISA is a typical example of a traditional CISC machine. Nowadays, to handle with that, the newest X86 compatible architectures spend extra pipeline stages

plus a considerable area in control logic and microprogrammable ROM just to decode these CISC instructions into RISC like ones. This way, it is possible to implement deep pipelining and all other high performance RISC characteristics maintaining the x86 instruction set and, as a consequence, backward compatibility.

Although new instructions have been included in the x86 original instruction set, like the SIMD MMX and SSE instructions, targeted to multimedia applications, there is still support to the original 80 first instructions implemented in the very first X86 processor. This means that any software written for any x86 in any year, even at the end of seventies, can be executed on the last Intel processor. This is one of the keys to success of this family: the possibility of reusing the existing binary code, without any kind of modification. This characteristic, called software compatibility, was one of the reasons of why this product became the leader in its market. Intel could guarantee to its consumers that their programs would not be surpassed during a long period of time and, even when changing the system to a faster one, they would still be able to reuse the same software again without any kind of modification.

Probably this is the main reason why companies such as Intel and AMD keep implementing more power consuming superscalar techniques and trying to increase the frequency of operation to the extreme. More accurate branch predictors, more advanced algorithms for parallelism detection, or the use of SMT architectures like the Intel Pentium IV with Hyperthreading (KOUFATY; MARR, 2003) or SIMD extensions instructions such as MMX and SSE (CONTE, 1997), are some of them. However, the basic principle of high performance architectures keeps the same: superscalarity. While the x86 market is expanding even more, we observe a decline in the use of more elegant and efficient instruction system architectures, such as the Alpha and the PowerPC processors.

1.1.4 Increasing Yield and Reducing Manufacture Costs

In (OR-BACH, 2001), a discussion is made about the future of the processes of fabrication using new technologies. According to it, standard cells, as they are today, will not exist anymore. As the manufacturing interface is changing, regular fabrics will soon become a necessity. How much regularity versus how much configurability is still an open question, as well as the granularity of these regular circuits. Regularity can be understood as a part which composes a whole, in the level of gates, standard-cells, standard-blocks and so on. What is almost a consensus is the fact that the freedom of the designers, represented by the irregularity of the project, will be more expensive in the future. By the use of regular circuits, the design company will decrease costs, as well as the possibility of manufacturing faults, since the reliability of printing the geometries employed today in 65 nanometers and below will be a big issue.

Nowadays, the resources to create an ASIC design of moderate high volume, complexity and low power, are considered very high. Some design companies can do it because they have experienced designers, infrastructure and expertise. However, for the same reasons, there are companies that just cannot afford it. For these companies, a more regular fabric seems the best way to go as a compromise using an advanced process. As an example, in 1997 there were 11,000 ASIC design startups. This number dropped to 1,400 in 2003 (VAHID et al., 2003). The mask cost seems to be the primary problem. The estimative in 2003 for the ASIC market is that it had 10,000 designs per year with a mask cost of $20,000. The mask cost for 90-nanometer technology is around

$2 million. This way, to maintain the same number of ASIC designs, their costs need to return to tens of thousands of dollars, not millions.

Moreover, it is very likely that the cost of doing the design and verification is growing in the same proportion, increasing even more the final cost. Table 1.1 shows sample non-recurring engineering (NRE) costs for different CMOS IC technologies (VAHID et al., 2003). At 0.8 mm technology, the NRE costs were only about $40,000. With each advance in IC technology, the NRE costs have increased dramatically. NRE costs for 0.18 mm design are around $350,000, and at 0.13 mm, the costs are over $1 million. This trend is expected to continue at each subsequent technology node, making it more difficult for designers to justify producing an IC using nowadays technologies.

Furthermore, the time it takes for a design to be manufactured at a fabrication facility and returned to the designers in the form of an initial IC is also increasing. Table 1.1 also provides the turnaround times for various technology nodes. The turnaround times for manufacturing an IC have almost doubled between 0.8 and 0.13 mm technologies. Longer turnaround times lead to larger design costs and even possible loss of revenue if the design is late to the market.

Table 1.1: IC non-recurring engineering (NRE) costs and turnaround time (VAHID et al., 2003)

	Technology (µm)			
	0.8	***0.35***	***0.18***	***0.13***
NRE (K)	$40	$100	$350	$1000
Turnaround (days)	42	49	56	76

Because of all these reasons discussed before, there is a limit in the number of situations that can justify producing designs using the latest IC technology. Less than 1000 out of every 10,000 ASIC designs have high enough volumes to justify fabrication at 0.13 mm (VAHID et al., 2003). Therefore, if design costs and times for producing a high-end IC are becoming increasingly large, just few of them will justify their production in the future.

The problems of increasing design costs and long turnaround times are made even more noticeable due to increasing market pressures. The time during which a company seeks to introduce a product into the market is shrinking. This way, the designs of new ICs are increasingly being driven by time to market concerns. With the constant increase of the productivity gap, regularity has also another very interesting characteristic: its scalability. Being regular, a circuit can be adapted to the product needs, according to the niche of market it is targeted to. As scalability is one of the consequences of regularity, it can be applied at different or higher levels. For instance, the IBM Cell processor relies on the regularity of its multimedia processors (called Synergistic Processing Elements), which compose the majority of the total system area. Because of that, using regular circuits will also amortize costs related to NRE, since it facilitates the design, testing and reuse of circuits.

Summarizing, there will be a crossover point where, if the company needs a more customized silicon implementation, it needs be to able to afford the mask and production costs. However, economics are clearly pushing designers toward more

regular structures that can be manufactured in larger quantities. Regular fabric would solve the mask cost and many other issues such as printability, extraction, power integrity, testing, yield etc.

1.2 Main Contributions

Taking into consideration all the motivations discussed before, the main novelty of this work is the complete dynamic nature of the reconfigurable array: besides being dynamic reconfigurable, the sequences of instructions to be executed on it are also detected and transformed to an array's reconfiguration at run-time. The reconfigurable logic is represented by a coarse-grain array, tightly coupled to the processor, meaning that it works as another ordinary functional unit in the processor. It is composed by off the shelf functional units, as ALUs and multipliers, to perform the computation, and by a set of multiplexers, responsible for the routing.

As already explained, the approach is based on a special BT hardware called Dynamic Instruction Merging (DIM). DIM is designed to detect and transform instruction groups for reconfigurable hardware execution at run time. As can be observed in Figure 1.4, this is done concurrently while the main processor fetches other instructions. When a sequence of instructions is found, following given policies that will be explained later, a binary translation is applied to it. This BT transforms this sequence of instructions in a configuration of the array, which will perform exactly the same function. Thereafter, this configuration is saved in a special cache, and indexed by the program counter (PC).

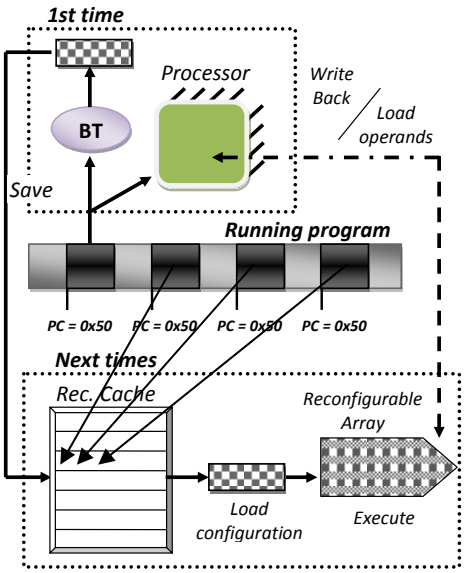


Figure 1.4: The proposed approach

The next time the saved sequence is found, the dependence analysis is no longer necessary: the processor loads the previously stored configuration from the special cache, the operands from the register bank, and activate the reconfigurable hardware as functional unit. Then, the array executes that configuration in hardware (including write back of the results), instead of using the datapath of the processor. Finally, the PC is updated, in order to continue with the execution of the normal (not translated)

instructions. Figure 1.5 shows a systematic illustration, separating in steps both detection and reconfiguration/execution phases.

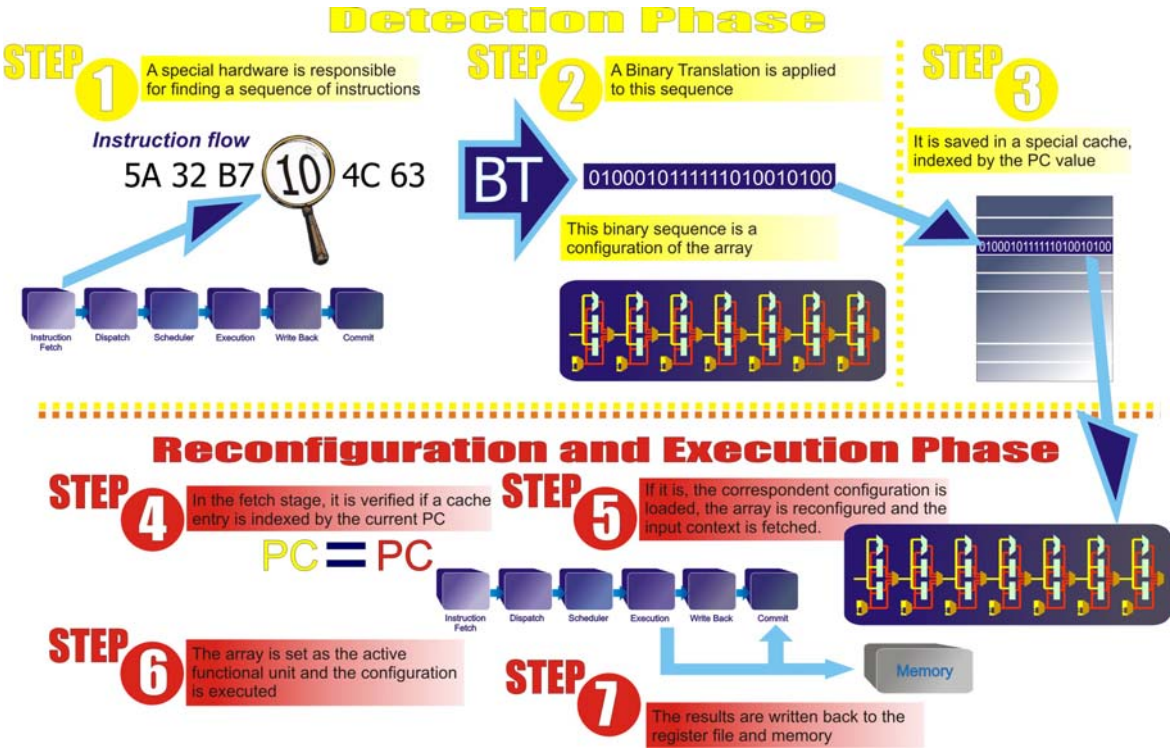


Figure 1.5: A systematic illustration of the technique

By transforming any sequence of opcodes into a single combinational operation in the array, one can achieve great gains, since fewer accesses to program memory and less iterations on the datapath are required. Depending on the size of the special cache used to store the configurations, the optimization can be extended to the entire application, not being limited to very few hot spots. Hence, it is possible to avoid the use of a reconfigurable hardware for just a single algorithm, which would have a prohibitively high cost. Furthermore, the proposed technique is not dependent on the available application parallelism to speed up the code to be executed, but rather on sequences of instructions that appear several times in the code.

In a certain way, the approach saves the dependence information of the sequences of instructions. This way, repetitive dependence analysis for the same sequence of instructions is avoided, as superscalar processors do. It is interesting to point out that almost half of the number of pipeline stages of the Pentium IV processor is related to dependence analysis (INTEL, 2008); and half of the power consumed by the core of the Alplha 21264 processor is also related to extraction of dependence information among instructions (WILCOX; MANNE, 1999).

In this technique, by coupling the array with a BT mechanism, software compatibility in any level of the design cycle can be assured, without requiring any tools for the hardware/software partitioning or special compilers, allowing easy software porting for different machines tracking technological evolutions. Both the DIM engine and the reconfigurable array are designed to work in parallel to the processor and do not introduce any delay overhead or penalties for critical path of the pipeline structure.

In the first part of this work, the proposed approach was implemented on a Java software compliant architecture targeted to the embedded system domain (BECK; CARRO, 2005) (BECK; CARRO, 2005b) (GOMES et al., 2005) (RUTZIG et al., 2007). The employed processors were two different versions of the Femtojava Processor (BECK; CARRO, 2003B) (GOMES et al., 2004). It was demonstrated great performance improvements and reduction in energy consumption, even when compared to a VLIW version of the same architecture (BECK; CARRO, 2004) (BECK; CARRO, 2004b). It was also showed that the BT Algorithm can take advantage of the particular computational method of stack machines in order to perform the detection with a low complexity (BECK; CARRO, 2005c) (GOMES et al., 2005b). Furthermore, it was compared to traditional methods of detection of RISC machines (GOMES et al., 2005).

The same technique was implemented in two RISC like architectures, one representing the general purpose computation and the other one the embedded systems field. For the first, studies have been done (BECK et al., 2007) (BECK et al., 2006) (BECK et al., 2006b) using the Simplescalar Toolset (BURGER; AUSTIN, 1997) together with the benchmark suite MIBench (GUTHAUS et al., 2001). For the second, a processor based on the MIPS R3000 was used (BECK et al., 2008), executing the same benchmark set. As could be expected, there are differences in the structure of the array as well as in the detection algorithm when comparing to the Java implementation, since in the previous implementation a stack machine was used – although similar results in terms of performance and energy were achieved.

As it will be demonstrated in the next chapters, the following advantages can be obtained using this approach, that overcomes problems presented by high performance nowadays architectures.

It achieves:

- High performance;
- Low energy consumption.

In opposite to existing reconfigurable systems:

- It is applicable to any kind of algorithm with different behaviors (control and dataflow oriented software);
- It can optimize even algorithms with no distinct kernels available.
- It is technology independent – an FPGA is not necessary for its implementation;
- It maintains binary compatibility. This way, the process is totally transparent for the programmer and there is no need of any kind of modification in the source code nor the use of special tools.

Moreover, other advantages are demonstrated:

- It is highly based on a regular circuit. It means that is possible to increase the yield in future technologies.
- It is easily scalable – the size of the reconfigurable logic can vary depending on the application needs.
- Although it is area consuming, it still can be implemented even considering nowadays technologies.

Chapters 2 and 3 present the related work, clarifying some design choices that have been done and that guided this work. Chapter 2 discusses issues related to reconfigurable fabric. The potential of executing sequences of instructions in pure combinational logic is also shown. Moreover, a high-level comparison between two different array granularities is performed, together with a detailed analysis of the behaviors of a set algorithms and which one is more suitable to be executed on each kind of reconfigurable logic. The potential of performance improvements presented by various reconfigurable systems are also discussed in Chapter 2, demonstrating that these architectures can present performance boosts just on a very specific subset of benchmarks – which does not reflect the reality of the whole set of applications both embedded and general purpose systems are executing in these days. In Chapter 3 two techniques related to dynamic optimization – trace reuse and binary translation – are shown in details. Then, studies that already used in somehow both approaches with reconfigurable architectures are discussed. Finally, the contribution and main novelty of this work is pointed out, comparing it against these other studies.

In Chapter 4 the structure of the reconfigurable array and how it is coupled to the target architectures are demonstrated. Chapter 5 discusses the algorithm: how instructions are detected and transformed at run time in configurations to be executed on the reconfigurable logic. As stated before, as case studies three different architectures were evaluated, representing both embedded systems and general purpose domains, with RISC and Java machines. This way, Chapters 4 and 5 are each divided in two different sections targeting these different computational methods. The impact of using a stack machine against a RISC based one when designing the reconfigurable array and the binary translation algorithm is also analyzed.

Chapter 6 shows the methodology and tools employed to gather the results considering all architectures executing a large range of benchmarks, concerning performance, area, power and energy consumption. Finally, the last chapter discusses future work and concludes this thesis.

2 RECONFIGURABLE SYSTEMS

2.1 Principles

Reconfigurable systems have already shown to be very effective, implementing some parts of the software in a hardware reconfigurable logic. Using the same idea of instruction reuse, by translating a sequence of operations into a combinational circuit performing the same computation, one could speed up the system and reduce energy consumption at the price of extra area. Huge software speedups (GUPTA; MICHELI, 1993) (VENKATARAMANI et al., 2001) (GAJSKI, 1998) (HENKEL, ERNST, 1997) as well as system energy reduction have been previously reported (HENKEL, 1999) (STITT; VAHID, 2002). In Figure 2.1, the basic principle of a computational system working together with a reconfigurable hardware is illustrated. As can be observed, the processor is responsible for the execution of a given part of the code, while the reconfigurable logic is employed to execute the rest of it, in a more efficient manner.

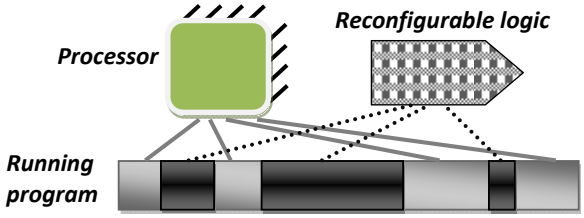


Figure 2.1: The basic principle of a system making use of reconfigurable logic

Reconfigurable systems have the capability to adapt themselves to a given application, providing hardware specialization to it. Through this adaptation, they are expected to achieve a great improvement in performance, when compared to fixed instruction set processors. However, because of this certain level of flexibility, the gains are not as high as in Application Specific Instruction Set Processors (ASIPs) (JAIN et al., 2001). This way, as ASIPs have specialized hardware that accelerate the execution of the applications it was designed for, a system with reconfigurable capabilities would have the almost same benefit without having to commit the hardware into silicon: it can be adapted after design, in the same way programmable processors can adapt to application changes.

A reconfigurable system includes a set of programmable processing units called reconfigurable logic, which can be reconfigured in the field to implement logic operations or functions, and programmable interconnections called reconfigurable fabric. The reconfiguration is achieved by downloading from a memory a set of configuration bits called configuration context, which determines the functionality of reconfigurable logic and fabric. The time needed to configure the whole system is called reconfiguration time, while the memory required for storing the reconfiguration data is

called context memory. Both the reconfiguration time and context memory constitute the reconfiguration overhead.

Reconfigurable Instruction Set Processors, also known as RISP (COMPTON; HAUCK, 2000), will be the focus of this section. Usually a RISP has a special unit called RPU (Reconfigurable Processor Unit), responsible for the actual computation of a part of the software in the reconfigurable logic.

2.2 Advantages of using Reconfigurable Logic

The widely used Patterson (HENESSY; PATTERSON, 2003) metrics of relative performance through measures such as IPC are well suited for comparing different processor technologies and ISA, as it abstracts concepts such as clock frequency. As described in (SIMA; FALK, 2004), however, to better understand the performance evolution in the microprocessor industry, it is interesting to note the absolute processor performance (*Ppa*) metric denoted as:

$$P_{pa} = fc * 1/CPII * IPII * OPI \text{ (operations/sec)} \quad (1)$$

In equation (1), *CPII*, *IPII* and *OPI* are described respectively as Cycles Per Issue Interval, Instructions Per Issue Interval and Operation per Instructions, while *fc* is the operating clock frequency. The first two metrics, when multiplied, form the known *IPC* rate. Nevertheless, it is interesting to keep these factors separated in order to better expose speed-up potentials.

The *CPII* rate informs the intrinsic temporal parallelism of the microarchitecture, showing how frequently new instructions are issued to execution. The *IPII* variable is related to the issue parallelism, or the average number of dynamically fetched instructions issued to execution per issue interval. Finally, the *OPI* metric measures intra-instruction parallelism, or the number of operations that can be issued through a single binary instruction word. It is important to note that one should distinguish the *OPI* from the *IPII* rate, since the first reflects changes in the binary code that should be adapted statically to boost intrainstruction parallelism, such as data parallelism found in SIMD architectures. Figure 2.2 illustrates these three metrics.

Throughout the microprocessor evolution history, several approaches have been considered to improve performance by manipulating one or more of the factors of equation (1). One of these approaches, for example, dealt with the *CPII* metric by increasing instructions throughput with pipelining (HENNESSY, PATTERSON, 2003). Moreover, the *CPII* metric has also been well covered with efficient branch prediction mechanisms and with memory hierarchies, though this metric is still limited by pipeline stalls such as the ones obtained with cache misses. The *OPI* rate has also been dealt with the development of complex *CISC* instructions or *SIMD* architectures.

On the other hand, few solutions other than the superscalar approach since the 90's explored the opportunity of increasing the *IPII* rate. This is an interesting fact, since differently from the *OPI* rate, increasing the *IPII* raises excellent possibilities of improvements, as it does not require changes or extensions to the ISA and, as a consequence, maintains backward software compatibility.

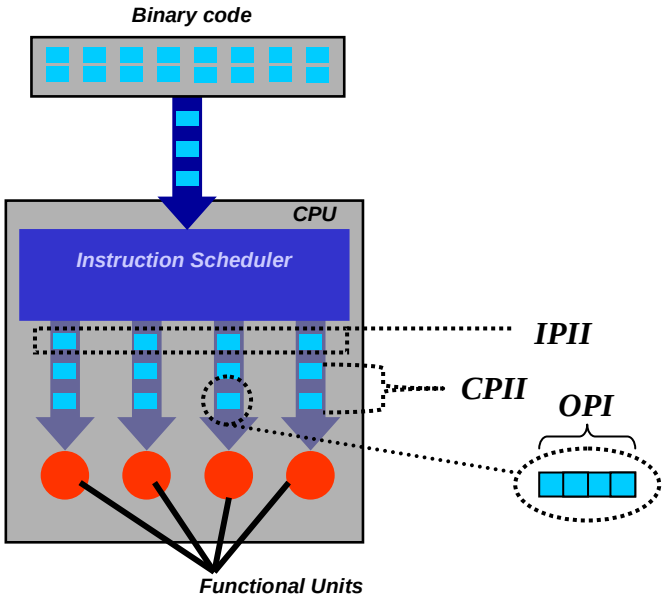


Figure 2.2: Definitions of *IPII*, *CPII* and *OPI*

2.2.1 Application

A reconfigurable system targets to increase exactly the *IPII* rate. The basic idea is to identify instructions that can be grouped and configured to execute in a reconfigurable array. Consequently, more instructions will be issued by issue interval (increasing the *IPII* rate). However, it can also influence the *CPII* rate, as it will be analyzed later. Therefore, temporal and issue parallelisms are both dynamically explored, and can be illustrated by the following equations:

IPII = (Number of Instructions)/(Number of issues) (2)

CPII = (Number of Cycles)/(Number of Issues) (3)

In order to increase the *IPII* number, it is necessary to increase the execution efficiency by decreasing the number of issues. Equation (4) shows how it is affected by the technique:

Number of Issues = Total number of executed Instructions + Number of Merged Instructions(1 – AMIL)* (4)

where the *Average Merged Instructions Length* (*AMIL*) is the average group size in number of instructions; while the *Number of Merged Instructions* counts how many merged instructions[1] were issued for execution in combinational logic. This can be represented by the following equation:

*Number of Merged Instructions = MIR * Total number of executed Instructions* (5)

[1] In this work the set of instructions that are executed on reconfigurable logic is called of *merged instructions*, because this name is related to the proposed technique; in previous works, several and different names have been using. However, there is no consensus about this nomenclature

MIR is denoted as the *Merged Instructions Rate*. This is an important factor as it exposes the density of grouped operations that can be found in an application. If *MIR* is equal to one, then the whole application was mapped into an efficient mechanism and there is no need of a processor, which is actually the case of a specialized ASIC or complete dataflow architectures.

Furthermore, doing a deeper analysis, one can conclude that the ideal *CPII* also equals to one, which means that the functional units are constantly fed by instructions every cycle. However, due to pipeline stalls or to instructions with higher delays, the *CPII* variable tends to be of a greater value. In fact, manipulating this factor is a bit more complicated, as both the number of cycles and the number of issues are affected by the execution of instructions in reconfigurable logic. As it will be shown in the example, there are times when the *CPII* will increase; this is actually a consequence of the augmented number of operations issued in an group of instructions.

This way, one thing that must be assured is that the *CPII* rate will not grow proportionally to hide the *IPC* gains caused by the increase of *IPII*. In other words, if the number of issues decreases, the number of cycles also has to decrease. Consequently, a fast mechanism is necessary for reconfiguring the hardware and executing instructions.

2.2.2 An Example

The following example illustrates the concept previously proposed.

Figure 2.3a shows a hypothetical trace with instructions *a*, *b*, *c*, *d* and *e*, and the cycles at which the instruction execution ends. If one considers that the implemented architecture has an *IPII* rate of one, typical of RISC scalar architectures, and that *inst d* causes a pipeline stall of 5 cycles, while all other instructions are executed in one cycle, this trace of 14 instructions would take 18 cycles to execute. This results in a *CPI* of 1.28.

If, however, instructions of number one to five are merged (which is represented by *Inst M*, as shown in Figure 2.3b), and executed in two cycles, the whole sequence would then be executed in 14 cycles. Note that the left column in Figure 2.3b represents the issue number of the instruction group. Therefore, one would find the following numbers: *CPII* = 1.5, *AMIL* = 5, and *MIR* = 1/14 = 0.07. Because of the capability of speeding up the fetch and execution of the merged instructions, the final *IPII* would increase to 1.4. Even though the *CPII* would increase from 1.28 to 1.5, the *IPC* rate would grow from 0.78 to 1.

Nevertheless, one could expect further improvements if merged instructions included *Inst d*, which caused a stall of 5 cycles in the processor pipeline. Supposing that the sequence of instructions *b*, *d* and *e* (issue numbers of 5, 6 and 7 in Figure 2.3b) is merged into instruction M2 and executed in 3 cycles, it would produce an impact on the *CPII* that would go down to 1.375 while the *IPII* would rise to 1.75, resulting in an IPC equals to 1.27. This is illustrated in Figure 2.3c. In other words, using a reconfigurable system the interval of execution between a set of instruction and another is longer than the usual. However, as more instructions are executed per time slice, *IPC* increases.

(a)

Number	Instruction	Cycle
1	inst a	1
2	inst b	2
3	inst b	3
4	inst a	4
5	inst c	5
6	inst b	6
7	inst a	7
8	inst a	8
9	inst b	9
10	inst d	10
11	inst e	15
12	inst b	16
13	inst c	17
14	inst a	18

(b)

Issue	Instruction	Cycle
1	*inst M*	1
2	inst b	3
3	inst a	4
4	inst a	5
5	inst b	6
6	inst d	11
7	inst e	12
8	inst b	13
9	inst c	14
10	inst a	15

(c)

Issue	Instruction	Cycle
1	*inst M*	1
2	inst b	3
3	inst a	4
4	inst a	5
5	*inst M2*	8
6	inst b	9
7	inst c	10
8	inst a	11

Figure 2.3: a) Execution trace of a given application; b) Trace with one merged instruction; c) Trace with two merged instructions

Later in this thesis, an ideal solution is analyzed, which is capable of executing merged instructions in just one cycle, meaning that the *CPII* inside the array is 1. This will show the potential gains of combinational logic when affecting the *AMIL* and *IPII* rates. Although this kind of assumption can be theoretically feasible, the area overhead in this case would be enormous. For example, operations such as multiplication normally are not performed in just one cycle.

Figure 2.4 graphically shows how the gains are obtained. As it can be seen, the upper part of the figure demonstrates the execution of several instructions, which are represented as boxes. Those that have the same texture represents instructions that have data-dependency and hence cannot be executed in parallel. Still, non-dependent instructions can be parallelized. On the other hand, by using the combinational data-driven approach, one is able to reduce the time spent executing several minor operations in the processor pipeline at the cost of extra area. This represents the tradeoff between sequential and combinational logic.

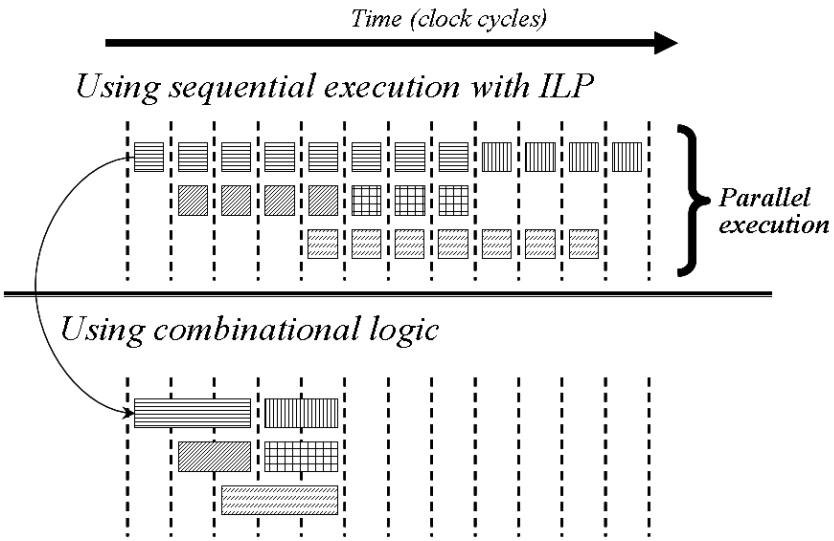


Figure 2.4: Gains obtained when using combinational logic

2.3 Classification

In the reconfigurable field, there is a great variety of classifications, as it can be observed in some surveys published about the subject (BARAT; LAUWEREINS, 2000) (COMPTON; HAUCK, 2000). In this revision, the most common ones are cited, since there is still no consensus about this taxonomy.

2.3.1 RPU Coupling

How the RPU is coupled, or connected to the main processor, defines how the interface between both of them works, including issues related to how data is transferred and how the synchronization between the parts is performed.

The position of the RPU, relative to the microprocessor, directly affects performance. The benefit obtained from executing a piece of code in the RPU depends on communication and execution costs. The time needed to execute an operation in the RPU is the sum of the time needed to transfer the processed data and the time required to process it. If this total time is smaller than the time it would normally take in the processor alone, then an improvement can be obtained.

The RPU can be allocated in three main places relative to the processor:

- Attached to the processor: The reconfigurable logic communicates to the main processor through a bus.
- Coprocessor: The reconfigurable logic is located next to the processor. The communication usually is done using a protocol similar to those used for floating point coprocessors.
- Functional Unit: The logic is placed inside the processor. It works as an ordinary functional unit. The decoder of the processor is responsible to activate it, when necessary.

Figure 2.5 illustrates these three different types of RPU coupling. The two first interconnection schemes are usually called loosely coupled. The functional unit approach, in turn, is named as tightly coupled. As stated before, the efficiency of each

technique depends on two things: the time of data transfer between the components, where, in this case, the functional unit approach is the fastest one and the attached processor, the slowest; and the quantity of instructions executed by the RPU. Usually, loosely coupled RPUs can execute larger chunks of code, and are faster than the tightly coupled ones – mainly because they have more area available. For this kind of RPU, there is a need for faster execution times: it is necessary to overcome some of the overhead brought by the high delays presented by the data transfer.

A tightly coupled RPU, although occupying more die area (where the processor is implemented), makes the control logic simpler, and diminishes the overhead required in the communication between the reconfigurable array and the rest of the system, because it can share some resources with the processor, such as the access to the register bank. Then, when there is a RPU working as functional unit in the main processor, it is called a Reconfigurable Functional Unit, or RFU. The first reconfigurable systems were implemented as co-processors, or as attached processors. However, with the manufacturing advances, with more transistors available within the same die, the RFU approach is becoming a very common implementation.

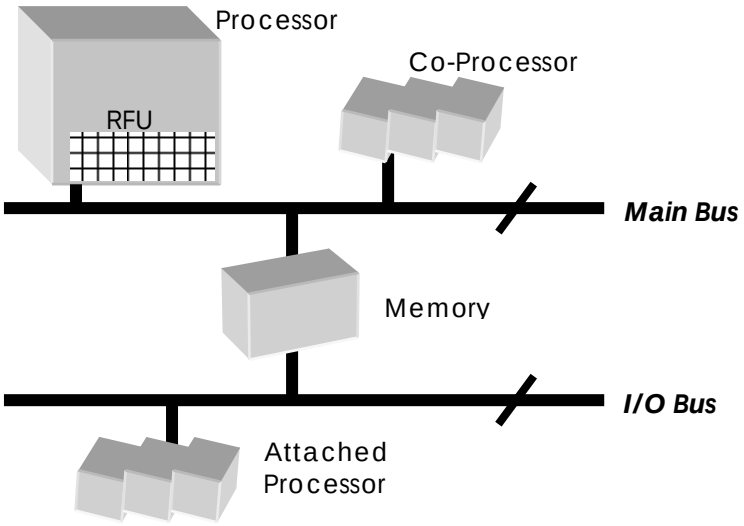


Figure 2.5: Different types of RPU Coupling

2.3.2 Granularity

The granularity of a reconfigurable unit defines its level of data manipulation: the building blocks for fine-grained logic are gates (efficient for bit level operations), while in coarse-grained RFUs the blocks are larger (therefore better suited for bit parallel operations). A fine-grain reconfigurable system consists of Processing Elements (PEs) and interconnections that are configured at bit-level. The PEs implement any 1-bit logic function and vast interconnection resources are responsible for the communication links between these PEs. Fine-grain systems provide high flexibility and can be used to implement theoretically any digital circuit. A coarse-grain reconfigurable system, in turn, consists of reconfigurable PEs that implements word-level operations and special-purpose interconnections retaining enough flexibility for mapping different applications onto the system. Usually, bit-oriented algorithms can take better benefit from fine-

grained approach, while for computation intensive applications, the coarse-grain approach can be the best alternative.

Coarse grain architectures are implemented using off the shelf functional units and multiplexers or yet using special functional units targeted to a given domain of application. Fine grain reconfigurable systems are usually implemented with FPGA. An example of an FPGA architecture is shown in Figure 2.6. It is consists of a 2-D array of Configurable Logic Blocks (CLBs) used to implement both combinational and sequential logics. Each CLB typically contains two or four identical programmable slices. Each slice usually contains two programmable cores with few inputs (typically four) that can be programmed to implement any 1-bit logic function. Programmable interconnects surround CLBs ensures the communication between them. These interconnections can be either direct connections via programmable switches or a mesh structure using Switch Boxes (S-Box), as illustrated in the example. Each S-Box contains a number of programmable switches (e.g., pass transistor) to perform the required interconnections between the input and output wires. Finally, programmable I/O cells surround the array, which are responsible for the communication with the external environment.

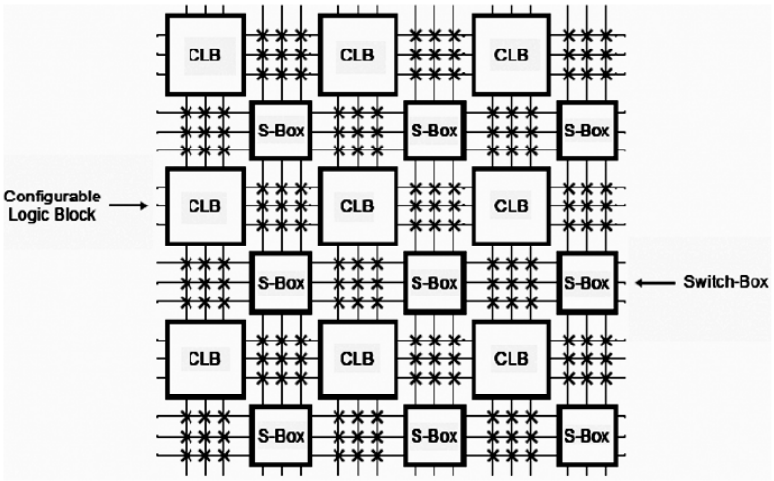


Figure 2.6: A typical FPGA architecture

Granularity also affects the size of the configuration stream and the configuration time. With fine-grained logic, more information is needed to describe the reconfigurable instruction. Coarse-grained logic descriptions are more compact, but on the other hand, some operations can be limited due to its higher level of data manipulation.

Another issue related to the granularity is the segment size. A segment is the minimum hardware unit that can be configured and assigned to a reconfigurable instruction (which will be explained in the following sub-section). Segments allow instructions to share the reconfigurable resources. If segments are used, the configuration of the reconfigurable logic can be performed in a hierarchical manner. Each instruction is assigned to one or more segments, and inside those segments, the processing elements are configured.

The interconnect that connects the elements inside a segment is referred to as intra-segment interconnect. Intersegment interconnect is used to connect different

segments. In FPGAs, there are different levels of intra-segment interconnect. With coarse-grained architectures, the interconnect tends to be done using buses and crossbar switches.

2.3.3 Instruction Types

Reconfigurable instructions are those responsible for controlling the reconfigurable hardware, as well as for the data transfer between it and the main processor. They are identified by special opcodes in the processor instruction set. Which operation a reconfigurable instruction will perform is usually specified using an extra field in the instruction word. If one considers that a list of possible operations to be executed on reconfigurable logic is encountered in a special table, this field can give two different kinds of information:

- Address: The address in the memory of the configuration data for the instruction is specified in the instruction word. Example: DISC (WIRTHLIN; HUTCHINGS, 1995).
- Instruction number: An instruction identifier of small length is embedded in the instruction word. This identifier indexes a configuration table where an information, such as the configuration data address, is stored. The number of reconfigurable instructions at one time is limited by the size of the table. Example: OneChip98 (WITTIG; CHOW, 1996).

The first approach needs more instruction word bits but has the benefit that the number of different instructions is not limited by the size of a table, as in the second case. When using the configuration table approach, the table can be changed on the fly, so the processor can adapt to the task at hand at runtime. However, specialized scheduling techniques have to be used during code generation in order to configure what instructions will be available in the table at a given moment, during program execution.

Moreover, there are other issues concerning instructions in reconfigurable systems. For example, the memory accesses performed by these instructions can be made by specialized load/store operations or implemented as stream based operations. If the memory hierarchy supports several accesses at the same time, then the number of memory ports can be greater than one. Moreover, the register file accessed by the RFU can be shared with other functional units or be dedicated (such as the floating point register file in some architectures). The dedicated register file would need less ports than if it was shared, becoming cheaper to be implemented. Its major drawback is register heterogeneity, resulting in more control for synchronizations.

Furthermore, reconfigurable instructions can be implemented as stream based ones or customized. The first type can process large amounts of data in a sequential or blocked manner. Only a small set of applications can benefit from this type, such as FIR filtering, discrete cosine transformation (DCT) etc. Custom instructions take small amounts of data at a time (usually from internal registers) and produce another small amount of data. These instructions can be used in almost all applications as they impose fewer restrictions on the characteristics of the application. Example of these operations are bit reversal, multiply accumulate (MAC) etc.

Finally, instructions can also be classified in many other ways, such as execution time, pipelining, internal state etc.

2.3.4 Reconfigurability

The reconfigurable logic inside the RFU can be programmed at different moments. If the RFU can only be programmed at startup, this unit is not reconfigurable (it is configurable). If the RFU can be configured after initialization, the supported instruction set can be bigger than the size allowed by the reconfigurable logic. If the application is divided in functionally different blocks, the RFU can be reconfigured to the needs of each individual block. In this manner, the instruction adaptation is done in a per block basis. Most of the reconfigurable processors belong to this kind.

Reconfiguration times depend on the size of the configuration data, which can be quite large. These times depend on the configuration method used. For instance, in the PRISC processor (ATHANAS; SILVERMAN, 1993), the RFU is configured by copying the configuration data directly into the configuration memory using normal load/store operations. If this task is performed by a configuration unit that is able to fetch the configuration data while the processor is executing code, a performance gain can be obtained. Furthermore, prefetching the instruction configuration data can reduce the time the processor is stalled waiting for reconfiguration, which could be done. The insertion of prefetching instructions could be done automatically by software tools.

The reconfigurable logic is simpler if the RFU is blocked during reconfiguration. However, if the RFU can be used while reconfiguring, it is possible to increase performance. This can be done, for example, by dividing the RFU in segments that can be configured independently from each other, with no necessity of reconfiguring the whole RFU at a time.

2.4 Examples

In the following subsections, some of the most cited works regarding reconfigurable systems are discussed. A special subsection is added to each architecture description, discussing briefly the behavior of the benchmark set employed for their evaluation. Later, in this same chapter, the impact of using these benchmarks will be better discussed. For even more details about existent reconfigurable architectures, some recent surveys about the theme, both on coarse (THEODORIDIS et al., 2007) (HARTENSTEIN, 2001) and fine grain (TATAS et al., 2007) systems, can be found.

2.4.1 Chimaera (1997)

Chimaera (HAUCK, 1997) was created with the claim that the current custom computing units at that time used to suffer with communication delays. Therefore, large chunks of the application code should be optimized to achieve reasonable performance improvements, so it could overlap this delay. In order to decrease communication time, this was one of the first proposals of a reconfigurable system that actually works together with the host processor, as a tightly coupled unit, with direct access to its register file.

The main component of the system is the reconfigurable array, which consists of FPGA-like logic designed to support high-performance computations. It is there that all RFU instructions will be executed. To the array, it is given direct read access to a subset of the registers in the processor (either by adding read connections to the host's register file, or by creating a shadow register file which contains copies of those registers' values).

The granularity of this array is fine, based on FPGA, but a modified one: there is no state holding elements (such as flip flops or latches) inside of it, making it totally combinational. Its routing mechanism as well as its logic blocks are illustrated in Figure 2.7a and Figure 2.7b, respectively. Furthermore, there are no pipeline stages, but there is no claim about how this would influence the system's critical path. The routing mechanism was also modified (as the way lines are structured) to allow partial reconfiguration at faster speeds. The unit accepts partial reconfiguration, although its mechanism is not clearly described on the paper.

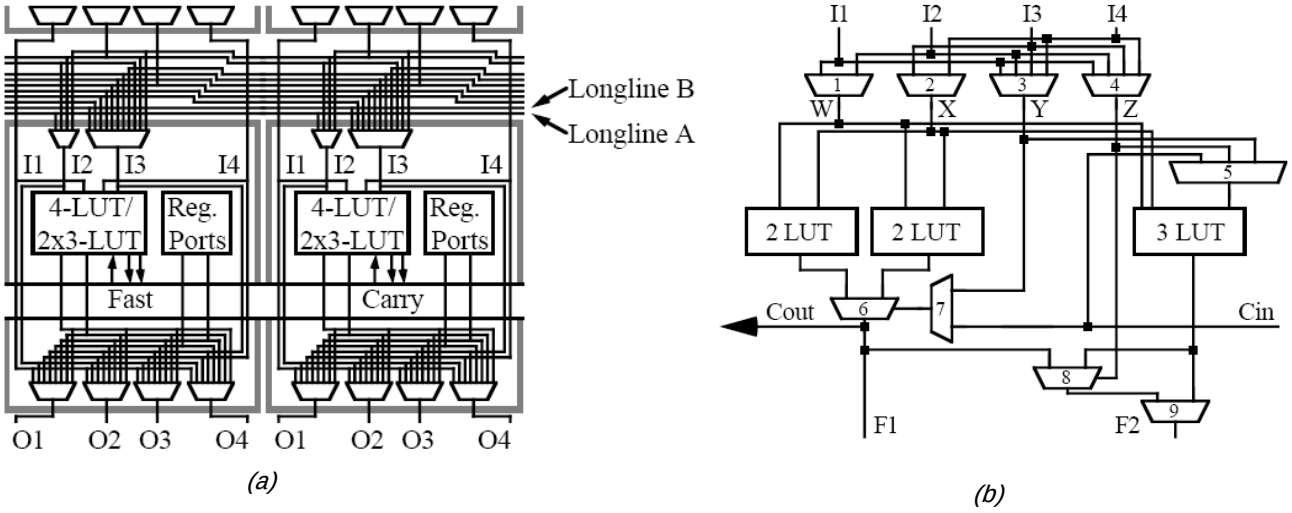


Figure 2.7: (a) The Chimaera Reconfigurable Array routing structure, and its (b) logic block (HAUCK, 1997)

Another interesting aspect of this architecture is the downward flow of information and computation through the array. There is no way to send signals back to a higher row in the system. Moreover, the system supports more than one instruction in the reconfigurable unit at the same time, treating the reconfigurable logic not as a fixed resource, but instead as a cache for RFU instructions. Those instructions that have recently been executed, or those it can otherwise predict might be needed soon, are kept in the reconfigurable logic.

The array is coupled to a MIPS R4000 processor. As part of the host processor's decode logic, it is determined if the current instruction is a RFUOP opcode. If so, it configures the RFU to produce the next result. In order to use instructions in the RFU, the application code includes calls to the RFU (using special instructions), and the corresponding RFU mappings are contained in the instruction segment of that application. These special instructions are hand-coded and manually scheduled in the original source code, which usually also suffer of some kind of transformation.

The RFU call consists of the RFUOP opcode, indicating that an RFU instruction is being called, an ID operand that determines which specific instruction should be executed, and the destination register operand. The information from which registers an RFU configuration reads its operands is intrinsic in the instruction. A single RFU instruction can use up to nine different operands. If that instruction is already present (meaning that it is already programmed, or configured) in the RFU, the result of that instruction is written to the destination register during the instruction's write back cycle. In this way, the RFU calls act just like any other instruction, fitting into the processor's

standard execution pipeline. If the requested instruction is not currently loaded into the RFU, the host processor is stalled while the RFU fetches the instruction from memory and properly reconfigures itself.

The Content Addressable Memory (CAM) determines which are the loaded instructions in the array, where they are, and if they are completed. When a RFUOP is found, and if the value in the CAM matches the RFUOP ID, the result from that row in the reconfigurable array is written onto the result bus, and thus sent back to the register file – considering that the computation is done. If the instruction corresponding to the RFUOP ID is not present, the Caching/Prefetch control logic stalls the processor, and loads the proper RFU instruction from memory into the array. The caching logic also determines which parts of the reconfigurable array are overwritten by the instruction being loaded, and attempts to retain those RFU instructions most likely to be needed in the near future. Reconfiguration is done on a per-row basis, with one or more rows making up a given RFU instruction.

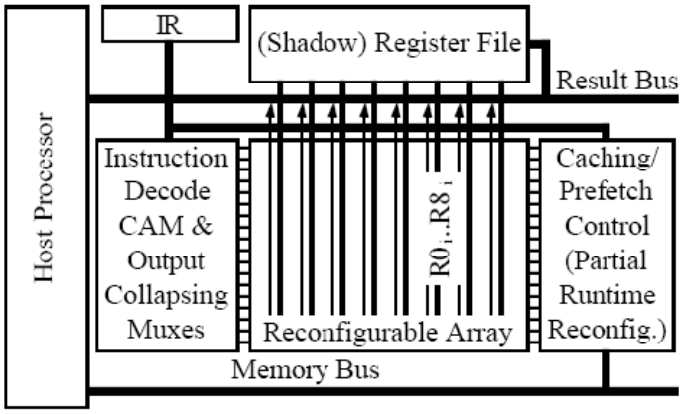


Figure 2.8: Organization of the Chimaera system (HAUCK, 1997)

BENCHMARK EVALUATION

It was used three different algorithms for the system validation:

- Compress/SPEC92 – speedup of 1.11. The small speedup can be explained because it is very likely that there are no distinct kernels for optimization. This work (LIPASTI et al., 1996), about value prediction, shows that this algorithm has a small value locality concerning loads, which could be a reflex of a small reutilization of kernels.
- Eqntott/SPEC92 – speedup of 1.8. According to the paper: "spends about 85% of its time in a single routine, 'cmppt'". In the same work cited before, Eqntott has a high degree of load value locality.
- Conway's Game of Life – According to the paper, it is basically an array computation. In the software version of the algorithm, more than half of the time is spent in the routines "get_bit" and "put_bit", which read and write the value of individual cells. By simply replacing these routines with RFU instructions, it is possible to get a speedup of 2.06. With manual modification via careful hand mapping to bit parallel, a speedup of 160 times was achieved.

2.4.2 GARP (1997)

The GARP machine is a reconfigurable system coupled to a MIPS II instruction set processor (HAUSE; WAWRYNEK, 1997). With GARP, the loading and execution of configurations in the reconfigurable array is always under the control of a program running on the main processor. As Chimaera, the reconfigurable instructions are hand-coded and statically scheduled. It is used a modified GCC-like design flow, using a pseudo language bounded together with the assembly generated from a C source.

It is interesting to point out that it uses FPGA technology for the reconfigurable logic. However, because of that, it is necessary to overcome some obstacles, such as (according to the authors) (HAUSE; WAWRYNEK, 1997):

- FPGA machines are rarely large enough to encode entire interesting programs all at once. Smaller configurations handling different pieces of a program must be swapped in over time. However, configuration time is too expensive for any configuration to be used only briefly and discarded. In real programs, much code is not repeated often enough to be worth loading into an FPGA.
- No circuit constructed with an FPGA can be as efficient as the same circuit in dedicated hardware. Standard functions like multiplications and floating-point operations are big and slow in an FPGA when compared to their counterparts in ordinary processors.
- Problems that are worth solving with FPGAs usually involve more data than can be kept in the FPGAs themselves. No standard model exists for attaching external memory to FPGAs. FPGA-based machines typically include ad hoc memory systems, designed specifically for the first application envisaged for the machine.
- Wide acceptance in the marketplace requires binary compatibility among a range of implementations. The current crop of FPGAs, on the other hand, must be reprogrammed for each new chip version, even within the same FPGA family.

Garp's reconfigurable array is composed of entities called *blocks* (Figure 2.9). One block on each row is known as a *control block*. The rest of the blocks in the array are *logic blocks*, which correspond roughly to the *CLBs* of the Xilinx 4000 series (XILINX, 2008). The Garp architecture fixes the number of columns of blocks at 24. The number of rows is implementation-specific, but can be expected to be at least 32. The basic "quantum" of data within the array is 2 bits. Logic blocks operate on values as 2-bit units, and all wires are arranged in pairs to transmit 2-bit quantities. This way, operations on 32-bit quantities generally require 16 logic blocks. Compared to typical FPGAs, Garp expends more hardware on accelerating operations like *adds* and variable *shifts*. The decision to make everything 2 bits wide is based on the assumption that a large fraction of most configurations will be taken up by multi-bit operations that are configured identically for each bit. By doubling up bits, the size of configurations— and thus the time required to load configurations and the space taken up on the die to store them—is reduced at the cost of some loss of flexibility

Rather than specify component delays as precise times that would change with each processor generation, delays in Garp are defined in terms of the sequences that can be fit within each array clock cycle. Only three sequences are permitted:

- short wire, simple function, short wire, simple function;
- long wire, any function not using the carry chain; or
- short wire, any function.

The loading and execution of configurations is under control of the main processor. Several instructions have been added to the MIPS-II instruction set for this purpose, including ones that allow the processor to move data between the array and the processor's own registers. The main processor has a number of instructions for controlling the array. These include instructions for loading configurations, for copying data between the array and the processor registers, for manipulating the array clock counter, and for saving and restoring array state on context switches. The Garp reconfigurable hardware can access directly the main memory system, in opposite to the Chimaera architecture.

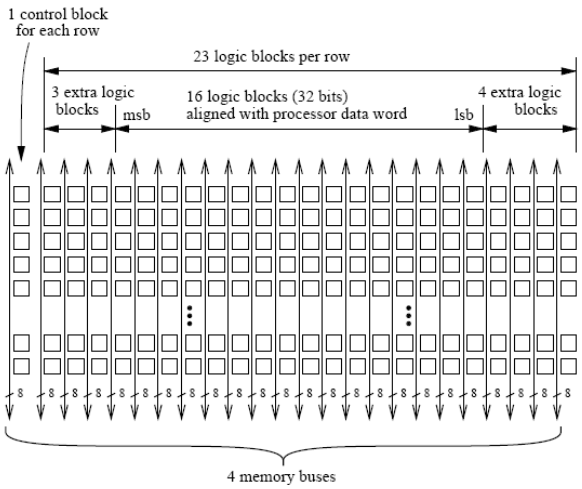


Figure 2.9: A block of the GARP machine (HAUSE; WAWRYNEK, 1997)

Each block in the array requires exactly 64 configuration bits (8 bytes) to specify the sources of inputs, the function of the block, and any wires driven with outputs. No configuration bits are needed for the array wires. A configuration of 32 rows requires approximately 6 KB. Assuming a 128-bit path to external memory, loading a full 32-row configuration takes 384 sequential memory accesses. At that time, a typical processor external bus might need 50 μs to complete the load.

Since not all useful configurations will require the entire resources of the array, Garp allows partial array configurations. The smallest configuration is one row, and every configuration must fill exactly some number of contiguous rows. Distributed within the array is a cache of recently used configurations, similar to an ordinary instruction cache. Two configurations can never be active at the same time, no matter how many array rows might be left unused by a small configuration.

BENCHMARK EVALUATION

Simulations were performed in order to gather results for Garp, since at that time no actual hardware implementation existed. It was compared against a Sun UltraSPARC

1/170, a 4-way superscalar 64-bit processor with 16 kB each of on-chip instruction and data caches. Performance estimations can be observed in Figure 2.10. In Figure 2.11 one can observe the area estimative of a hypothetical implementation in hardware of this reconfigurable system. It would be implemented in a 0.5 µm, 4-metal-layer process in a die size of 17.5 x 17.8 mm2. It is also compared with the same UltraSPARC.

Benchmark	167 MHz SPARC	133 MHz Garp	ratio
DES encrypt of 1 MB	3.60 s	0.15 s	24
Dither of 640 × 480 image	160 ms	17 ms	9.4
Sort of 1 million records	1.44 s	0.67 s	2.1

Figure 2.10: Performance estimations for GARP machine, compared to the SPARC (HAUSE; WAWRYNEK, 1997)

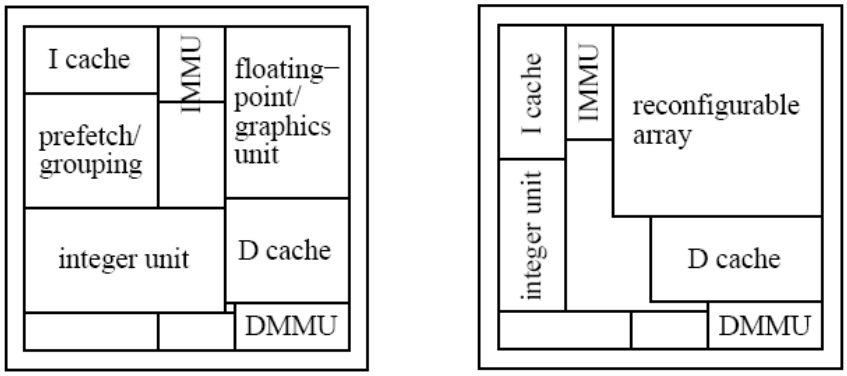


Figure 2.11: Area estimation for the GARP system (HAUSE; WAWRYNEK, 1997)

This system was evaluated with the following algorithms:

- DES - A well-know cryptography algorithm, with just one hot spot responsible for almost 100% of execution time (ANANIAN, 1997).
- Sorting – Several kinds of sorting, including Quicksort (1 million objects). Sorting algorithms usually have just one kernel used for sorting the components, which is repeated several times.
- Image Dithering – It is a vector processing. Again, it is based on the same kernel that is repeated several times through the image. In this case, a dither was applied to a full color image of 640x480 pixels to a fixed palette of fewer than 256 colors.

2.4.3 Remarc (1998)

REMARC comes from “Reconfigurable Multimedia Array Coprocessor” (MIYAMORI et al., 1998). It is a reconfigurable unit, coupled to a MIPS II ISA based RISC machine. As the name states, REMARC was specifically designed to speed up multimedia applications. As the MIPS ISA can support up to four coprocessors, and coprocessor 0 is already used for memory management and exception handling, coprocessor 1 is used for a floating point unit; REMARC operates as coprocessor 2.

A coarse grain reconfigurable system was employed, because, according to the authors, fine grain FPGA based reconfigurable architectures have the following drawbacks:

- The small width of the programmable logic blocks results in large area and delay overheads to implement wider datapaths, such as 8 or 16 bits long.
- FPGAs are slower than a custom integrated circuit and have lower logic density.

This way, REMARC consists of an 8x8 array of nano processors and a global control unit. The nano processor can communicate to the four adjacent ones through the dedicated connections and to the processors in the same row and the same column through the 32-bit Horizontal Bus (HBUS) and the 32-bit Vertical Bus (VBUS), respectivelly. A general overview of the system can be observed in Figure 2.12.

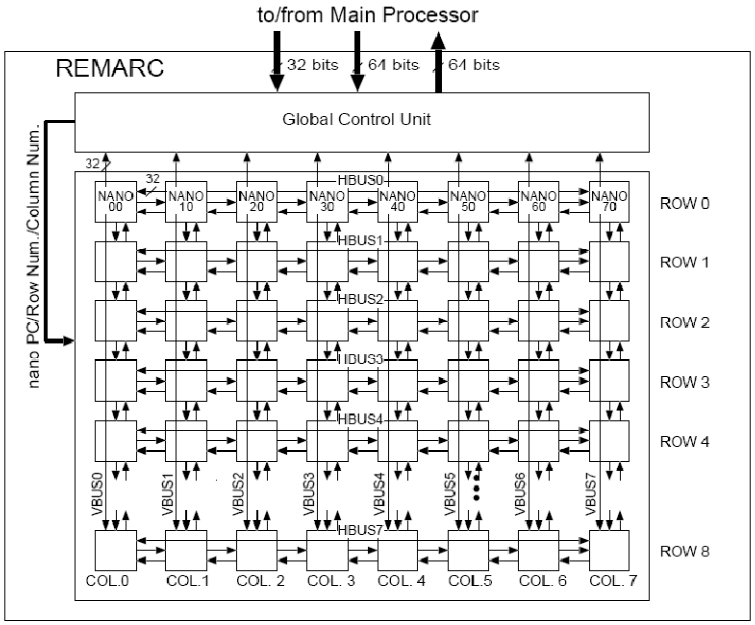


Figure 2.12: General overview of the REMARC reconfigurable system (MIYAMORI et al., 1998)

The nano processor consists of a 32-entry instruction RAM (nano instruction RAM), a 16-bit ALU, a 16-bit entry data RAM, an instruction register (IR), eight 16- bit data registers (DR), four 16-bit data input registers (DIR), and a 16-bit data output register (DOR). The DOR registers are used to accept data from the four adjacent nano processors (up, down, left, and right) through dedicated connections (DINU, DIND, DINL, and DINR). The DOR register data can also be used for source data of ALU operations or data inputs of a DIR register. These local connections provide high bandwidth pathways within the processor array. The 16-bit ALU can execute 30 different instructions. The nano processor is demonstrated in Figure 2.3.

The nano processors do not have Program Counters (PCs) by themselves. The global control unit generates the PC value (nano PC) for all nano processors every cycle. All nano processors use the same nano PC and execute the instruction indexed by it. However, each nano processor has its own nano instruction RAM. Therefore, each nano processor can operate differently according to the nano instructions stored in this

local RAM. This makes it possible to achieve a limited form of Multiple Instruction Stream, Multiple Data Stream (MIMD) operation in the processor array. At this point, according to the authors, REMARC can be regarded as a VLIW processor in which each instruction consists of 64 operations.

As already stated before, the global control unit controls the nano processors and the transfer of data between them and the main processor. It includes a 1024-entry instruction RAM (global instruction RAM), data registers, and control registers. These registers can be accessed by the main processor directly using main processor instructions: move from/to coprocessor or load/store coprocessor. Eight 32-bit VBUSs are used for communication between the global control unit and the nano processors.

The reconfigurable instructions are programmed in the special REMARC assembly language, and can be added to a regular C code using GCC.

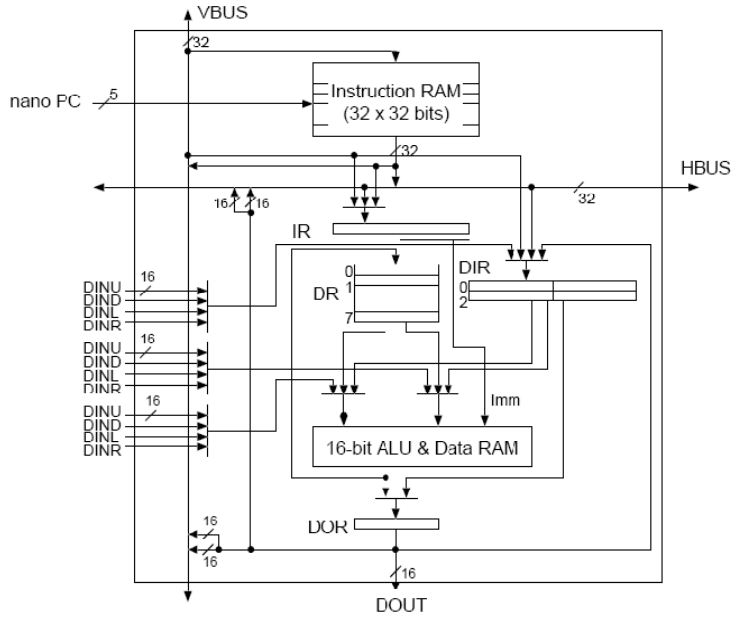


Figure 2.13: One nano processor in the REMARC system (MIYAMORI et al., 1998)

BENCHMARK EVALUATION

Remarc executes MPEG2 decoding, optimizing just two kernels: IDCT and MC that, according to the paper, cover more than 70% of the total execution time. It also executes MPEG2 encoding, optimizing just Motion Estimation, which covers 98% of total execution time, as shown in Figure 2.14. The third algorithm employed is DES, already discussed in this same section, which can be observed in Figure 2.15. A high-level simulation of the system demonstrated speedups ranging from a factor of 2.3 to 21.2 in the aforementioned applications.

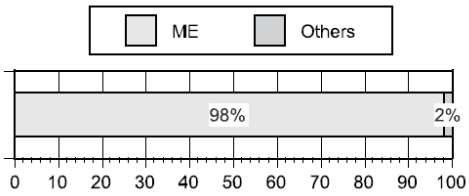


Figure 2.14: Motion Estimation is responsible for 98% of execution time in the MPEG2 encoder (MIYAMORI et al., 1998)

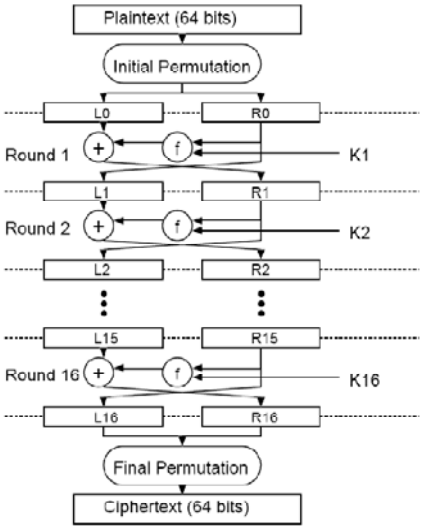


Figure 2.15: Steps of the DES algorithm (MIYAMORI et al., 1998)

2.4.4 Rapid (1998)

RaPiD (CRONQUIST et al., 1998) is a coarse-grain architecture that allows the dynamic construction of deeply pipelined computational datapaths from a mix of ALUs, multipliers, registers and local memories. The goal of RaPiD is to compile regular computations like those found in DSP applications into both an application-specific datapath, and the program for controlling that datapath. RaPiD-I is a linear array of functional units which can be configured to form a (mostly) linear computational pipeline. This array of functional units is divided into identical cells. One cell for RaPiD-I is shown in Figure 2.16. This cell comprises an integer multiplier, two integer ALUs, six general-purpose registers and three small local memories. The complete RaPiD-I array contains 16 of these cells.

The functional units are interconnected using a set of ten segmented busses that run the length of the datapath. Each input of the functional units is attached to a multiplexer that is configured to select one of eight busses. Each output of the functional units is attached to a demultiplexer comprised of tristate drivers, each driving one of eight busses. Each output driver can be configured independently, which allows an output to fan out to several busses, or none at all if the functional unit is not being used. The ALUs perform the usual logical and arithmetic operations on signed or unsigned fixed-point 16-bit data. The two ALUs in a cell can be combined to perform a

pipelined 32-bit operation, most typically as a 32-bit adder for multiply-accumulate computations.

RaPiD is programmed for a particular application by first mapping the computation onto a datapath pipeline. The control signals are divided into static control signals provided by configuration memory, and dynamic control which must be provided on every cycle. The static programming bits are used to construct this pipeline and the dynamic programming bits are used to schedule the operations of the computation onto the datapath over time. A controller is programmed to generate the information needed to produce the dynamic programming bits. At the time the paper was written, the applications were mapped to the RaPiD architecture by hand. There was no compiler or tool support at all.

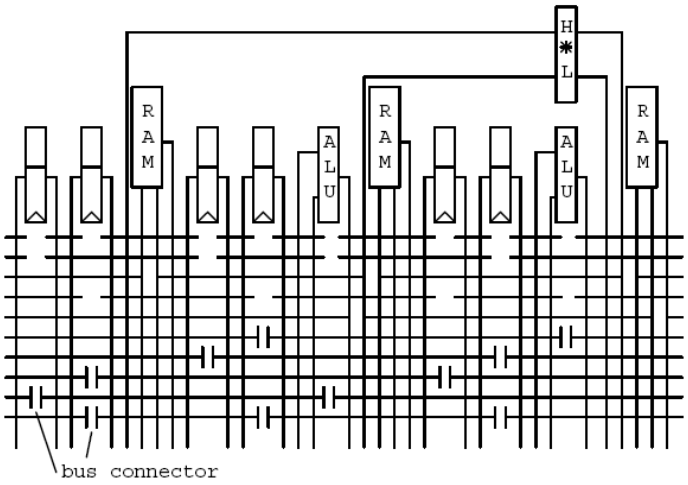


Figure 2.16: RaPiD-I cell (CRONQUIST et al., 1998)

BENCHMARK EVALUATION

RaPiD executes two algorithms that have already been discussed on this section: FIR filter and Matrix multiply, proving once more that traditional reconfigurable architectures in general just attack one niche of applications. A performance of up 1.6 billion of operations per second was achieved in a FIR filter and Matrix multiplication. However, there is no comparison against any other architecture.

2.4.5 Piperench (1999)

The basic principle of Piperench (GOLDSTEIN et al., 1999) is the so-called “pipelined reconfiguration”. It means that a given kernel is broken into pieces, and these pieces can be reconfigured and executed on demand. This way, the parts of a given kernel are multiplexed in time and space into the reconfigurable logic. This process is called virtualization process, and it is illustrated in Figure 2.17. In the upper part of it (Figure 2.17a), it is demonstrated an application which was divided in 5 different pipeline stages, taking the total of 7 cycles to be configured and executed (each stage can be configured and used independently of each other), representing the normal operation. Figure 2.17b shows how this application can fit in the reconfigurable hardware after virtualization: just 3 stages of the equivalent pipeline stages presented

before are necessary. The pipeline stages are reconfigured on demand, according to the kernel needs. Note that the virtual stage 1 is used to execute the equivalent of stages 1 and 4 of the original operation. This is feasible because it is done in different periods of time. Since some stages are configured while others are executed, reconfiguration does not decrease performance. Consequently, it is possible to execute the same piece of software taking the same time, but with a smaller area overhead.

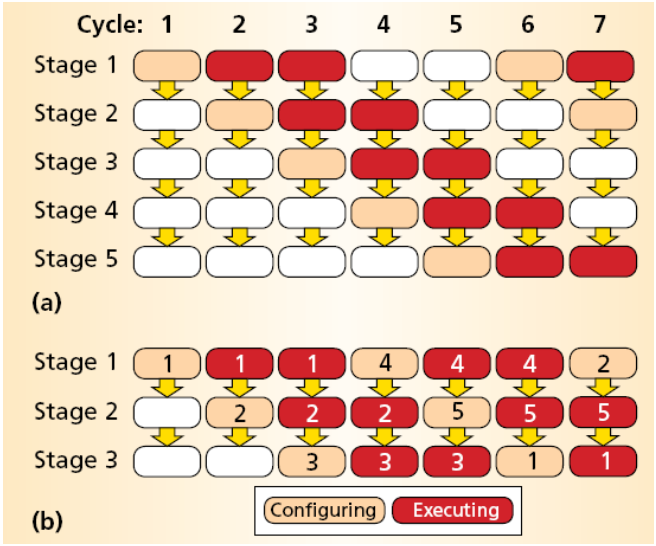


Figure 2.17: The virtualization process, technique used by Piperench. (a) Normal execution. (b) With virtualization (GOLDSTEIN et al., 1999)

In its current implementation, PipeRench can be classified as an attached processor. Figure 2.18 presents a general overview of the PipeRench architecture. A set of physical pipeline stages are called *stripes*. Each stripe has an interconnection network and a set of Processing Elements (PEs). In Figure 2.19 one can observe a more detailed view of a PE. Each PE contains an arithmetic logic unit and a pass register file. Each ALU in the PE contains lookup tables (LUTs) and extra circuitry for carry chains, zero detection, and so on. Designers can implement combinational logic using a set of *NB*-bit-wide ALUs. They can also cascade the carry lines of these ALUs to construct wider ALUs by chaining them together via the interconnection network, so it is possible to build complex combinational functions. The ALU operation is static while a particular virtual stripe resides in a physical stripe.

Through the interconnection network, PEs can access operands from registered outputs of the previous stripe, as well as registered or unregistered outputs of the other PEs in the same stripe. The pass register file provides a pipelined interconnection from a PE in one stripe to the corresponding PE in subsequent stripes. A program can write the ALU's output to any of the *P* registers in the pass register file. If the ALU does not write to a particular register, that register's value will come from the value in the previous stripe's corresponding pass register. For data values to move laterally within a stripe, they must use the interconnection network. In each stripe, the interconnection network accepts inputs from each PE in that stripe, plus one of the register values from the previous stripe. Moreover, a barrel shifter in each PE shifts its inputs $B - 1$ bits to the left. Thus, PipeRench can handle the data alignments necessary for word-based arithmetic. The PEs can also access global I/O buses. These buses are necessary because an application's pipeline stages may physically reside in any of the fabric's stripes.

Inputs to and outputs from the application must use a global bus to get to their destination. Because of hardware virtualization constraints, the buses cannot be used to connect consecutive stripes.

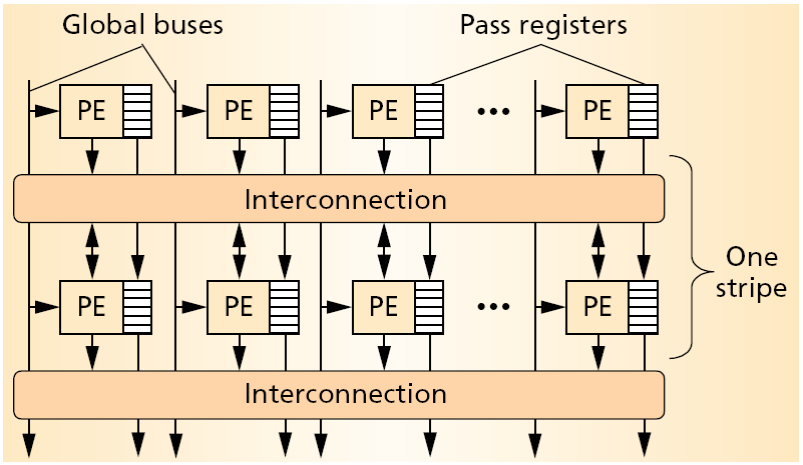


Figure 2.18: General overview of the Piperench structure (GOLDSTEIN et al., 1999)

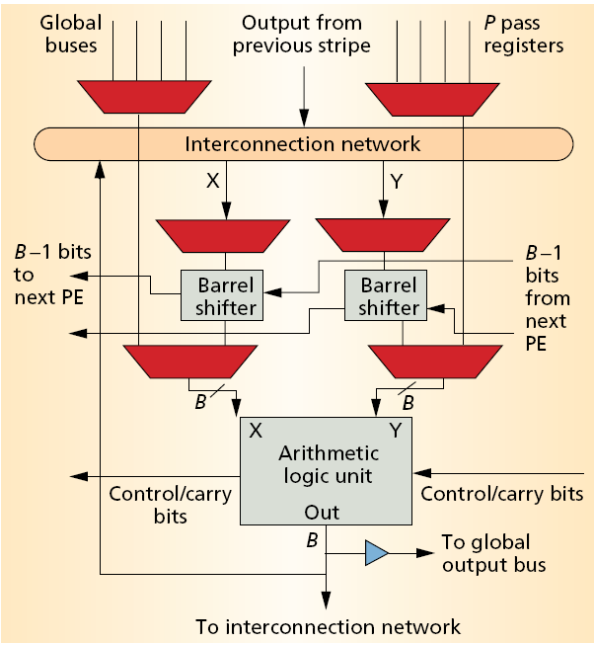


Figure 2.19: Detailed view of the Process Element and its connections

The process of code generation uses a parameterized compiler. The compiler begins by reading a description of the architecture. This description includes the number of PEs per stripe, each PE's bit width, the number of pass registers per PE, the interconnection topology, PE delay characteristics, and so on. The source language is a dataflow intermediate language. After parsing, the compiler inlines all modules, unrolls all loops, and generates a straight-line, single-assignment program.

It interesting to repeat some of the reasons that motivated the authors to build this architecture without using FPGA:

- Logic granularity: It is claimed that FPGAs are designed for logic replacement. The granularity of the functional units is optimized to replace random logic, not to perform multimedia computations.
- Configuration time: The time to load a configuration in the fabric ranges from hundreds of microseconds to hundreds of milliseconds. For FPGAs to improve processing speed over that of a general-purpose processor, they must amortize this start-up latency over huge data sets, limiting their applicability.
- Forward compatibility. FPGAs require redesign or recompilation to benefit from future chip generations.
- Hard constraints. FPGAs can implement only kernels of a fixed and relatively small size. This size restriction makes compilation difficult and causes large, unpredictable discontinuities between kernel size and performance.
- Compilation time. A kernel's synthesis, placement, and routing design phases take hundreds of seconds, taking longer than the compilation of the same kernel for a general-purpose processor.

BENCHMARK EVALUATION

To evaluate PipeRench's performance, the authors have also chosen dataflow oriented software with very distinct kernels, which is a characteristic of algorithms that are highly based on filters or transforms, as can be observed:

- Automatic target recognition (ATR): it implements the shape-sum kernel of the Sandia algorithm for automatic target recognition;
- Cordic: it implements the Honeywell timing benchmark for Cordic vector rotations;
- DCT: it is a 1D, 8-point discrete cosine transform;
- DCT-2D: it is a 2D discrete cosine transform;
- FIR: it is a finite-impulse response filter with 20 taps and 8-bit coefficients;
- IDEA: it implements a complete 8-round International Data Encryption Algorithm;
- Nqueens: it is an evaluator for the N queens problem on an 8 x 8 board;
- Over: it implements the Porter-Duff over operator;
- PopCount: it is a custom instruction implementing a population count instruction;
- IDEA: it is a block cipher.

Results for the Piperench system can be seen in Figure 2.20. This figure shows the performance improvements of the 100MHz Piperench, built as a 128-bit-wide fabric having 8-bits PEs with 8 registers each, over a 300MHz Ultrasparc II.

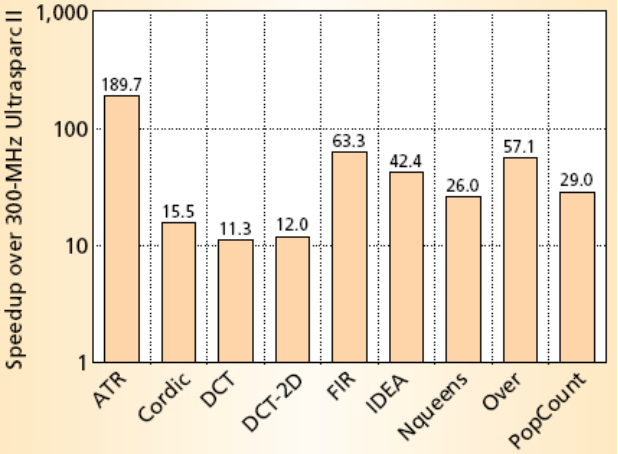


Figure 2.20: Performance improvements over a 300-mhz Ultrasparc II

2.4.6 Molen (2001)

The Molen processor is a FPGA based reconfigurable system with a loosely coupled reconfigurable array. The two main components in the Molen organization (VASSILIADIS et al., 2001) are depicted in Figure 2.21. More precisely, they are the *Core Processor*, which is a GPP, and the *Reconfigurable Unit* (RU). The Arbiter issues instructions to both processors; and data transfers are controlled by the *Memory MUX*. The reconfigurable unit (RU), in turn, is subdivided into the *pμ-code unit* and the *Custom Computing Unit* (CCU). The CCU is implemented in reconfigurable hardware, e.g., a field-programmable gate array (FPGA), and memory. The application code runs on the GPP except of the accelerated parts implemented on the CCU used to speed up the overall program execution. Exchange of data between the main and the reconfigurable processors is performed via the *exchange registers* (XREGs).

The reconfigurable processor operation is divided into two distinct phases: *set* and *execute*. In the set phase, the CCU is configured to perform the targeted operations. Subsequently, in the execute phase, the actual execution of the operations takes place. Such decoupling allows the set phase to be scheduled well ahead of the execute phase, thereby hiding the reconfiguration latency. As no actual execution is performed in the set phase, it can even be scheduled upward across the code boundary in the instructions preceding the RU targeted code.

A sequential consistency programming paradigm is used for MOLEN (VASSILIADIS et al., 2003). It requires only a one-time architectural extension of a few instructions that supports a large user reconfigurable operation space. Although the complete ISA extension comprises 8 instructions, the minimal instruction set (*πISA*) of the pμ-code unit is enough to provide a working scenario. The instructions in this class are: *set*, *execute*, *movtx* and *movfx*. By implementing the first two instructions (*set/execute*), any suitable CCU implementation can be configured and executed in the CCU space. The *movtx* and *movfx* instructions are needed to provide the input/output interface between the RU targeted code and the remaining application code to pass data, parameters or data references.

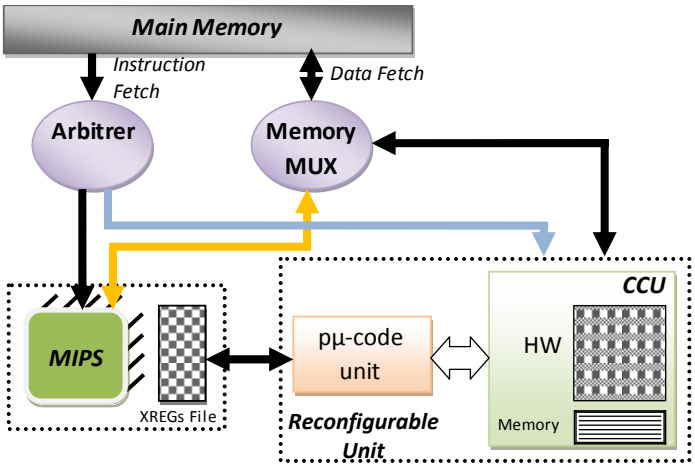


Figure 2.21: A general overview of the Molen System

BENCHMARK EVALUATION

Molen was evaluated with the MPEG2 encoder/decoder. The most time consuming operations among SAD (sum of absolute difference), 2D-DCT (two dimensional discrete cosine transform), and 2D-IDCT (two dimensional inverse DCT) were optimized. These kernels, in turn, are the most time consuming ones in the MPEG2 algorithm and, as already discussed before, highly dataflow oriented. Figure 2.22 demonstrates the impact of implementing these kernels as Molen hardware when comparing against a PowerPC processor without it. Columns labeled "theory" present the theoretically achievable maximum speed up. Columns labeled with "impl." contain data for the projected speedups with respect to the considered Molen implementation.

	MPEG2 encoder			MPEG2 decoder		
	theory	impl.	impl./th.	theory	impl.	impl./th.
carphone	2.85	2.64	93%	2.02	1.94	96%
claire	2.99	2.80	94%	1.60	1.56	98%
container	3.12	2.96	95%	1.68	1.63	97%
tennis	3.37	3.18	94%	1.68	1.65	98%

Figure 2.22: Molen Speed ups (VASSILIADIS et al., 2004)

2.4.7 Other Reconfigurable Architectures

Other processors are worth to be briefly discussed in this section. ConCISe (RAZDAN; SMITH, 1994) has a tightly coupled reconfigurable array in the processor core, limited to combinational logic – in the same way Chimaera was implemented. The array is, in fact, an additional functional unit in the processor pipeline, sharing the same resources of the other ones. As more examples, some designs employ standard fine-grained FPGA resources, such as DISC (WIRTHLIN; HUTCHINGS, 1995), OneChip (WITTG; CHOW, 1996), PRISM-I (ATHANAS; SILVERMAN, 1993), PRISM-II (WAZLOWSKI et al., 1993). In the group of coarse grain reconfigurable systems, one

can include: Pact-XPP (CARDOSO et al., 2002), Morphosys (SINGH et al., 1998), Pleiades (ZHANG et al., 1998) and ADRES (MEI et al., 2003). Other reconfigurable architectures with smaller impact on the scientific community also have been implemented, such as: Motium (HEYSTERS et al., 2003), XiRISC (LODI et al., 2003) and ReRISC (VASSILIADIS et al, 2006).

Table 2.1, (BARAT; LAUWEREINS, 2000) shows some of the most popular reconfigurable architectures, demonstrating their different aspects and characteristics. However, in opposite to what is stated on this table, the Chimaera reconfigurable unit can take more than one cycle to execute its instructions, and it was implemented together with a MIPS R4000 processor. It is very interesting to point out that the only architecture presented on this table which is not fine grained is the newest one: Piperench. That is because coarse grain reconfigurable architectures started to become popular after the year of 2000.

Table 2.1: General characteristics of several reconfigurable architectures (BARAT; LAUWEREINS, 2000)

	PRISM-I [6]	PRISM-II [7]	Nano Processor [4]	PRISC [8]	DISC [5]	OneChip [10]	Garp [3]	Chimaera [9]	NAPA [12]	OneChip98 [11]	PipeRench [13]
Year	1993	1993	1994	1994	1995	1996	1997	1997	1998	1999	1999
Processor Type	M68010	Am29050	RISC?	R2000	CISC?	DLX	MIPS	?	RISC	S-DLX	?
Application Type	GP	GP	GP	GP	GP	GP	GP	Multimedia	GP	GP	Multimedia
Coupling	Attach	Copro	RFU	RFU	RFU	RFU	Copro	RFU	Copro	RFU	Attach
Inst. Types	All	All	Custom	Custom	All	All		Custom		Stream	Stream
Duration	Var.	Var.	Fixed	Fixed	Var.	Var.	Var.	Fixed	Var.	Var.	Var.
Inst. Coding			Fixed	Fixed	Fixed	Fixed	Fixed	Fixed		Fixed	
Configuration Table			No	No	Yes	No	Yes	Yes		Yes	
Operand Coding			Fixed	Fixed	Wired	Fixed	Fixed	Wired		Fixed	
Shared Register File			Yes	Yes	Yes	Yes	Yes	Yes		Yes	
Memory Ports				No	Yes	Yes	Yes	No	Yes (2)	Yes (1)	Yes
Granularity	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Coarse
Reconfigurable	No	No	No	Yes	Partial	No	Yes	Yes	Yes	Yes	Yes
Configuration Method				SW	SW		SW	Ctlr.	Ctlr.	Ctlr.	
RFU Blocked during Reconfiguration				Yes	Yes		Yes	No		No	No
RPU Segmented				No	Yes		Yes	Yes	Yes	No	Yes
Can Support Prefetching?				No	No		No	?		Yes	
Relocatable Hardware				No	Yes		Yes	Yes	Yes	No	Yes
Instruction Caching				No	Yes		Yes	Yes		Yes	

2.4.8 Recent Dataflow Architectures

More recently, new dataflow architectures were proposed. These architectures differ from regular reconfigurable systems mainly because they do not have any kind of processor working together with it. Moreover, they abandon program counter and the linear von-Neumann execution that could limit the amount of parallelism to be explored. However, they are highly dependent on compilers and tools to code generation, which involves placing parts of the code in the correct order in the processing elements, for the synchronism, parallelism analysis etc. This way, the main effort is on the development of these compilers and tools, not on the hardware design, which is usually very simple.

As a first example, TRIPS (SANKARALINGAM et al., 2003) is a hybrid von-Neumann/dataflow architecture that combines an instance of coarse-grained, polymorphous grid processor cores with an adaptive on-chip memory system. To better explore the application parallelism and provide a large use of available resources, TRIPS uses three different modes of execution: D-morph that search parallelism in

instruction level; T-morph that works at the thread level, mapping multiple threads onto a single TRIPS core; and S-morph that is targeted to applications like streaming media with high data-level parallelism. Figure 2.23 gives an overview of the TRIPS architecture.

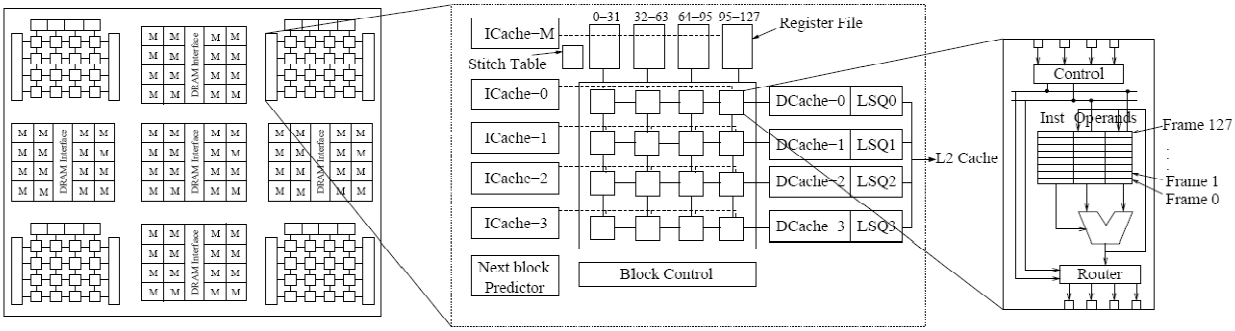


Figure 2.23: General overview of the TRIPS architecture. From left to right: the TRIPS Chip, TRIPS core, and an execution node (SANKARALINGAM et al., 2003)

Another example of a dataflow machine is Wavescalar (SWANSON et al., 2003) that, likewise TRIPS, relies on the compiler to statically place instructions into its hardware structures. Another similarity is that there is no central processing unit at all, which is replaced by many processing nodes. As it can be observed in Figure 2.24, the basic processing element is very similar to the one found in TRIPS. However, this architecture is even more regular when considering its structure.

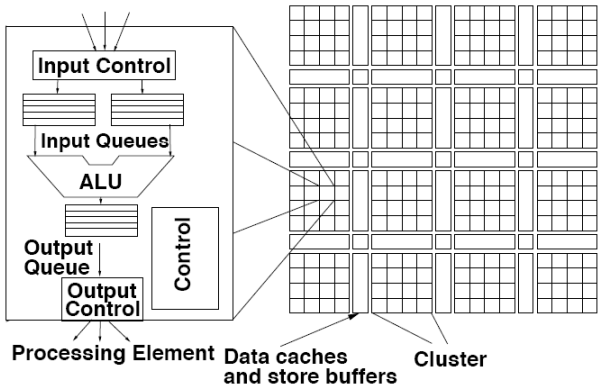


Figure 2.24: The Wavescalar architecture (SWANSON et al., 2003)

In the same work, the motivations of building a dataflow architecture are discussed. These motivations are related to some limitations that superscalar processors present, mainly because they are basically Von-Neumann architectures. The first thing discussed is the so-called *processor scaling wall*, which emerges because of three reasons:

- The difference in terms of speed between (fast) transistors and (slow) wires is increasing – meaning that there is a disparity between computation and communication;
- The increasing cost of circuit complexity;
- The decreasing of reliability of these circuits.

According to the authors, superscalar processors will suffer a lot because of these reasons, since they have a huge infrastructure with slow broadcast networks, associative searches, complex control logic and inherently centralized structures. Moreover, other drawbacks regarding superscalar architecture can be cited:

- Their inherent complexity makes efficient implementation a daunting challenge,
- They ignore an important source of locality in instruction streams,
- Their execution model centers around instruction fetch, an intrinsic serialization point.

On the other hand, dataflow machines must convert control dependencies into data dependencies. To accomplish this, they explicitly send data values to the instructions that need them instead of broadcasting them via the register file. The potential consumers are known at compile time, but depending on control flow, only a subset of them should receive the values at run-time.

2.5 Directions

In this sub-section, some directions that should be taken while developing a new reconfigurable architecture are analyzed. First, we evaluate a known benchmark set in order to figure what is the best strategy to take in terms of granularity. Then, we study the impact of this analysis in both fine and coarse grain reconfigurable systems performing high levels simulations. Finally, other issues are taken into account, such as reconfiguration and execution times, and the growing number of applications being executed at the same time on a system.

2.5.1 Heterogeneous Behavior of the Applications

In (BECK et al., 2008b), it is used a subset of the Mibench Benchmark Suite (GUTHAUS et al., 2001), which represents the complete set of diverse algorithm behaviors. As a matter of fact, this suite has been chosen because, according to (GUTHAUS et al., 2001), it has a larger range of different behaviors when compared against other benchmark sets, e.g. SPEC2000 (HENNING, 2000). This way, the following 18 benchmarks were evaluated: *Quicksort*, *Susan Corners/Edges/Smoothing*, *Jpeg Encoder/Decoder*, *Dijkstra*, *Patricia*, *StringSearch*, *Rinjdael Encode/Decode*, *Sha*, *Raw Audio Coder/Decoder*, *GSM Coder/Decoder*, *Bitcount* and *CRC32*.

First, a characterization of the algorithms regarding the number of instructions executed per branch is done (classifying them as control or dataflow oriented based on these numbers). As it can be observed in Figure 2.25, the *RawAudio Decoder* algorithm is the most control flow oriented one (a high percentage of branches executed per program) while the *Rijndael Encoder* is quite the opposite. It is important to point out that, for reconfigurable architectures, the more instructions a basic block has, the better, since there is more room for exploiting parallelism. Furthermore, more branches mean additional paths that can be taken, increasing the execution time and the area consumed by a given configuration, when implemented in reconfigurable logic.

Figure 2.26 shows the analysis of distinct kernels based on the execution rates of the basic blocks in the programs. The methodology involves investigating the number of basic blocks responsible for covering a certain percentage of the total number of basic block executed. For instance, in the *CRC32* algorithm, just 3 basic blocks are

responsible for almost 100% of the total program execution time. Again, for typical reconfigurable systems, this algorithm can be easily optimized: one just needs to concentrate all the design effort on that specific group of basic blocks and implement them to reconfigurable logic.

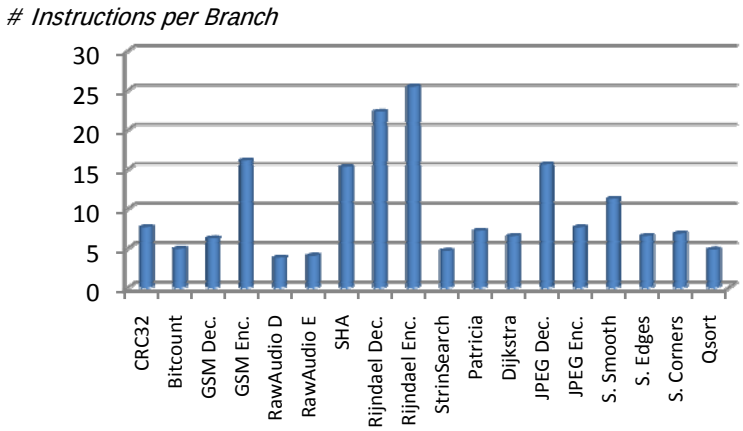


Figure 2.25: Instruction per Branch Rate

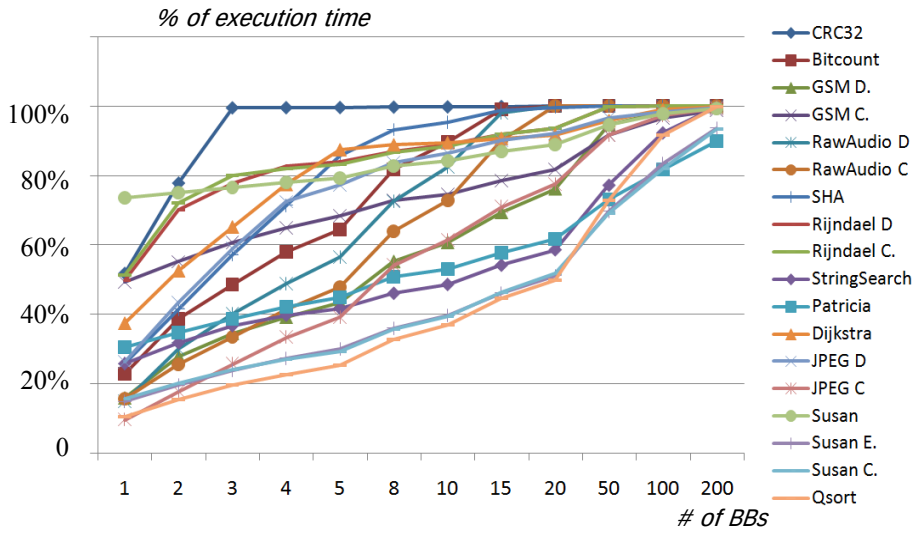


Figure 2.26: How many BBs are necessary to cover a certain amount of execution time?

However, other algorithms, such as the widely used *JPEG decoder*, have no distinct execution kernels at all. In this algorithm, 50% of the total instructions executed are due to 20 different BBs. Hence, if one wished to have a speedup of 2x (according to Amdahl's law), considering ideal assumptions, all 20 different basic blocks should be mapped into reconfigurable logic. This analysis will be presented in more details in the next section.

The problem of not having a clear group of most executed kernels becomes even more evident if one considers the wide range of applications that embedded systems are implementing nowadays. In a scenario when an embedded system runs *RawAudio decoder*, *JPEG encoder/decoder*, and *StringSearch*, the designer would have to transform approximately 45 different basic blocks into the reconfigurable fabric to achieve a maximum of 2 times performance improvement.

Furthermore, it is interesting to point out that the algorithms with a high number of instructions per branch tend to be the ones that need fewer kernels to achieve higher speedups. Figure 2.27 illustrates this scenario by using the cases with 1, 3 and 5 basic blocks. Note that, mainly when it is considered the most executed basic block only (first bar of each benchmark), the shape of the graph is very similar to the instructions per branch ratios shown in Figure 2.25 (with some exceptions, such as the *CRC32* or *JPEG decoder* algorithms). A deeper study about this issue is envisioned to indicate some directions regarding the reconfigurable arrays optimization just based on very simple profile statistics.

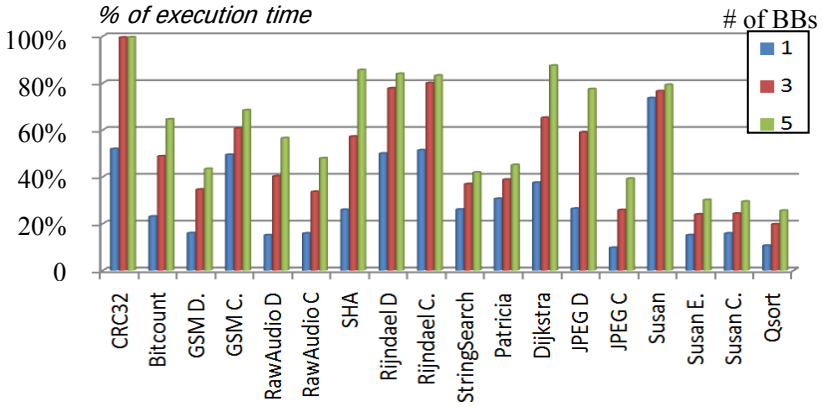


Figure 2.27: Amount of execution time covered by 1, 3 or 5 basic blocks in each application

2.5.2 Potential of using Fine Grained Reconfigurable Arrays

In this section, the potentiality of fine grain reconfigurable arrays is evaluated. Considering the optimization of loops and subroutines, the level of performance gains if a determined number of hot spots is mapped to a fine grain reconfigurable logic is analyzed. In this first experiment, it is assumed that just one piece of reconfigurable hardware is available per loop or subroutine. This means that the only part of the code that will be optimized by the reconfigurable logic is the one which is common in all iterations. For example, let us assume that a loop should be executed 50 times. 100% of the code is executed 49 times, but just 20% is executed 50 times (all the iterations). This way, just this 20% is available for optimization, since it comprises the common instructions executed in all loop iterations. Figure 2.28 illustrates this case. The dark part is always executed, so just this part can be transformed to reconfigurable logic. Moreover, subroutines that are called inside loops are not suited for optimization.

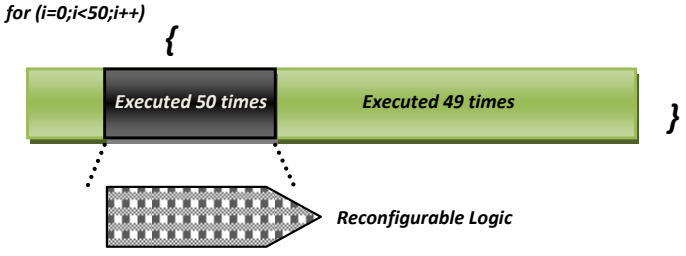


Figure 2.28: Just a small part of the loop can be optimized

Figure 2.29a and Figure 2.29b show, in the y-axis, the performance improvements (speedup factor) when implementing a different number of subroutines or loops (x-axis) on reconfigurable logic, respectively. The hot spots are chosen in order of relevance, where the first on the list is the most executed one (number of iterations times number of instructions in the hot spot). It is assumed that each one of these hot spots would take just one cycle for being executed on reconfigurable hardware. As it can be observed, the performance gains demonstrated are very heterogeneous. For a group of algorithms, just a small number of subroutines or loops implemented on fine grain reconfigurable logic are necessary to show good speedups. For others, the level of optimization is very low. One reason for the lack of optimization is the methodology used for code allocation on the reconfigurable logic, explained above. This way, even if there are a huge number of hot spots subject to optimization, but presenting different dynamic behaviors, just a small number of instructions inside these hot spots could be optimized. This shows that automatic tools, aimed at searching the best parts of the software to be transformed to reconfigurable logic, might not be enough to achieve the necessary gains. Consequently, human interaction for changing and adapting parts of the code would be required.

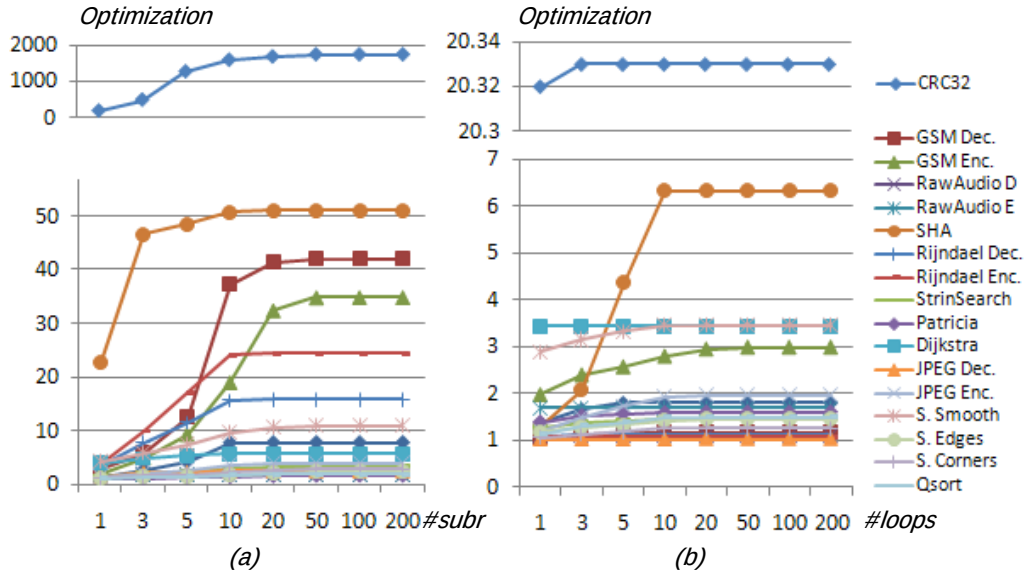


Figure 2.29: Performance gains considering different numbers of (a) subroutines and (b) loops being executed in 1 cycle in reconfigurable logic

In the first experiment, besides considering infinite hardware resources and no communication overhead between the processor and reconfigurable logic, it is also assumed an infinite number of memory ports with zero delay, which is practically infeasible for any relatively complex configuration. Now, in Figure 2.30, a more realistic assumption is considered: each hot spot would take 5 cycles to be executed on the reconfigurable logic. When comparing this experiment with the previous one, although the algorithms that present performance speedups are the same, the speedup levels vary. This obviously demonstrates that the performance impact of the optimized hot spots is directly proportional to how much they represent considering total algorithm execution time.

Figure 2.31 presents the same analysis that was done before, but considering more pessimistic assumptions. Now, each hot spot would take 20 cycles to be executed on the reconfigurable hardware. Although usually a reconfigurable unit would not take that

long to perform one configuration, there are some exceptions, such as large code blocks or those that have massive memory accesses. In the same Figure, one can observe that some algorithms present losses in performance. This means that, depending on the way the reconfigurable logic is implemented and how the communication between the GPP and RU is done, some hot spots may not be worth to be executed on reconfigurable hardware.

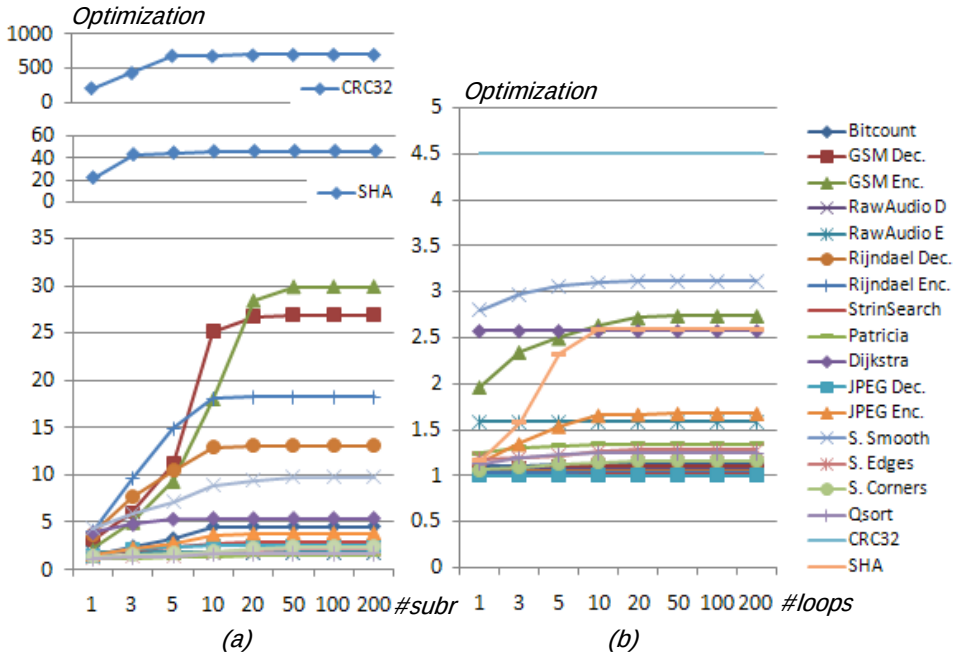


Figure 2.30: Same as presented before, but now considering 5 cycles per hot spot execution. (a) Subroutines and (b) loops

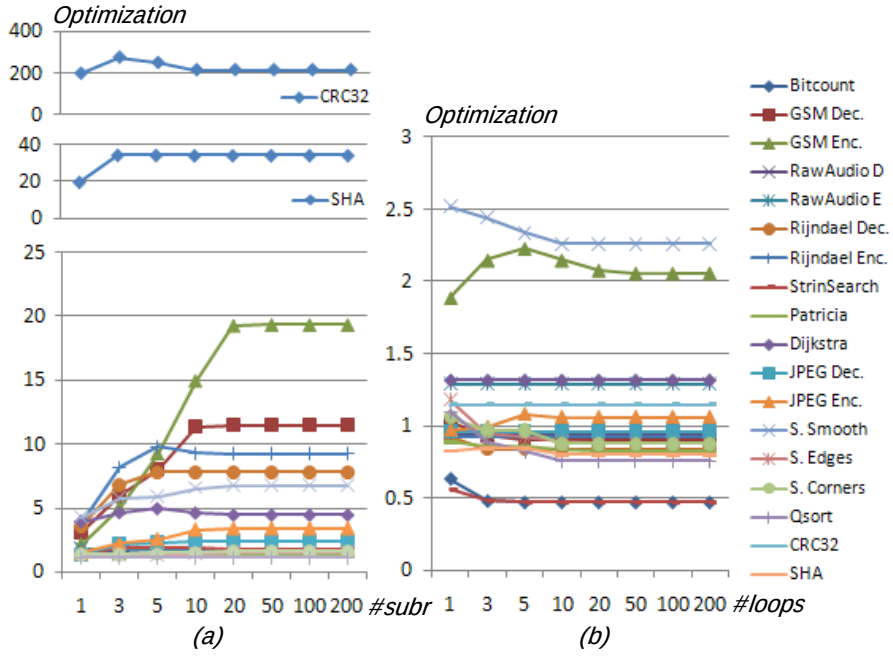


Figure 2.31: Now considering 20 cycles per hot spot execution. (a) Subroutines and (b) loops

In Figure 2.33a and Figure 2.33b a different methodology is considered: a subroutine or loop that can have as much reconfigurable logic as needed to be optimized, assuming that enough reconfigurable hardware is available to support infinite configurations. This way, entire loops or subroutines could be optimized, regardless if all instructions inside them are executed in all iterations, in opposite to the previous methodology. Figure 2.32 illustrates this assumption. A reconfigurable unit would be available for each part of the code.

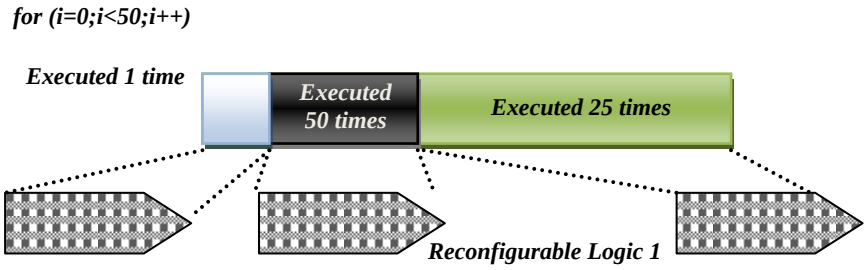


Figure 2.32: Different pieces of reconfigurable logic are used to speed up the entire loop

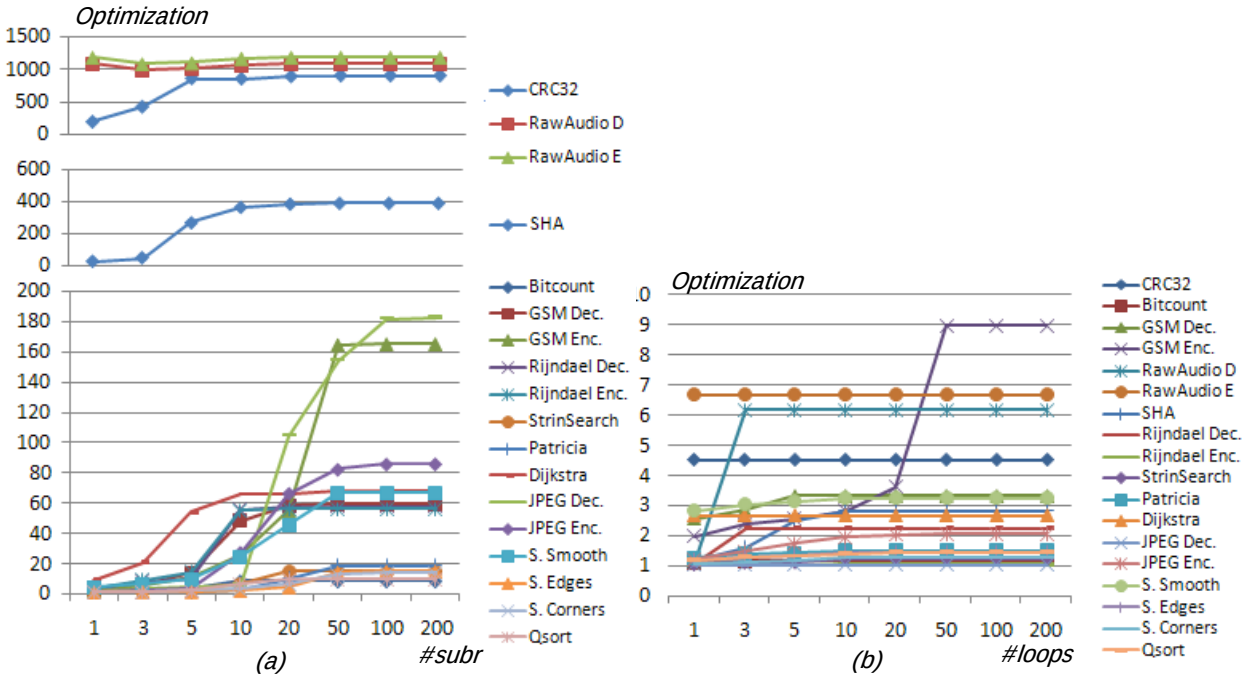


Figure 2.33: Infinite configurations available for (a) subroutine optimization: each one would take 5 cycles to be executed. (b) The same, considering loops.

In this experiment, it is considered that the execution of each configuration would take 5 cycles. Comparing against Figure 2.30 (same experiment using a different methodology), huge improvements are shown, mainly when considering subroutine optimizations. This, in fact, reinforces the use of totally or partially dynamic reconfigurable architectures, which can adapt to the program behavior during execution. For instance, considering a partially reconfigurable architecture executing a loop: the part of the code that is always executed could remain in the reconfigurable unit during

all the iterations, while sequences of code that are executed in certain time intervals could be configured when necessary.

2.5.3 Coarse Grain Reconfigurable Architectures

Now, the performance improvements when considering such architecture are analyzed. Since it works at the instruction level and, in this case, no speculative execution is supported, the optimization is limited to basic block boundaries. The level of optimization is directly proportional to the usage of BBs (Figure 2.26): for a determined basic block, the more it is executed, more performance boosts it represents. Even though this coarse grain reconfigurable array does not demonstrate the same level of performance gains as fine grain reconfigurable systems show, more and different configurations are available to be executed on this kind of system. This way, applications that do not have very distinct kernels could be optimized.

Considering the ideal assumption of one configuration taking just one cycle to be executed, let us compare the instruction level optimization against the subroutine level, which had shown more performance improvements than the loop level, as expected. When comparing the results of Figure 2.34a against the ones of Figure 2.29, one can observe that for some algorithms the number of basic blocks optimized does not matter: just executing one subroutine in reconfigurable logic would achieve a high performance speedup. However, mainly for the complex algorithms at the bottom of the figure, the level of optimization is almost the same for basic blocks or subroutines. This way, using the instruction level reconfigurable unit would be the best choice: it is easier and cheaper to implement 10 different configurations for a coarse grain logic than 10 for the FPGA based one.

When assuming that 5 cycles are necessary for the execution of each configuration in coarse grain reconfigurable hardware, there is a tradeoff between execution time and how complex the basic blocks are (in number of instructions, kind of operations, memory accesses etc). This assumption is demonstrated in Figure 2.34b: in the *Rinjdael* algorithms, the optimization is worth until a certain number of basic blocks being implemented on reconfigurable logic. After that, there is a performance loss. In Figure 2.34c, considering 20 cycles per basic block execution on the reconfigurable array, this situation is even more evident. This shows that, as for fine grain reconfigurable architectures, there is a necessity of small reconfiguration time and context loading. However, this is easier to be achieved in this simulated coarse grain architecture: the size of each configuration is much smaller than fine grain configurations.

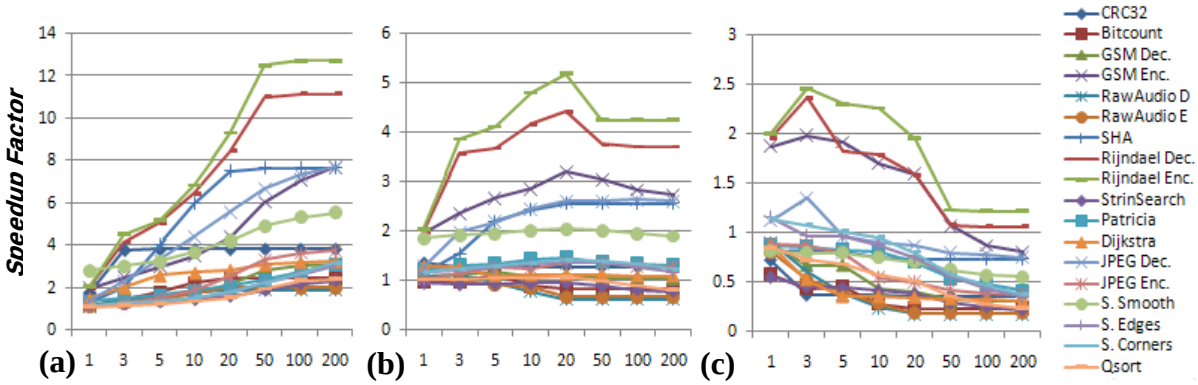


Figure 2.34: Optimization at instruction-level with the basic block as limit. (a) 1 cycle, (b) 5 cycles, (c) 20 cycles per BB execution

2.5.4 Comparing both granularities

Considering fixed applications, or yet those with long lifetime periods such as an MP3 player, FPGA based reconfigurable systems with high granularity grains still can be a good choice. Some algorithms present huge performance improvements, such as *CRC32*, *SHA* or *Dijkstra*. They need to optimize a small number of hot spots (the most executed kernels) to achieve such gains. This strategy, however, usually requires long development times that may not be acceptable. Furthermore, the industry trend goes to the opposite direction: the number of different applications being executed on the systems is increasing and the characteristics of these workloads have been changing, getting more heterogeneous: considering the embedded systems field, some of the applications are not as datastream oriented as they used to be in the past. Applications with mixed (control and data flow) or pure control flow behaviors, where sometimes no distinct kernel for optimization can be found, are gaining popularity. These affirmatives are reinforced by the MIBench analysis in sub-section 2.5.1.

Hence, for each application, different optimizations are required. This, in consequence, lead to an increase in the design cycle, since mapping code to reconfigurable logic usually involves some transformation, manual or using special languages or tool chains. The solution would be the employment of simpler coarse grain based reconfigurable architectures. Although they do not bring as much improvement as the fine grained approaches show, they could be easier to implement due to its simplicity.

Furthermore, according to the authors in (THEODORIDIS et al., 2007), there are some other reasons about why one should employ a coarse grained reconfigurable system, as follows:

- *Small configuration contexts*. Coarse grain reconfigurable units need a few configuration bits, which are order of magnitude less than those required if FPGAs were used to implement the same operations. In the same way, a small amount of bits is necessary to establish the interconnections among its basic processing elements because the interconnection wires are also configured at word level.
- *Reduced reconfiguration time*. Due to the previous statement, the reconfiguration time is reduced. This permits coarse-grain reconfigurable

systems to be used in applications that demand multiple and run-time reconfigurations.

- *Reduced context memory size.* Still because the first statement, the context memory size also reduces. This allows the use of on-chips memories, which permits switching from one configuration to another with low configuration overhead.
- *High performance and low power consumption*. This stems from the hardwired implementation of coarse grained units and the optimally design of interconnections for the target domain.
- *Silicon area efficiency and reduced routing overhead*. This comes because coarse grained units are optimally-designed hardwired units which are not built by combing a number of CLBs and interconnection wires, which results in reduced routing overhead and better area utilization.

In contrast, these are the main disadvantages of using a fine grain reconfigurable array such as the ones based on FPGA, according to the same authors:

- *Low performance and high power consumption.* This happens because word level modules are built by connecting a number of CLBs using a large number of programmable switches, causing performance degradation and power consumption increase.
- *Large context and configuration time*. The configuration of CLBs and interconnections wires is performed at bit-level by applying individual configuration signals for each CLB and wire. This results in a large configuration context that has to be downloaded from the context memory, increasing the configuration time. The large reconfiguration time may degrade performance when multiple and frequently-occurred reconfigurations are required.
- *Large context memory*. As a consequence of the previous statement, large reconfiguration contexts are produced which demand a large context memory. Because of that, in many times the reconfiguration contexts are stored in external memories increasing even more the time for reconfiguration.
- *Huge routing overhead and poor area utilization*. To build a word-level unit or datapath a large number of CLBs must be interconnected, resulting in huge routing overhead and poor area utilization. In many times a great number of CLBs are used only for routing purposes and not for performing logic operations. It has been shown that in many times for the commercially available FPGAs, up to 80–90% of the chip area is used for routing purposes (HON, 1996).

However, still according to the authors in (THEODORIDIS et al., 2007), the development of universal coarse-grain architecture to be used in any application is an “unrealistic goal”. This statement comes mainly from the fact that it is very hard to adapt the reconfigurable unit for a great number of different kernels, since the optimization is usually done at compile time. This way, even coarse grained architectures would be restricted to a specific domain.

Reinforcing this idea, it is very interesting to note that the totally of the referenced works about reconfigurable architectures, analyzed in section 2.4, employ as benchmark set exactly the ones which have very distinct kernels subject of optimization, and those that are very dataflow oriented. These two characteristics make these benchmarks the ones that are the most suitable for execution in reconfigurable fabric, as previously discussed. They correspond to just one area in a graph considering two axis (number of distinct kernels and control/dataflow behavior), as one can observe in Figure 2.35. As explored in the sub-section 2.5.1, this case is far for being the reality of embedded systems and, of course, of the general purpose computation field.

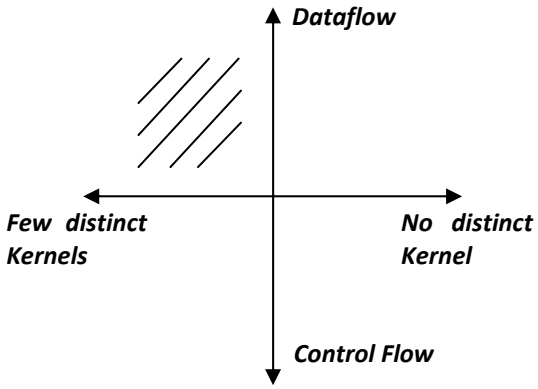


Figure 2.35: Different algorithm behaviors that can influence the usability of a reconfigurable system

2.5.5 The necessity of dynamic optimization

As commented before, even using coarse-grain reconfigurable architectures, one main problem still stands: they are efficient just for a determined field of application. To make this scenario even worse, the new era of embedded systems gives to the user the opportunity of installing and execute different applications, which behavior is non-predictable while the production of the devices, as in the general purpose computation. This lack of flexibility can be just solved with dynamic optimization: the system's ability of adapt itself during execution. This will be the subject of the next chapter.

3 DYNAMIC OPTMIZATION TECHNIQUES

In this chapter, two different techniques regarding dynamic optimization are analyzed: Trace Reuse and Binary Translation. As some of their principles are used for dynamic optimization with reconfigurable systems, recent works regarding this subject are discussed next. Finally, the main differences and advantages of the proposed technique when comparing against the previously reported ones are demonstrated.

3.1 Trace Reuse

The instruction reuse approach (SODANI; SOHI, 1998) is based on the principle of instruction repetition. This approach relies on the idea of an instruction with the same operands is repeated a large number of times during the execution of a program (SODANI; SOHI, 1998b). This way, instead of executing the instruction again using a functional unit, the result of this instruction is fetched from a special memory.

The main advantage of this technique is that instructions with larger delays (such as multiplications) can be executed faster. Additionally, there are secondary positive effects regarding the resources of the processor, such as freeing functional units, slots in the reservation stations and in the reorder buffer, the reduction of the instruction fetch and data bandwidth (fewer accesses in the register bank and in the memory). These effects potentially increase the possibility of executing additional instructions, if there is still ILP available.

The idea of trace reuse (GONZALEZ et al., 1999) extends the previous approach, in the sense that it is applied to a group of instructions (called of trace by the authors) instead of just one, as illustrated in Figure 3.1. It is based on the input and output contexts. A context is composed by the program counter, registers and memory addresses. Trace reuse works as follows: for a given sequence of instructions, the context of the processor, considering the first instruction of this sequence, is saved. Then, the output context, which is the result of the whole set of instructions that belongs to that sequence, is also saved, after this sequence was normally executed by the processor. After that, each time that the first instruction of this sequence is sent for execution again, the processor state is updated with the output context fetched from a special memory, avoiding the execution of that trace on the processor. This memory is called Reuse Trace Memory (RTM). Each entry of the RTM is illustrated in Figure 3.2. These entries can be indexed by the PC register, for example.

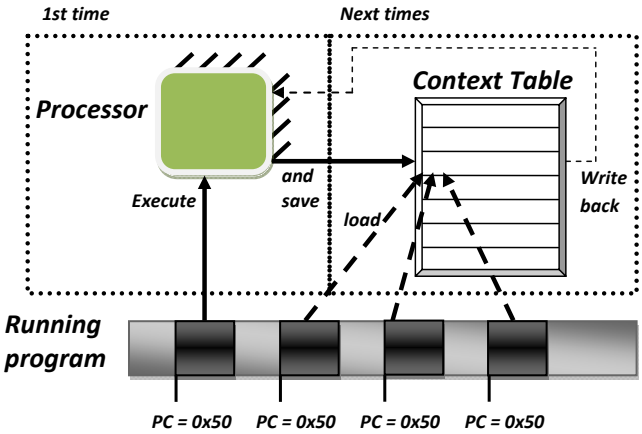


Figure 3.1: The trace reuse approach

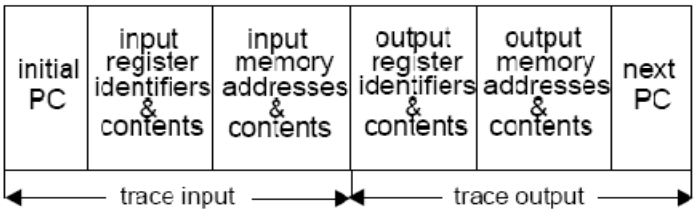


Figure 3.2 : A RTM entry (GONZALEZ et al., 1999)

The results presented are very promising. However, they can only be achieved when considering optimal resources or ideal assumptions. The minimum table size evaluated in the referred paper has 512 entries. This size would imply in a huge memory footprint, even for nowadays on die cache implementations. Moreover, it seems that the authors assume that the accesses in the table take just one cycle, which is very optimistic when considering the minimum size (512 entries), and almost impossible to be implemented with 256k entries (the maximum proposed).

The authors also implemented three different scheduling policies. Although it is not clearly stated on the paper, it is very likely that these policies consider an infinite window size of instructions to be analyzed. Furthermore, the scheduling is done by some kind of “oraculus”, which means that always the best composition of traces is considered to be saved in the special memory. It is important to stand out that defining the best policy for scheduling these instructions can be a very complex job to be done: multiples instructions can compose multiple traces and finding the best combination can demand a huge computational effort – which is very hard to be executed on the fly.

This way, the study lacks of realistic assumptions that should include at same time: a finite realistic window size, smaller RTM sizes with different and larger delays, less registers and memory accesses allowed per cycle; a study about the costs of the scheduling algorithm using a finite window; the costs of comparing registers and memory values with the current trace context etc.

In (JUAN HUANG, 1999) another technique is presented, called block reuse, with the purpose of, analogous to the one previously shown, reuse sequence of instructions. Although this technique is less general in the sense that it is limited to basic blocks (in the trace reuse approach a loop could be reused, for instance), the author analyzes its

possibilities with more realistic examples, as well as its costs. The Simplescalar Toolset (BURGER; AUSTIN, 1997) was employed for this case-study. It simulates a MIPS-like processor, using a configuration with four integer ALUs, one integer multiply/divide unit – and the same number of functional units for floating points computation – issuing and committing up to four instructions per cycle. The resulting speedup values range from 1.01 to 1.37, with an average of 1.15. The benchmarks were compiled with the GCC –O2 level of optimization.

Finally, in (COSTA et al., 1999) the authors presented a technique called Dynamic Trace Memoization, which uses *memoization tables* in order to detect at real time traces that can be potentially reused. In (PILLA et. al, 2003) this approach is extended in order to support speculative execution. In (PILLA et al., 2006) the technique is combined with value prediction and restricted hardware resources, reducing the number of trace candidates and the size of their contexts, achieving a good speedup of 1.21, on average.

Some of the main drawbacks of these techniques can be cited. Some algorithms do not present gains in performance because of the low level of input locality. Moreover, even if there is a high level of instruction reuse (or sequences of them), usually these instructions do not have the same input context. Therefore, the main disadvantage of this technique emerges: besides the necessity of saving the locality of the instruction, basic block or trace, it is also demanded to save the input context. This can lead to a huge number of possibly variations, consequently increasing the memory necessary to keep the configurations.

3.2 Binary Translation

The concept of binary translation (ALTMAN; SHEFFER, 2000) (ALTMAN et al., 2001) is very ample and can be applied in various levels. Basically, there is a system, which can be implemented in hardware or software, responsible for analyzing the running program. Then, some kind of transformation is done in the code, with the purpose of keeping the software compatibility (the reuse of legacy code without the need of recompilation), to provide means to enhance the performance or even both (Figure 3.3).

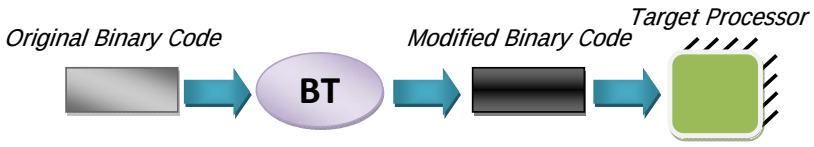


Figure 3.3: Binary Translation Process

According to (ALTMAN et al., 2000), there are three different kinds of binary translation:

- Emulator: interprets program instructions at run time. However, the transformed instructions are not saved or cached for future reuse;
- Dynamic Translator: besides interpreting the program, it saves previous translations to be used next time so that the overhead of translation and optimization can be amortized over multiple executions. One example of dynamic translation is just-in-time (JIT) compilers, as the ones used for Java execution.

- Static Translator: this kind does the job offline, having the opportunity of more rigorous optimization. It can also be used to generate execution profiles to give some sort of assistance for the processor in order to enhance performance during run-time.

While in the emulator and dynamic translator approaches there are run-time overheads due to the analysis during program execution, static translators as a stand-alone tool requires end-user involvement – being not transparent.

Nevertheless, there are other concepts regarding Binary Translation (ALTMAN; SHEFFER, 2000):

- Source architecture: The original (legacy) architecture *from* which translation occurs;
- Target Architecture: The architecture *to* which translation occurs;
- Virtual Machine Monitor (VMM): the system responsible for controlling the binary translation;
- Translation Cache: The memory where the translations can be stored. This cache is not necessarily implemented in hardware.

Another concept intrinsically connected to binary translation is dynamic optimization. While dynamic binary translation is JIT compilation from the binary code of one architecture to another, dynamic optimization is run-time improvement of the code. Usually, the general term Binary Translation is also applied when both techniques are used together.

Besides the JIT compiler, commented before, there are other examples of the different types of Binary Translation. The Hewlett-Packard Dynamo (BALA; BANERJIA, 1999) operates entirely at runtime in order to dynamically generate optimized native retranslations of the running program's hot spots. In fact, it is a software optimizing software, previously compiled and executing on the target machine. It operates transparently (any kind of interference from the user is not necessary) monitoring program behavior in order to find these hot spots to be optimized, using low-overhead techniques. Then, this modified code is executed again when necessary. Operating on HP-UX, Dynamo has a code size of less than 265 Kilobytes. Another example of the same approach, but with a different purpose, is the Compaq's FX!32, aimed to allow the execution of 32-bit x86 Windows applications on Alpha computers.

There are other architectures that mix hardware and software to perform BT. DAISY (GSCHWIND, 2000), from IBM, is one of those. It uses the PowerPC as source architecture and a special architecture based on VLIW, named DAISY VLIW, as target. The DAISY software is the VMM, responsible for the translation, and runs on the PowerPC, as can be observed in Figure 3.4. It is important to point out that, in opposite to Dynamo, which runs above the HPUX operating system, DAISY runs below its operating system. This way, it can be considered even more transparent to the final user, in the sense that one cannot identify it as a service or software running in the operating system.

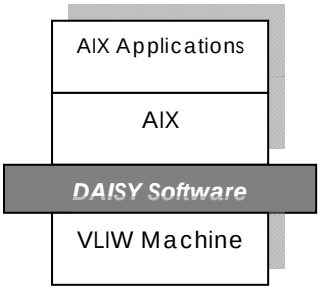


Figure 3.4: Daisy layers

Figure 3.5 shows how DAISY system is composed. The DAISY VMM code is stored in the DAISY flash ROM. When the system powers up, the VMM code is copied to the DAISY portion of memory in the PowerPC, and execution begins. After the VMM software initializes itself and the system, it begins translating the code of the PowerPC flash ROM to be executed on the VLIW processor. Then, this translated firmware loads the operating system (in this case, AIX Unix), which DAISY likewise translates and executes. After that, any application that is executed on the AIX can benefit from the binary translation and be executed on the VLIW processor.

The decision on where blocks of instructions to be translated begins is done based on loop back branches and function returns. The decision on what sequence worth to be translated and executed is based on a minimal number of instructions that compose a block, or if it has sufficient parallelism. These requirements vary depending on whether the instructions in the block have been executed frequently or not. DAISY performs a variety of optimizations as: ILP scheduling with data and control speculation, loop unrolling, alias analysis, load-store telescoping, dead code elimination etc (EBCIOGLU, 1996).

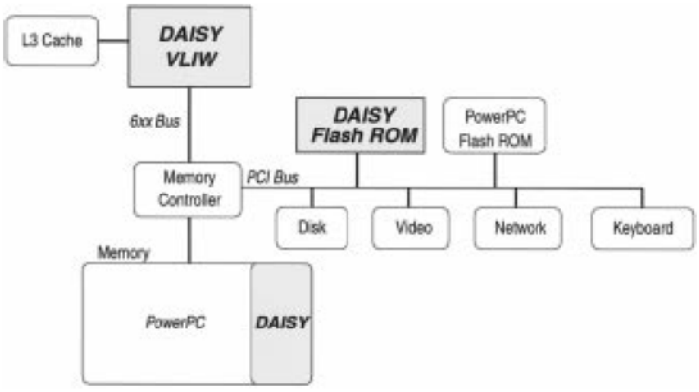


Figure 3.5: DAISY system (GSCHWIND, 2000)

The Transmeta Crusoe (KLAIBER, 2000) (Figure 3.6) shares several similar elements with DAISY. The significant difference is that Crusoe emulates an x86 system, while DAISY emulates a PowerPC. Both perform full system emulation including not only application code, but also operating systems and other privileged code. Furthermore, both use an underlying VLIW chip specifically designed to support BT as target architecture, aimed for high performance. There are also similarities regarding the optimization process: code is first interpreted and profiled and, if a

fragment turns out to be frequently executed (more than 50 times), it is translated to native Crusoe instructions.

Aside from the different source architectures emulated, Crusoe and DAISY differ in their intended use. DAISY is designed for use in servers and consequently is a big machine capable of issuing 8–16 instructions per cycle, with gigabytes of total memory. Given this large machine, the DAISY VMM emphasizes extraction of parallelism when translating from PowerPC code. DAISY reserves 100 MB or more for itself and its translations. Crusoe is aimed at low power and mobile applications such as laptops and palmtops. The processor issues only 2 to 4 instructions per cycle and has 64–128 MB of total memory in a typical system. Thus, Crusoe reserves 16 MB for itself and its translation.

In benchmark tests, DAISY can complete the equivalent of 3 to 4 PowerPC instructions per cycle. Transmeta has claimed that the performance of a 667-MHz Crusoe TM5400 is about the same as a 500-MHz Pentium III (SHANKLAND, 2000).

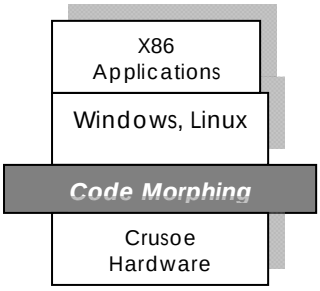


Figure 3.6: Transmeta layers

Binary translation can also produce other effects in the future, following the tendency of *write once, run everywhere.* For example, using Binary Translation in order to perform transformations from different ISAs to a unique target architecture, all efforts for optimization could be targeted to just one kind of hardware.

3.3 Dynamic Detection and Reconfiguration

3.3.1 Warp Processing

Trying to unify some of these ideas with reconfigurable systems, Vahid et al. (STITT; VAHID, 2002) (LYSECKY; VAHID, 2004) presented the first studies about the benefits and feasibility of dynamic partitioning using reconfigurable logic, producing good results for a number of popular embedded system benchmarks. The structure of this approach, called warp processing, is a SOC. It is composed by a microprocessor to execute the software, another microprocessor where the CAD algorithm runs (responsible for the hardware/software portioning), a dedicated memory and an FPGA.

The system is illustrated in Figure 3.7, and the follow steps for its functioning are necessary:

1. Initially, the software binary is loaded into the instruction memory;
2. The microprocessor executes the instructions from this software binary;
3. Profiler monitors the instructions and detects critical regions in binary;

Then, the on-chip CAD:

4. reads in critical regions;
5. decompiles a given critical region into a control data flow graph (CDFG);
6. synthesizes the decompiled CDFG to a custom (parallel) circuit;
7. maps this circuit onto FPGA;
8. replaces instructions in the original binary to use the FPGA hardware.

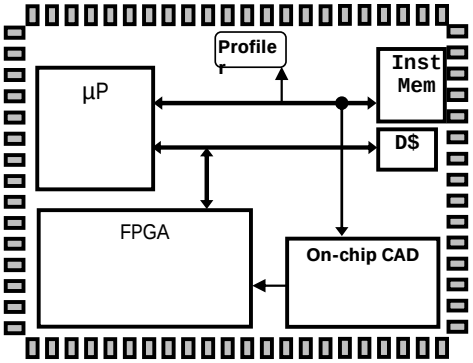


Figure 3.7: The Warp processor system

The steps performed by the on-chip CAD can be observed in more details in Figure 3.8. They can be executed on hardware because an optimized CAD algorithm was developed, which brings a relatively small memory overhead considering this kind of software. Moreover, besides this simpler CAD algorithm, the FPGA implementation has a simpler logic than usual. In its switch matrices, all nets are routed using only a single pair of channels and each Configurable Logic Block (CLB) is connected just to its adjacent. The way routing was implemented facilitates the development of the CAD software since, according to this same work, routing is by far the most time-consuming on-chip CAD task. Moreover, the CLB have fewer resources than the regular ones: two 3 inputs/2 output Look-Up Tables (LUT).

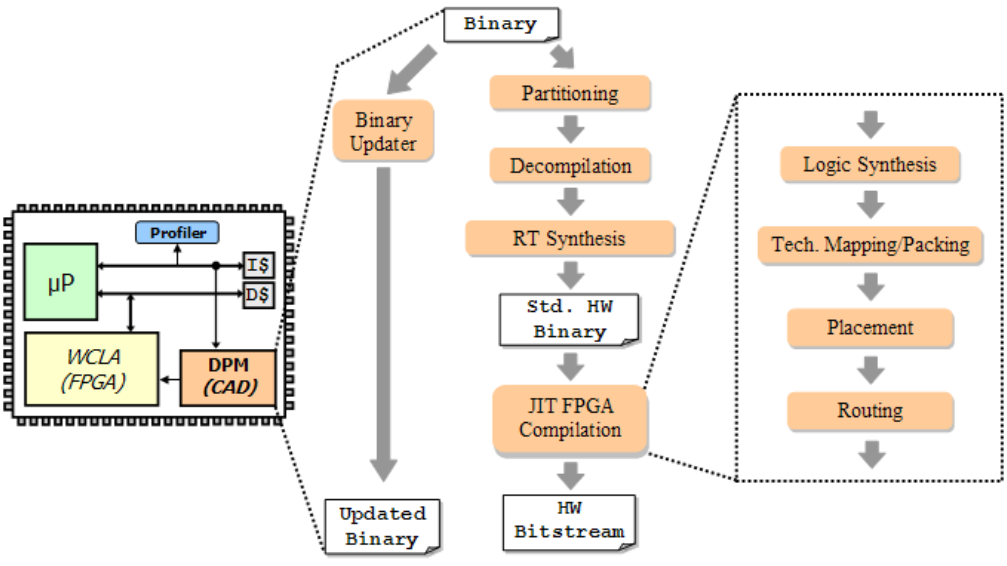


Figure 3.8: Steps performed by the CAD software

Last results show the benefits of warp processing for soft-core processors (LYSECKY; VAHID, 2005). The technique was implemented in a Microblaze-based FPGA. Several embedded systems applications from the Powerstone and EEMBC benchmark suites were analyzed. The experimental setup considers a MicroBlaze processor implemented using the Spartan3 FPGA. The MicroBlaze processor core has a maximum clock frequency of 85 MHz. However, the remaining FPGA circuits can operate at up to 250 MHz. The processor was configured to include a barrel shifter and multiplier, as the applications considered required both operations.

Figure 3.9 and Figure 3.10 present the performance speedup and energy reduction of the MicroBlaze-based warp processor compared with a standalone MicroBlaze processor. The software application execution was simulated on the MicroBlaze using the Xilinx Microprocessor Debug Engine, where instruction traces for each application were obtained. This trace was used to simulate the behavior of the on-chip profiler to determine the single most critical region within each application.

The system was also compared with readily available hard-core processors. Overall, the MicroBlaze warp processor has better performance than the ARM7, ARM9, and ARM10 processors and requires less energy than the ARM10 and ARM11 processors. The ARM11 processor executing at 550 MHz is on average 260% faster than the MicroBlaze warp processor but requires 80% more energy. Furthermore, compared with the ARM10 executing at 325 MHz, the MicroBlaze warp processor is on average 30% faster while requiring 26% less energy. Therefore, while the MicroBlaze warp processor is neither the fastest nor the lowest energy alternative, it is comparable and competitive with existing hard-core processors, while having all the flexibility advantages associated with soft-core processors.

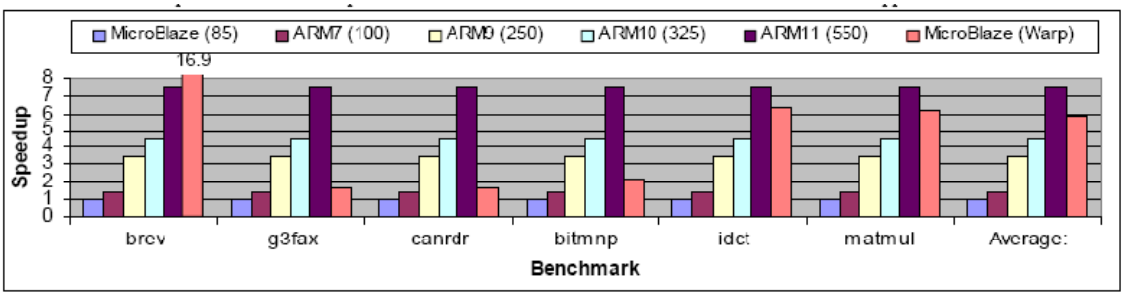


Figure 3.9: Speedups of MicroBlaze-Based warp processor when comparing against different versions of the an ARM. Powerstone and EEMBC benchmark applications were used.

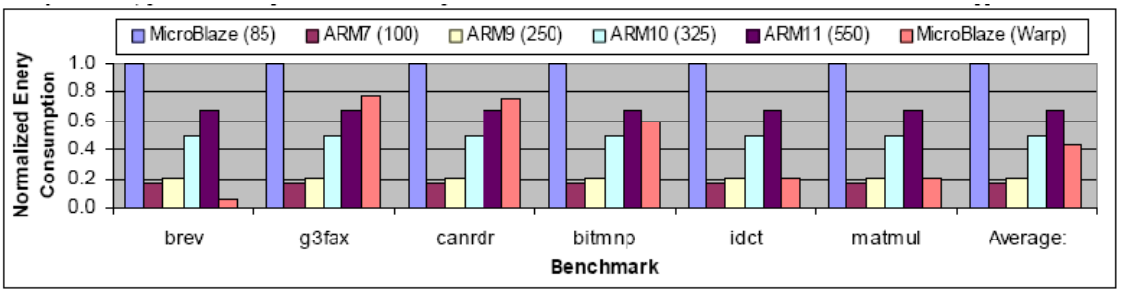


Figure 3.10: Normalized energy consumption in the different versions using the same benchmark set.

However, there are some drawbacks when using such technique. The first one is that it uses a complete SOC, with different hardware communicating with each other, which could increase the design cycle time and make it harder to test. Moreover, even if the CAD system used is simplified, it remains complex: it does decompilation, CFG analysis, place and route etc, requiring significant resources: up to 8 MB of memory are necessary for its execution, still big for nowadays on-die memories. Another deficiency is related to the FPGA: besides the long latency and area overhead, it is also power inefficient due to the excessive switches and the considerable amount of static power dissipated. Moreover, because of the memory footprint required for keeping configurations, this technique is just limited to critical parts of the software, working at its best just in very particular programs, such as the filter based ones.

3.3.2 Configurable Compute Array

In (CLARK et al., 2003) a similar reconfigurable structure comparing to the one used in this work is presented. This array is called Configurable Compute Array (CCA) and it is tightly coupled to an ARM processor. The proposed CCA is implemented as a matrix of heterogeneous FUs. There are two types of FUs in this design, referred to as type A and B, for simplicity. Type A FUs perform 32-bit addition/subtraction as well as logical operations. Type B FUs perform only the logical operations, which include and/or/xor/not, sign extension, bit extraction, and moves. To ease the mapping of subgraphs onto the CCA, each row is composed of either type A FUs or type B FUs.

The matrix can be characterized by the depth, width, and operation capabilities. Depth is the maximum length dependence chain that a CCA will support. This corresponds to the potential vertical compression of a dataflow subgraph. Width is the number of FUs that can work in parallel. This represents the maximum instruction-level parallelism (ILP) available to a subgraph execution. Figure 3.11 shows the block diagram of a CCA with depth 7. In this figure, type A functional units (FU) are represented with white squares and type B units with gray squares. The CCA has 4 inputs and 2 outputs. Any of 4 inputs can drive the FUs in the first level. The first output delivers the result from the bottom FU in the CCA, and the second output is optionally driven from an intermediate result from one of the other FUs.

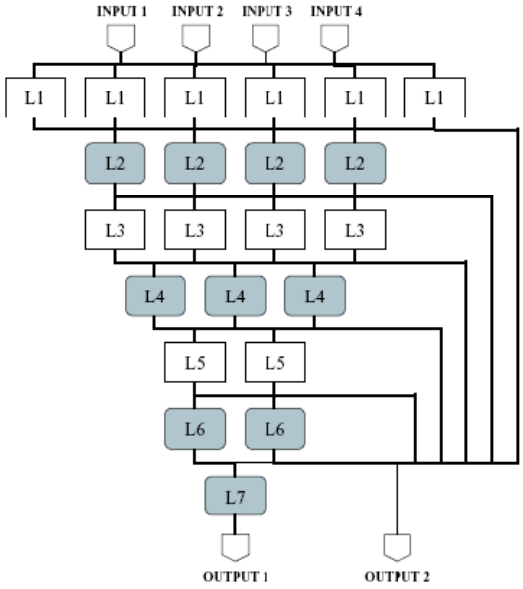


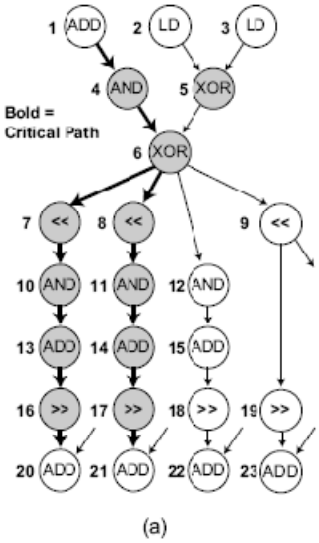
Figure 3.11: Example of a CCA with 4 inputs and 2 outputs, with 7 levels of operations allowed in sequence

Feeding the CCA involves two steps: the discovery of which subgraphs are suitable for running on the CCA, and their replacement by microops in the instruction stream. Two alternative approaches are presented: static and dynamic.

Static discovery finds subgraphs for the CCA at compile time. Those are marked in the machine code by using two additional instructions, so that a replacement mechanism can insert the appropriate CCA microops dynamically. Using these instructions to mark patterns allows for binary forward compatibility, meaning that as long as future generations of CCAs support at least the same functionality of the one compiled for, the subgraphs marked in the binary are still useful. However, as the code is changed, the backward compatibility is lost anyway.

Dynamic discovery, in turn, assumes the use of a trace cache to perform sub-graph discovery on the retiring instruction stream. Its main advantage is that the use of the CCA is completely transparent to the ISA. Theoretically, the static discovery technique can be much more complex than the dynamic version, since it is performed offline; thus, it does a better job on finding subgraphs.

Figure 3.12 demonstrates how a sequence of instructions is mapped into a typical CCA configuration. In Figure 3.12a the CFG representing a part of the code in Figure 3.12b is shown. The bold circles represent the instructions that are in the critical path. These instructions will be mapped in the CCA. Finally, Figure 3.12c shows the measurements in terms of delay for the functional units that will be used for this sequence. This measurement proves that it is possible to perform more than one single computation within a single clock cycle.




```
m ^= p [0];
r ^= p[ 1];
r ^= (((s[(m>>24L)] +
       s[0x0100+((m>>16L)&0xff)]) ^
       s[0x0200 + ((m>> 8L)&0xff)]) +
       s[0x0300+((m )&0xff)]) & 0xffffffff;
m ^= p[ 2];
m ^= (((s[(r>>24L)] +
       s[0x0100+((r>>16L)&0xff)]) ^
       s[0x0200+((r>> 8L)&0xff)]) +
       s[0x0300+((r )&0xff)]) & 0xffffffff;
```

(b)

Opcode	Adders	Cycles
+	1.0	0.30
AND	0.12	0.06
<<, >>*	0.01	~ 0
XOR	0.16	0.09

* These are shifts by compile time constants

(c)

Figure 3.12: An example of mapping a piece of software into the CCA (CLARK et al., 2003)

The instruction grouping discovery technique proposed in this paper is highly similar and based on the rePlay Framework (PATEL; LUMETTA, 2001). This process of identifying potential subgraphs for optimization works as follows: initially, the application is profiled to identify frequently executed frames. The most frequently executed ones are then analyzed and subgraphs that can be beneficially executed on the CCA are selected. Then, the compiler generates machine code for the application, with the subgraphs explicitly identified to facilitate simple dynamic replacement. Frames have the same purpose of superblocks or use the same principle of trace cache; they have one single entry point and one single exit point, encapsulating one single flow of control in an atomic fashion: if one instruction within a given frame is executed, the rest of them are also executed. A frame is composed by instructions based on speculative branch results. If one transformed branch (called as assertion) is miss predict inside the frame, the whole frame execution is discarded.

The subgraphs considered were limited to have at most four inputs and two outputs. Furthermore, memory, branch, and complex arithmetic operations were excluded from the subgraphs. Previous work (YU; MITRA, 2004) has shown that allowing more than four input or two output operands would result in very modest performance gains when memory operations are not allowed in subgraphs. In Figure 3.13 one can observe the potential of implementing a CCA together with the microprocessor, demonstrating the speedup versus a relative area cost of each CCA for three different applications. As can be seen, with a small cost in terms of hardware, good performance improvements can be achieved

Some evaluations were performed in order to analyze what would be the best configuration for the CCA, given a determined group of benchmarks. It was shown that the depths vary across a representative subset of three groups of benchmarks. For example, in blowfish (part of the MIBench set), 81.42% of dynamic subgraphs had a depth less than or equal to 4 instructions. On average of all the 29 applications executed through the system, about 99.47% of the dynamic subgraphs have a depth 7 instructions or less. Depth is a critical design parameter, since it directly affects the latency of the CCA. It was discovered that a CCA with depth 4 could be used to implement more than

82% of the subgraphs considering that diverse group of applications. Going below depth of 4 seriously affects the coverage of subgraphs that can be executed on the CCA. Therefore, only CCAs with depths between 4 and 7 were considered in this study.

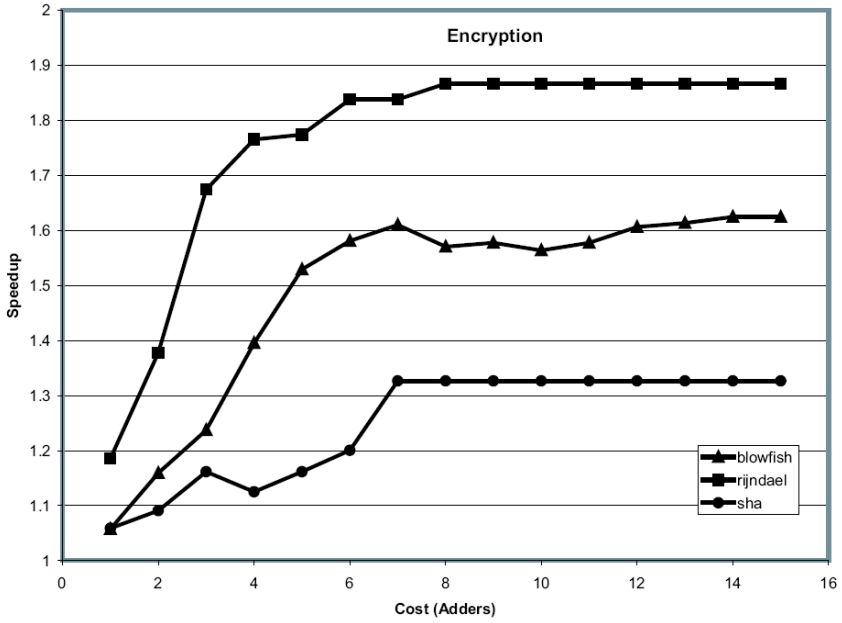


Figure 3.13: Speed-up versus Area overhead, represented by the cost of adders (CLARK et al., 2003)

The search for the ideal width was also performed. Using the same set of applications, it was figured that 4.2% of dynamic subgraphs had width of 6 or less in row 1, with only 0.25% of them having width 7 of more. In the following rows of the matrix, the widths decrease. For instance, the average width in row 2 is 4 or 5. This data suggests that a CCA should be triangularly shaped to maximize the number of subgraphs supported without wasting resources.

It is important to point out that operations involving more expensive multiplier/divider circuits were not allowed in subgraphs, because of latency considerations. Additionally, memory operations were also disallowed. Load operations have non-uniform latencies, due to cache effects, and so supporting them would entail incorporating stall circuitry into the CCA. This would increase the delay of the CCA and make integration into the processor more difficult. Although shifts did constitute a significant portion of the operation mix, barrel shifters were too large and incurred too much delay for a viable CCA.

This technique also has drawbacks. The CCA does not support memory operations, shifts and multiplications – or any operation that involves a different delay when comparing to the functional units employed, limiting its field of application. As a consequence of this fact, it has a limited number of inputs and outputs. Moreover, it uses very complicated graph analysis and changes the binary in the static discovery. In the same way, the dynamic approach also makes use of a complex graph analysis, since it is based on the RePlay Framework (PATEL; LUMETTA, 2001), which leads to a huge memory overhead. Because of that, just high-level simulations using the

Simplescalar Toolset are reported. No measurements are given in terms of area overhead, power consumption and timing and there are no details about how a CGF is transformed to an array's configuration. The overheads considering the array, and the detection and reconfiguration delays are not discussed at all.

Despite all these drawbacks, both papers discussed previously are very important to this thesis because they show the potential of executing parts of the software in reconfigurable logic and its feasibility.

3.4 Similarities and Differences of Previous Works

Comparing to the techniques cited before, the proposed approach also takes advantage of a reconfigurable system, but a coarse grain one, so it can be implemented in any technology, not being limited to FPGAs only. Together with that, the use of binary translation avoids the need for code recompilation or the utilization of extra tools, making the optimization process totally transparent to the programmer. Adding to the fact that the array is not limited to the complexity of fine-grain configurations, the binary code detection and translation algorithm are very simple, in the sense that they take advantage of the hierarchal structure of the reconfigurable array. The system can be implemented using trivial hardware resources, in contrast to the complex on-chip CAD software or graph analyzers used in the related work.

Moreover, the proposed approach relies on the same basic idea of trace reuse, where sequences of instructions are repeated. However, it presents the advantage that just one entry in the special memory is needed for the same sequence of instructions, even when they have different contexts (as input values from the registers). This takes the pressure off from the cache system, making possible its implementation with a small memory footprint, with realistic assumptions concerning execution and accesses times, even for present days technologies. Furthermore, using the proposed technique, the number of inputs of outputs for each context can be larger, and any instruction, including load/store and multiplications, are supported.

4 THE PROPOSED RECONFIGURABLE ARRAY

As already explained, the proposed technique can be divided in two main groups: the first one, which is the reconfigurable array and its implementation, and the second, which is the Binary Translation algorithm. This chapter focus on the reconfigurable array, leaving the discussion about the BT to the next chapter. In the following sub-sections it is demonstrated the structure of the array, the architectures where it was coupled and the particularities of each reconfigurable system according to these architectures. Different processors were used, two based on Java and the others based on the MIPS processor. As the former kind is a stack machine, and the MIPS is a pure RISC processor, two different architectures of the reconfigurable array were implemented, according to these two paradigms.

4.1 Java Processors targeted to Embedded Systems

While the number of embedded systems does not stop growing, new and different applications, like cellular phones, mp3 players and digital cameras keep arriving at the market. At the same time, embedded systems are getting more complex, smaller, more portable and with more stringent power requirements, posing great challenges to the design of this kind of system. Additionally, another issue is becoming more important nowadays: the necessity of reducing the design cycle.

This last affirmative is the reason why Java is becoming more popular in embedded environments, replacing traditional languages. Java has an object-oriented nature, which facilitates the programming, modeling and validation of the system. Furthermore, being multiplatform, a system that was built and tested in a desktop, for instance, can migrate to different embedded systems with a small number of modifications. Moreover, Java is considered a safe language, and has a small code size, since it was built to be transmitted through internet.

Not surprisingly, recent surveys revealed that the presence of Java in devices such as consumer electronics (digital TV, mobile phones, home networking) as well as industrial automation (manufacturing controls, dedicated hand held devices) is increasing day by day (MULCHANDANI, 1998) (LAWTON, 2002). Nowadays, most of the commercialized devices as cellular phones already provide support to the language. This means that current design goals might include a careful look on embedded Java architectures, and their performance versus power tradeoffs must be taken into account.

However, Java is targeted neither to performance nor to energy consumption, since it requires an additional layer in order to execute its *bytecodes*, called Java Virtual Machine (JVM), responsible for the multiplatform feature of Java. That is why executing Java through the JVM could not be a good choice for embedded systems. A

solution for this issue would be the execution of Java programs directly in hardware, taking off this additional layer, but at the same time maintaining all the advantages of this high-level language. Using this solution highlights again another execution paradigm that was explored in the past (KOOPMAN, 1989): stack machines. Since the JVM is based on this paradigm, obviously the hardware for native Java execution should follow the same approach, in order to maintain full compatibility.

Because of all the reasons discussed above, a Java processor, called Femtojava, was designed. This processor executes natively Java bytecodes and it is targeted to embedded systems. It is available in different versions, generated according to the designer preferences when using the Sashimi Tool (ITO et al., 2001). In this work, the reconfigurable array was coupled to two different versions of the Femtojava processor (called Multicycle and Low-Power) (BECK; CARRO, 2003B) and, for performance comparisons, its VLIW implementation is also studied (BECK; CARRO, 2004). In the next sub-sections, the structure of this processor, the architecture of the reconfigurable array and how they work together will be demonstrated.

4.1.1 A Brief Explanation of the Femtojava Processor

The Femtojava processor is a stack-based microcontroller that executes Java bytecodes. General characteristics of the Femtojava processor are: reduced instruction set, Harvard architecture, and small size. The size of its control unit is directly proportional to the number of different bytecodes utilized by the application. From the Sashimi Tool, the Java bytecodes of the application are analyzed, and the control unit is generated supporting only the bytecodes used by that application. The simplest architecture of the family is a multicycle Femtojava, which takes three to fourteen cycles to execute each instruction, shown in Figure 4.1.

The second architecture is called Femtojava Low-Power. It has a five stages pipeline: instruction fetch, instruction decoding, operand fetch, execution, and write back, as shown in Figure 4.2. The first stage, instruction fetch, is composed by an instruction queue of 9 registers. The first instruction in the queue is sent to the instruction decoder stage. The decoder has four functions: the generation of the control word for that instruction, to handle data dependencies, to analyze the forwarding possibilities and to inform to the instruction queue the size of the current instruction, in order to put the next instruction of the stream in the first place of the instruction queue. This is necessary because of the existence of variable length instructions: they can have one or two immediate operands, or none at all.

Operands fetch is done in a variable size register bank, defined a priori in earlier stages of the design. Stack and the local variable pool of the methods are available in this register bank. This structure facilitates the call and return of methods, taking advantage of the JVM specification, where each method is located by a frame pointer in the stack. Moreover, there are two extra registers: SP and VARS. They point to the top of the stack and to beginning of the local variable storage of the current method, respectively. Depending on the instruction, one of them is used as base for the operand fetch. Once the operands are fetched, they are sent to the fourth stage, where the operation is executed. The branch prediction is static, in order to save area. All branches are supposed to be not taken. If the branch is taken, a penalty of three cycles is paid. The write back stage saves, if necessary, the result of the execution stage back to the register bank, again, using the SP or VARS as base.

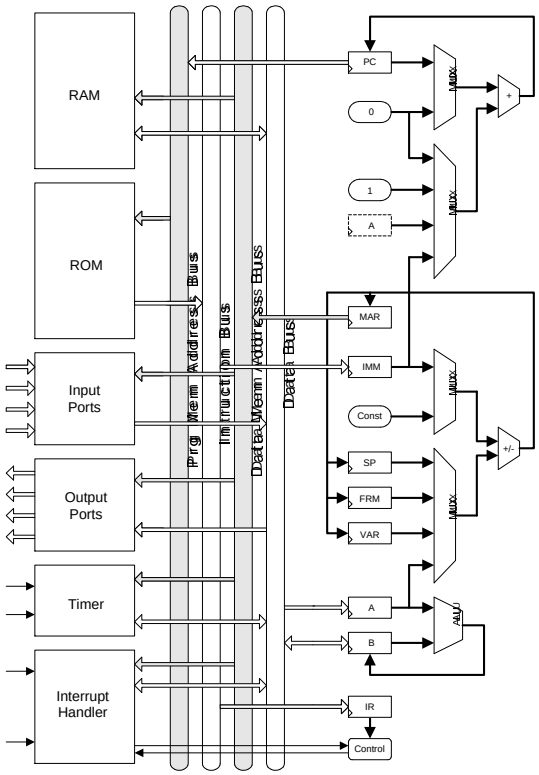


Figure 4.1: Femtojava Multicycle

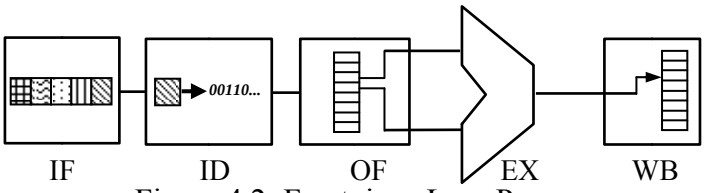


Figure 4.2: Femtojava Low-Power

In the results section, for performance comparisons, a VLIW version of the same Java processor was used (BECK; CARRO, 2004), which is an extension of the pipelined one. Basically, it has its functional units and instruction decoders replicated. The additional decoders do not support the instructions for call and return of methods, since they are always in the main flow. The local variable storage is placed just in the first register file. When the instructions of other flows need a value from the local variable pool, they must fetch from there. Each instruction flow has its own operand stack, which has less registers than the main stack, since the operand stacks for the secondary flows do not grow as much as the main flow does.

The VLIW packet has a variable size, avoiding unnecessary memory accesses. A header in the first instruction of the word informs to the instruction fetch controller how many instructions the current packet has. The search for ILP in the Java program is done at the bytecode level. The algorithm works as follows: all the instructions that depend on the result of the previous one are grouped in an operand block. The entire Java program is divided in these groups and they can be parallelized respecting the functional unit constraints. This approach is also used in the proposed reconfigurable system and will be explained in more details later.

4.1.2 Architecture of the Array

The reconfigurable array is tightly coupled to the processor. It is implemented as an ordinary functional unit in the execution stage. The array is divided in blocks, called cells. The operand block (a sequence of Java bytecodes) previously detected by the BT algorithm is fitted in one or more of these cells in the array. The approach to detect such operand blocks will be demonstrated separately in the next chapter.

The cell can be observed in Figure 4.3. Three functional units (ALU, shifter, ld/st), working in parallel, compose the initial group of the cell. After this first group, two more groups with the same structure follow in sequence. Each cell of the array has just one multiplier and takes exactly one processor equivalent cycle to have its execution completed. Being limited to the critical path of the Femtojava processor, it brings no delay overhead to the pipeline. At the end of each cell, there are two additional functional units: a branch unit and an extended ld/st one, made for the execution of the *iastore* instruction (fetches a static value from memory, taking the address from the stack), since it needs three operands, instead of two, as usual. It is necessary to highlight that there is no sequential logic at all in the array: no registers, flip flops etc. It is composed just by pure combinational logic. The cell was developed to be as small as possible, but at the same time to support the maximum number of simple instructions, respecting the processor´s critical path. This way, just one multiplier was included. However, because of their smaller delay, more ALUs to work in sequence are supported within a cell.

As it is stressed in the same figure, it is important to note that one of the operands always comes from the previous operation. This facilitates the routing of the cell, since just one multiplexer is necessary to choose the second operand. This characteristic will be better analyzed in the next subsection, when this array is compared to the one implemented for RISC-like processors.

These cells can be organized side by side or in a sequential fashion depending on the maximum number of parallel or data-dependent instructions that is desired to be executed in the array. Figure 4.4 illustrates an array composed by 6 cells. In this particular example, the array can execute up to two instructions in parallel at a time and up to 9 instructions in sequence, without considering multiplications. In the best case, it would be possible to execute 18 instructions in the array, taking a total of just 3 cycles.

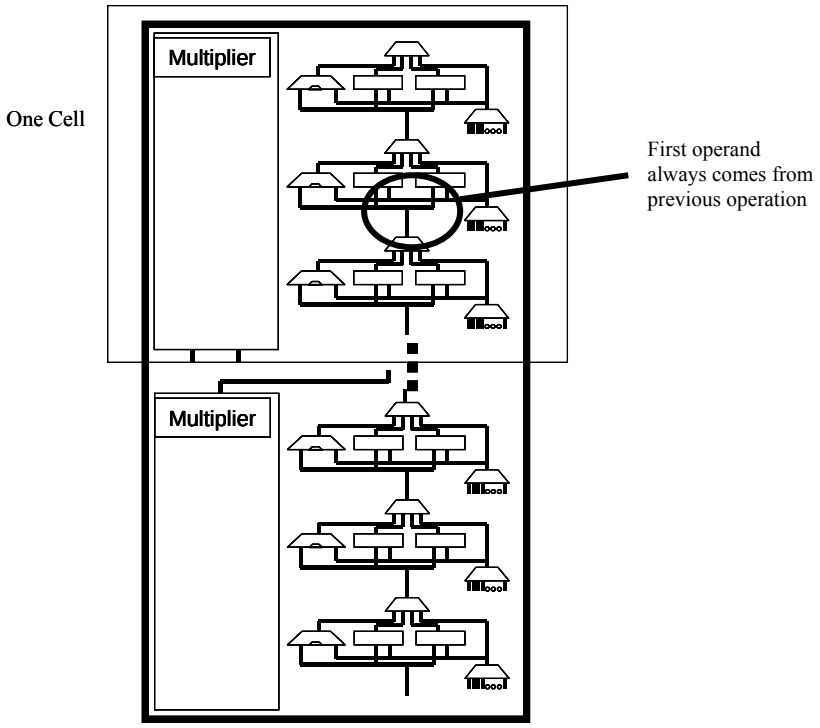


Figure 4.3: Two cells of the Array in sequence

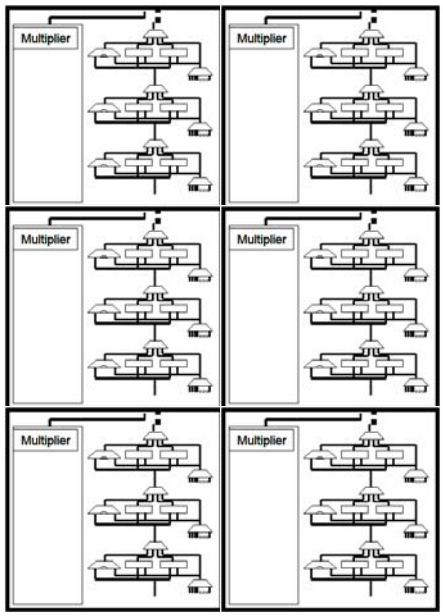


Figure 4.4: An example of an array's configuration.

4.1.3 Reconfiguration and execution

While the program is executed, when an address of a reconfigurable instruction is found, the reconfigurable unit detector sends information to the main processor. This address is the value of the Program Counter register of the first instruction of the sequence that was previously translated to a reconfigurable instruction by the BT mechanism. After that, the configuration for that reconfigurable instruction is sent to the array for its reconfiguration. As the PC is used for figure that a given sequence was already translated, and considering that this verification can be done at the first stage of the pipeline and the reconfigurable array is in the fourth stage, there are 3 cycles

available between the detection and the use of the array. As one cycle is necessary to find the cache line that has the array configuration, two cycles are still available for the reconfiguration. Finally, the control unit of the processor configures the array as the active functional unit; wait while the array performs its functions; and upgrades the Program Counter with the new PC address, in order to continue the normal operation.

Special attention must be given to some instructions, whenever they are in the reconfiguration or in the execution phases. If a *getstatic* instruction is found (a load access from main memory), its value will be fetched from the memory during the reconfiguration phase, since the address is static. The value fetched is saved in the operands field. It is also during the reconfiguration phase that the local variables of the method are fetched. As these local variables are kept in a dual-ported register bank in the processor, they can be fetched at the same time the static values from the memory are. Some unexpected actions can be taken during these fetches. Cache misses can occur in the case of *getstatic* accesses, or other instructions could be accessing the register bank making impossible the load of local variables. In these cases, extra cycles are necessary for the reconfiguration of the array.

Considering the execution phase, the two operands that will be used in the first operation group of the first cell always come from the register bank. After that, for each basic group of functional units of the cell, the first operand is the result of the previous operation. As it can be observed, in each cell it is possible to make 3 simple operations (arithmetic, logical, shift) in sequence. If in the middle of a cell it is necessary to perform a multiplication, the result of the current cell is bypassed to the end. Then, this multiplication will be executed in the next cell.

For instructions that save a value in the main memory, a buffer is used in order to avoid delaying the execution. Moreover, in the case of instructions that load/store values from/to main memory or to the local variable storage of the method, values can be bypassed. One example of this is when there is a load instruction in a local variable soon after a store in the same local variable. This avoids unnecessary accesses in the register bank or in the main memory, accelerating the execution and saving power. Finally, if there is an instruction *iaload*, which makes an access in the memory and calculates the access address dynamically, and a cache miss occurs, a penalty is paid and that cell in the array is executed again after the cache miss is resolved.

Considering the example to follow, the potential gains of using a reconfigurable array in a Java machine is shown: if there are five simple arithmetical or logic operations, one needs at least 11 cycles for the execution: 6 for pushing the operands into the stack, plus 5 cycles for the operations themselves. This is the optimal assumption, without considering pipeline stalls due to data dependency. On the other hand, the array would execute everything in just two cycles, after this sequence was properly translated by the BT hardware. If this sequence is repeated a certain number of times, meaningful gains can be achieved.

4.2 Differences in the structure: Stack vs. RISC

In this section we analyze the differences of the arrays implemented in the Java Processor and in the RISC ones. Although the basic idea is the same, the structure of the array changes as the computational paradigm changes as well.

As explained in the last sub-section, implementing the reconfigurable array in a Java processor has a great advantage: no routing for the first operand is necessary. This

occurs because Java is based on a stack machine. For instance, if one needs to subtract two operands using the result of a previous operation and a value from the stack, three steps are necessary: get the result from the previous operation, get the second operand from the stack, and finally operates them. Figure 4.5 illustrates this example if it was executed in the array. As can be observed, the subtraction is performed in the second level of the cell, using the ALU. The first operand for this operation comes from the add execution in the first level of the cell; and the second operand comes from a special table that holds immediate values, through another multiplexer. This multiplexer also could be used to bring some operand that was a result of an operation that was performed in lower levels of the array.

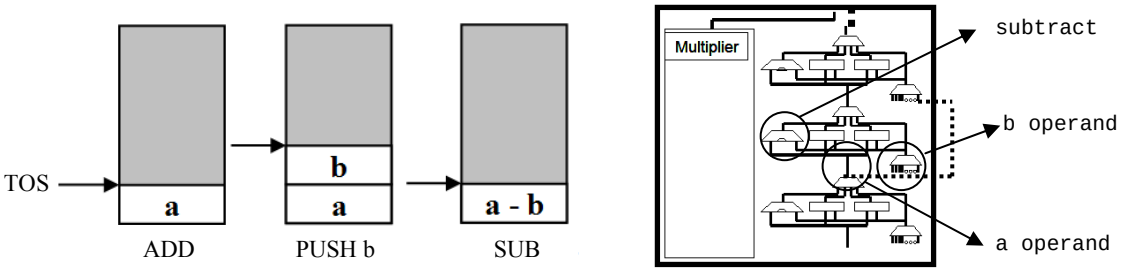


Figure 4.5: Because it is a stack machine, the routing in the array implemented in the Femtojava becomes simpler

A RISC operation, in turn, can be represented by this sequence of operations:

add R1, R3, R4

load R0, a

subtract R0, R0, R1

In opposite to stack operations, there is no guarantee that any operator in the Subtract instruction comes from the previous operation (it could exist any operation between the add and the subtract instruction, for instance). This way, the routing logic and the hardware itself are more complicated, as it will be observed in the rest of this chapter. Another important issue in stack machines is the temporality of the operands. Once they are consumed, they will not be used anymore (they were taken off from the stack forever). This also means that they do not need to remain in the array after they were used. It facilitates the distribution of the context inside the reconfigurable system, since each operand that comes from the stack will be used by just one functional unit and no more than that. On the other hand, in RISC machines, a source operand can be used even after it was already consumed. This way, it needs to remain available for the rest of execution in the array. Considering the previous example, right after the subtract operation, a multiplication, which also uses the value from R1, could be there. This fact will affect directly the policy of distribution of operands inside the reconfigurable logic.

4.3 RISC-like Architectures

To represent the general purpose computation field, an architecture very similar to the MIPS R10000 was employed (YEAGER, 1996), an out-of-order superscalar processor that executes the MIPS IV instruction set. In fact, the Simplescar Toolset

(BURGER; AUSTIN, 1997) is used. Simplescalar is a set of simulators that can work at different levels of abstraction. It has an instruction level simulator, a cache simulator, a superscalar out-of-order version and so on. It is highly customizable – for instance, in the superscalar version, innumerous options are available, from the cache size to the number of slots available in the instruction window. The Simplescalar is also customizable in the sense that it supports different ISAs. In this work it is used the PISA (Portable Instruction Set Architecture) (BURGER; AUSTIN, 1997), since it is highly based on the MIPS IV ISA. The second architecture employed is the MIPS R3000: the classic 5-stage RISC processor, which executes the first proposed MIPS ISA. This processor is still in use, mainly in the embedded system market.

4.3.1 Architecture of the array

A general overview of its organization is shown in Figure 4.6. The array is two dimensional, and each instruction is allocated in an intersection between one row and one column. If two instructions do not have data dependences, they can be executed in parallel, in the same row. Each column is homogeneous, containing a determined number of ordinary functional units of a particular type, e.g. ALUs, shifters, multipliers etc. Depending on the delay of each functional unit, more than one operation can be executed within one processor equivalent cycle. It is the case of the simple arithmetic ones. On the other hand, more complex operations, such as multiplications, usually take longer to be finished. The delay is dependent of the technology and the way the functional unit was implemented. Load/store (LD/ST) units remain in a different group of the array. The number of parallel units in this group depends on the amount of ports available in the memory. The current version of the reconfigurable array does not support floating point operations.

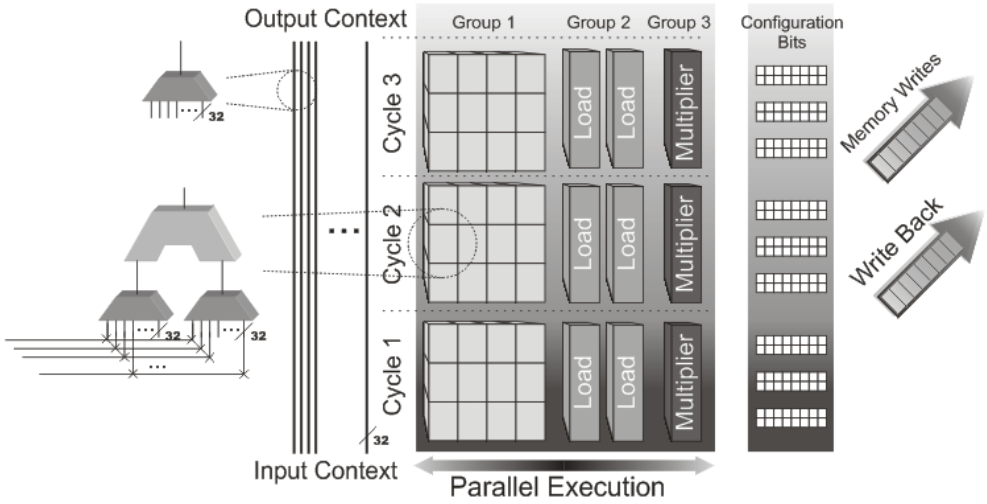


Figure 4.6: General overview of the reconfigurable array for RISC machines

For the input operands, there is a set of buses that receive the values from the registers. These buses will be connected to each functional unit, and a multiplexer is responsible for choosing the correct value. As can be observed in more details in Figure 4.7, where just a group with ALUs is shown, there are two multiplexers that will make the selection of which operand will be issued to the functional unit. We call them input multiplexers (Figure 4.7a). After the operation is completed, there is a multiplexer for each bus line that will choose what result will continue through that line. These are the output multiplexers (Figure 4.7b). As some of the values of the input context or old

results generated by previous operations can be reused by other functional units, the first input of each output multiplexer always holds the previous result of the same bus line. Note that, if one considers that the configuration of all multiplexers is set to zero at the beginning of any execution, the output context will be the same of the input context. Figure 4.8 shows in even more details another example: just one row, with five columns and two different groups – one composed by ALUs and the other one composed by LD/ST units. In the results chapter more details about the area overhead will be discussed.

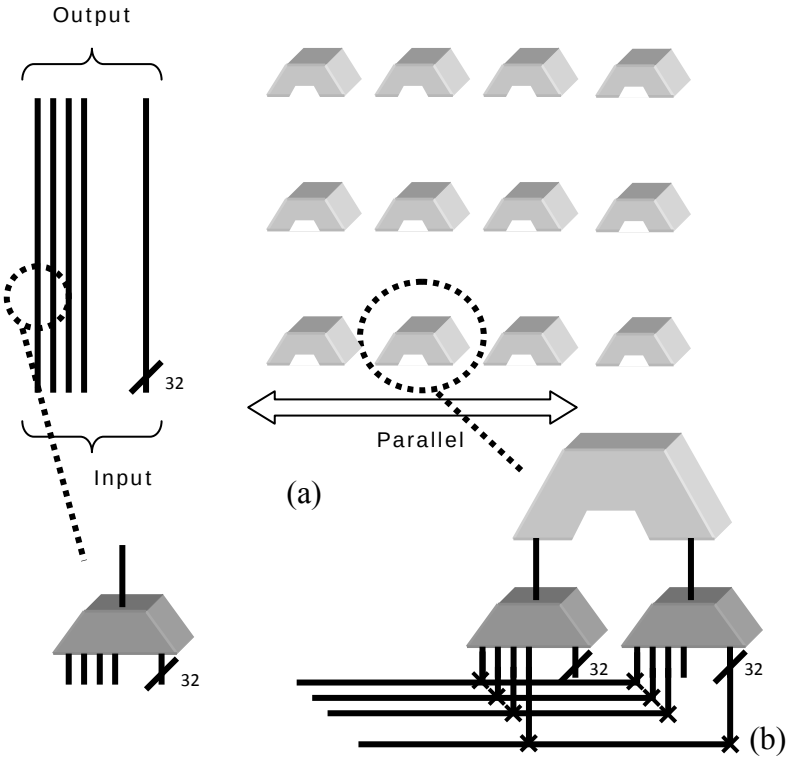


Figure 4.7: An overview of the basic architecture of the reconfigurable array

4.3.2 Reconfiguration and execution

The basic principle is the same as the Java architecture. The reconfiguration phase involves: the loading of the configuration bits for the multiplexers, functional units and immediate values from the special cache; and fetching of the operands that will be used by that configuration from the register bank. Again, a given configuration is indexed in the cache using the PC of its first instruction, and this address is obtained in the first stage of the pipeline (through the PC register). This way, since the array is supposed to start execution in the fourth stage (the execution stage in this case), there are three cycles available for the array reconfiguration. In cases three cycles are not enough (for example, there is a great number of operands to be fetched from the register bank) the processor will be stalled and wait for the end of the reconfiguration process.

After the reconfiguration is finished, execution begins. Memory accesses are done by the LD/ST units, and their access addresses can be calculated by ALUs located in previous lines, during execution, allowing memory operations even with those addresses that are not known at compile time. The operations that depend on the result of a load are allocated considering a cache hit as the total load delay. Then, if a miss occurs, the

whole array operation stops until the it is resolved. Finally, when the operands are not used anymore for that configuration, they are written back either in the memory or in the local registers. If there are two writes to the same register in a given configuration, just the last one will be performed, since the first one was already consumed inside the array by other instructions.

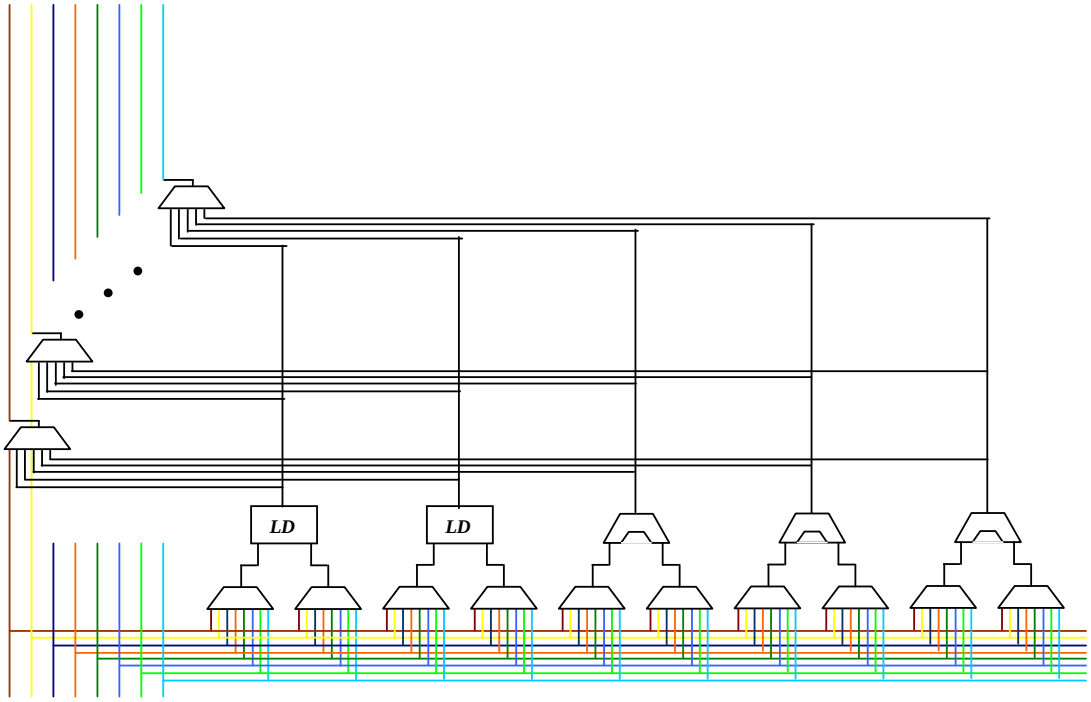


Figure 4.8: A row of the reconfigurable array. The input and output multiplexers and the functional units.

5 BINARY TRANSLATION

In the next two sections in this chapter, details about how the BT detection and transformation of the code to be executed in the reconfigurable hardware are described. This BT algorithm is called on this thesis of Dynamic Instruction Merging (DIM). The first section demonstrates DIM used in Java machines. The second one shows how the binary translation works for RISC processors.

5.1 BT Algorithm for Stack Machines

The search for the sequence of instructions in the Java program is done at the bytecode level in a very similar way of what the VLIW static analyzer does (BECK; CARRO, 2005) (BECK; CARRO, 2005b) (BECK; CARRO, 2005C). Sequences of instructions that depend on each other are grouped in a so-called operand block. The detection to find these blocks are very simple: when the stack pointer returns to the start address, previously saved, an operand block is found. Therefore, to detect an operand block just a single state machine is necessary. This is a particular characteristic of stack architectures as shown in (BECK; CARRO, 2004).

Figure 5.1 illustrates this procedure for a given sequence of instructions. Note that this sequence compounds an operand block. The stack pointer starts from a determined place in the stack and in the last instruction of this sequence it returns exactly to the same point.

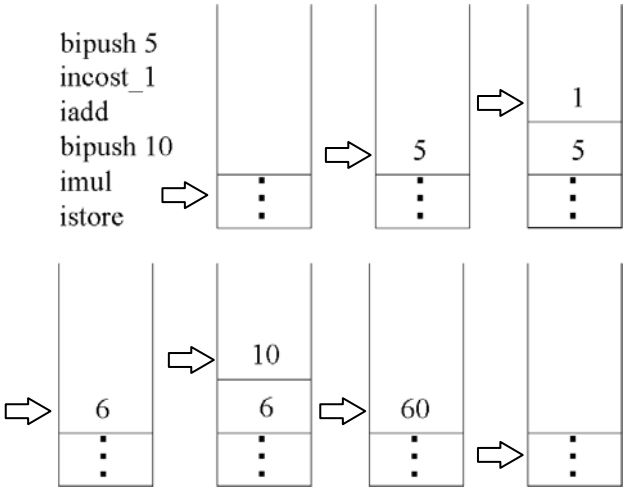


Figure 5.1: The simple process of finding an operand block in a stack machine

In order to better illustrate this approach, consider the sequence of instructions observed in Figure 5.2a. The first *imul* instruction will consume the operands pushed previously, by the instructions *bipush 10* and *bipush 5*. After that, the *ishl* instruction

will consume two more operands produced before by the previous *bipush*. The *iadd* instruction will consume the results of *imul* and *ishl*. Then, the *istore* will save the result of the *iadd* in the local variable pool. Finally, there are two more *bipush* instructions, which operands will be used by the last *imul*. However, they do not use any result of the set of instruction previously executed. In other words, their operand stacks are independent, forming two operand blocks (Figure 5.2b). Each one of these operand blocks compose a different configuration to be used in the future by the reconfigurable array, and will be saved in the reconfiguration cache (Figure 5.2c).

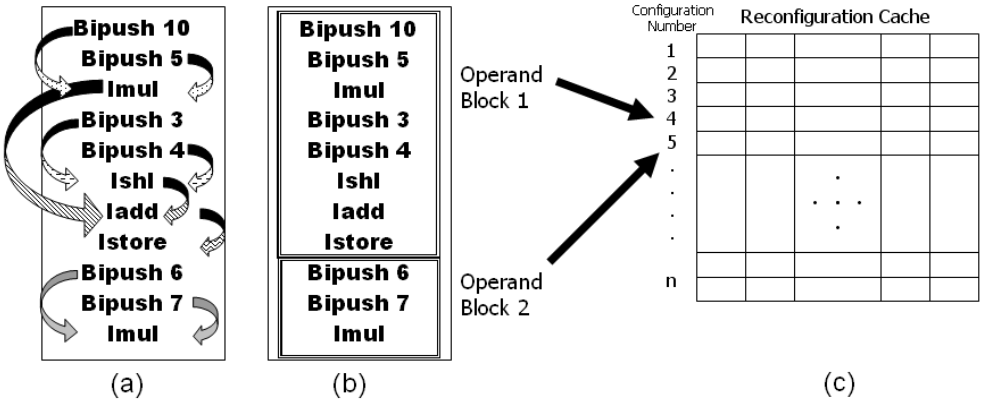


Figure 5.2: Identifying independent operand blocks

When a basic block limit is found, or the end of an operand block, the current operand block is finished, a write command for the reconfigurable cache is sent. This command saves the content of the buffer to the cache and the buffer is cleaned. This content is the list of the decoded instructions of the operand block. This list is made at real time, as the instructions are fetched from memory. Additionally, the Program Counter (PC) value of the beginning of the sequence must also be saved. This is how the detector will know when a sequence of instructions was previously saved in the cache and it is ready to be executed next time it is found. The PCs are saved in a bitmap fashion. This way, both writes and reads are fast and, as just one bit for address is necessary, no large amounts of memory are needed.

Each cell of the reconfigurable array (illustrated in Figure 4.3 of the previous chapter) must have the follow data in the reconfigurable cache:

- The operands field, where it can be found constant or immediate values; values from the memory or from the local variable storage; the value for the bypass of operands (16 bits).
- The bits for the configuration of the functional units (5 bits).
- A pointer indicating the address of a fetch in memory or local variable storage (16 bits).
- The bits for the multiplexers to make the routing (5 bits).

Moreover, these additional fields are necessary for the configuration of the array:

- The start and the relative end addresses, in order to update the PC value after the sequence of instructions is executed in the array (20 bits).
- The number of cycles taken by the operation in the array (5 bits)

- Additional 32 bits for immediate values or pointers for the first cell, and one more bit for the final multiplexer, for each cell in the array (33 bits).

Hence, for each cell in the array is necessary 159 bits of reconfiguration (42 bits of each part multiplied by 3, plus 33). Consequently, if the array were formed by 3 cells, it would be necessary 477 bits in the reconfiguration cache plus 58 bits of the additional information, totalizing 535 bits for saving each configuration of the array.

5.2 BT Algorithm for RISC machines

As explained in Section 3.3, there are differences in the detection in RISC machines when comparing to stack ones, making the BT process a little bit more complex. In this subsection, how the BT algorithm for RISC machines works is shown. To better explain it, the algorithm with many restrictions will be first demonstrated and, gradually, it will be improved until its current implementation.

5.2.1 Data Structure

Basically, some tables are necessary in order to make the routing of the operands inside the reconfigurable array as well as the configuration of the functional units. Other intermediate tables are also necessary, however, they are used just during the detection phase. These tables are illustrated in Figure 5.3, considering an array composed of twenty ALUs (five rows with four ALUs each). They can be described as follow:

- Write bitmap table: Saves information of data dependence of each row. This table is in fact composed by a large number of small bitmaps, more specifically, one per row. This bitmap just informs what registers in that row will be written. Note that it is not necessary to keep this information for each instruction as usual. Summarizing it in a bitmap for each row, it is possible to reduce the amount of hardware necessary to check true data dependences (RAW – read after write).
- Resource Table: Informs if a given functional unit is being used;
- Read Table: Informs what operand from the input context must be read by each functional unit. Note that this table has two inputs, since there are two source operands for each functional unit. It is important to stand out that the input context is basically an indirect table. In other words, not necessarily the first slot needs to keep the value of the register R1.
- Writes table: This table informs what value each context slot will receive. Note that this table is different when comparing to the read one. In the previous table, the multiplexers were responsible for choosing what values from the context slots would be issued to each functional unit. This table informs what values from the whole set of the functional units that compose each row will continue in each slot of the context bus.
- Context table: This table has two rows. The first one represents the input context, and it will be used in the reconfiguration phase for the operands fetch. The second one is called current table, and it will be used during the detection phase. Its final state represents what values will be written when the execution of the array finishes.

It is important to note that the tables follow the same structure as the reconfigurable array. In the case of the Resource Table, the X-axis represents the parallel execution of instructions through the time (y-axis). The Read Table is almost the same, with the difference that each column of the Resource Table is split in two (since each functional unit has two input operands). The X-axis of the Write Table is represented by each slot of the Context. The Y-axis is exactly the same as the previous ones: represents the result of each functional unit through the time.

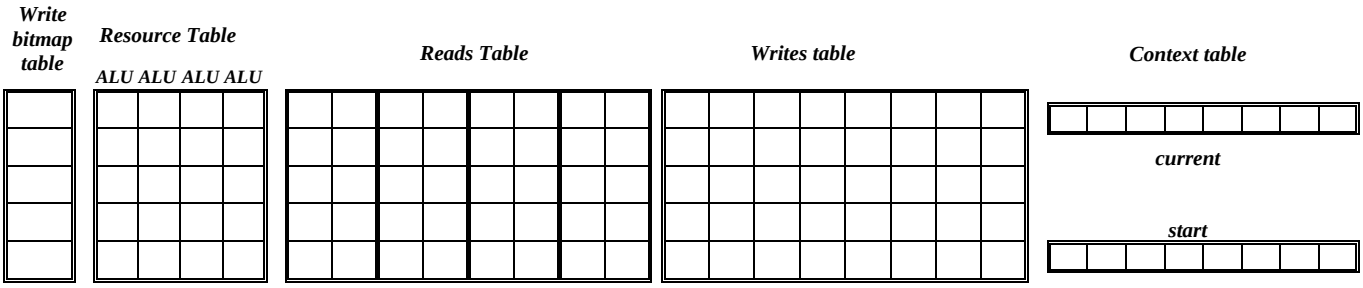


Figure 5.3: Tables necessary for the detection and configuration of the array

5.2.2 How it works

To better explain the algorithm, let us start with its simplest version, considering that the array is composed just by adders. The following steps represent pipeline stages when considering the implementation in hardware.

Considering that

```
inst op_w, op_r1, op_r2
```

where `inst` is the current instruction and `op_w`, `op_r1` and `op_r2` are the target and the source operands, respectively, the follow steps are necessary:

1st) Decode the instruction, returning the target and source registers of the current instruction;

2nd) In the write table, for each row from 0 to N, verify if `op_r1` and `op_r2` exist. If any one of them or both exist in the row *S*, row *O* equals to *S + 1*. Considering a bottom-up search, the row *S* is the last one where `op_r1` or `op_r2` appears, since they may be found in more than one row. If nor `op_r1` neither `op_r2` exist in any row of this table, row *O* equals to zero.

3rd) In the resource table, search in the columns of row *O*, from left to right, if there is a resource available for use. If there exists, we call this free column as *C*, and row *R* equals to *O*. If there is no resource available in row *O*, increment the value of *O* in 1 and repeat the same operation, until finding the resource. This way, row *R* equals to *O + N*, where *N* was the number of increments necessary until finding an available resource.

4th)

a) Update the bitmap based write table in row R with the value of `op_w`

b) Update column C in row R of the resource table as busy

c) Search in the current context table if there are `op_r1`, `op_r2` and `op_w`. For each one of these, if they exist, point *L1*, *L2* and *W* to `op_r1`, `op_r2`

and `op_w` respectively. If one of them does not exist in the table, the correspondent signal of write for each one of this values in this table is set, and the correspondent pointer (*L1*, *L2* or *W*) is updated.

5th)

a) Depending on the step 4c, the current context table is updated. If the pointer W is being written in the table, a flag indicating that it should be written back at the end of execution is set.

b) The initial context table is also updated, if one of the write signals concerning `op_r1` and `op_r2` are set.

c) In the write table, write the value of *W* in the row R, column *C*.

d) In the read table, write the values of *L1* and *L2* in row *R*, column *C* (it is important to remember that each column of this table has two slots, as explained earlier)

Summarizing the algorithm, for each incoming instruction, the first task is the verification of RAW (read after write) dependences. The source operands are compared to a bitmap of target registers of each row. If the current row and all above do not have that target register equal to any of the source operands of the current instruction, this instruction can be allocated in that row, in a column as left as possible. After that, the bitmap of target registers is updated. This way, for each incoming instruction it is necessary to analyze just one bitmap per row. Indirectly, such technique increases the size of the window of instructions, which is one of major limiting factors of increasing the ILP exploitation in superscalar processors, due to the number of comparators that is necessary (BURNS; GAUDIOT, 2002).

5.2.3 Example

Now it is shown a very simple example, where just add operations could be executed in the reconfigurable array. Consider the follow sequence of instructions:

```
add r7, r5, r6
add r8, r7, r6
add r9, r8, r6
add r1, r2, r7
add r4, r2, r7
```

The reconfiguration tables would be as demonstrated in Figure 5.4, gradually.

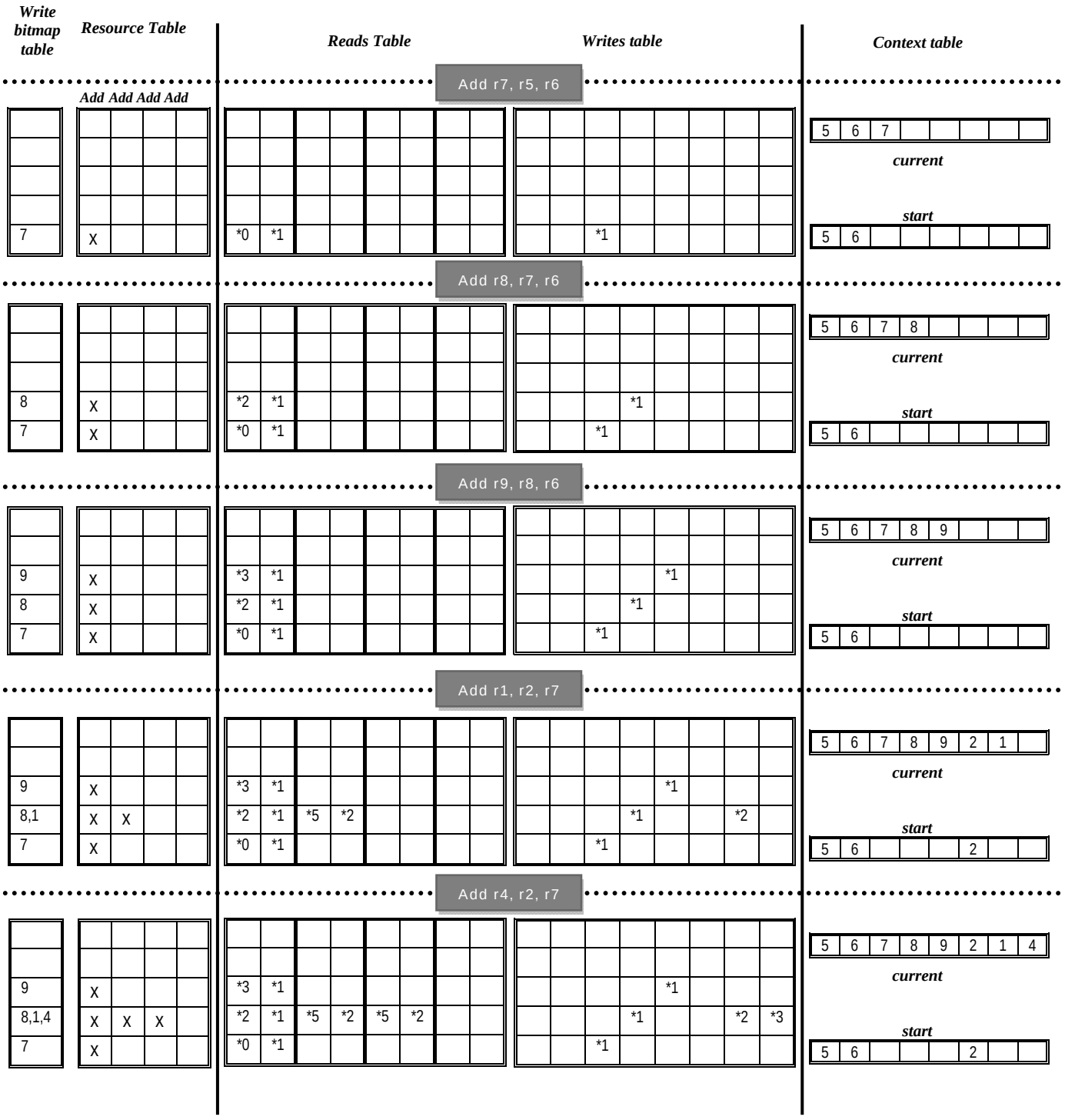


Figure 5.4: Behavior of the tables during the detection of instructions

Figure 5.5 shows how the configuration presented before would be represented in the structure of the array after its proper reconfiguration.

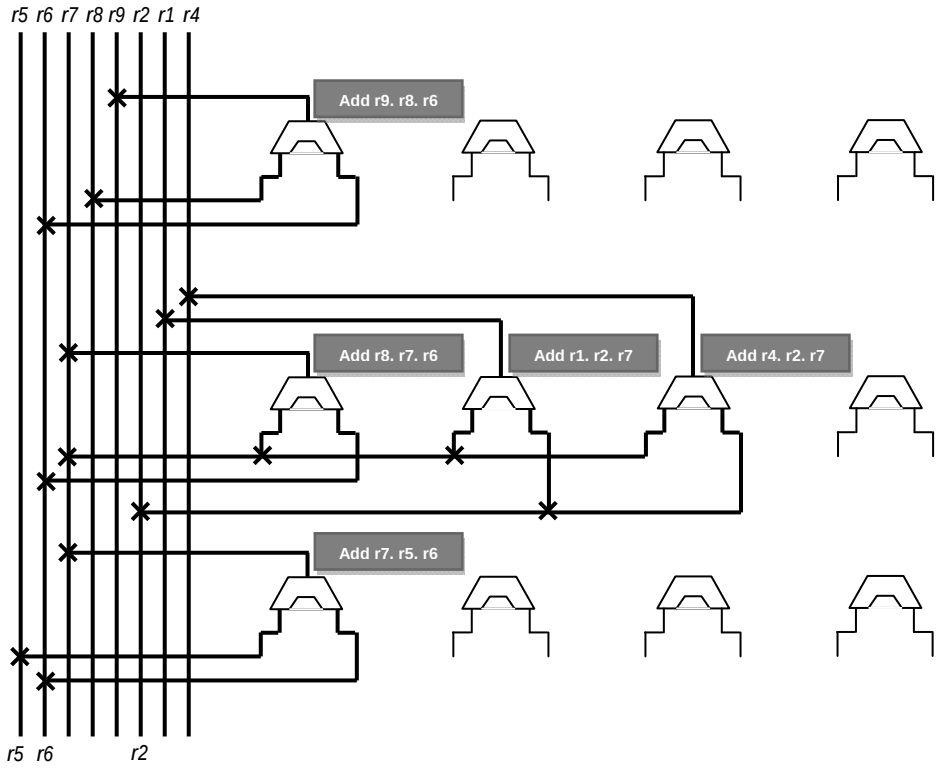


Figure 5.5: The configuration of the array for the previous example

The simplest version of the algorithm has been just explained, in order to facilitate its understanding. However, several improvements were done since the very first version. Although the basic principles of configuration and routing remain the same, the complete version has the additional functionalities that follow (each new functionality discussed is incremental, meaning that the previous one was also considered):

5.2.4 Support for immediate values

Immediate values are allowed for the input context. They are treated as registers in the start table. However, they cannot be changed (so, it is not necessary any entry for them in the current table) and they are fixed (are loaded together with the rest of configuration). It is important to note that, when for the upper bound limit for the input context is reached (for both immediate values or those from the register bank), a new configuration is started.

Now, the follow format is also available:

```
inst op_w, op_r1, (immed) | op_r2
```

These commands are added to the correspondent steps of the algorithm presented before:

1st) Support of immediate operands as the second source operand. An extra bit is included in this field indicating the use or not of an immediate value.

4th)

d) If the `op_r2` is an immediate value, search for it in the immediate table. If there exists one immediate with the same value, point *L2* to it. If it does not exist in the table, generate a write signal and update the *L2* pointer.

5.2.5 Support for different functions in the functional units

A new table is employed, called resource function table, with the same structure (same number of rows and columns) as the resource table. This table stores the information of what operation that functional unit will perform. This way, ALU can perform sums, subtractions, logics and, or, etc.

1st) Also makes the decoding of the type of the function

4th)

e) Update column *C* in row *R* of the resource function table with the value decoded in the first step.

5.2.6 Different groups of functional units

Until this moment, the whole array was composed by the same kind of functional unit: ALUs. Now, it is possible to use different units, although they have to present the same delay as ALU to perform their functions. As already stated before, they are divided in groups of columns, where each column is always homogeneous. For that, the steps one and three are modified:

1st) Include bits in the decode stage to inform what group the incoming instruction belongs.

3rd) The search for the free column *C* changes a little bit. Instead of starting from the begging of the row and finishing at the end of it, the search starts at the column *GS* and finishes at the column *GE*, where *GS* and *GE* are the bits decoded at the 1st stage that hold the information on where the group for that instruction begins and ends in the row, respectively. If there is no resource available in the current row *O*, increment the value of *O* in 1 and repeat the same operation, until finding the resource. In the same manner, the search in the above rows (if necessary) will respect the bounds of the current instruction group, using *GS* and *GE*.

5.2.7 Instructions with different delays

Now, different delays for each functional unit are allowed. Functional units within the same group have the very same delay. This way, the principle of homogeneity of the columns is maintained: each column always has the same type of functional units, which take exactly the same amount of time to perform its function as the others. The resource table remains the same, just those cells above the current functional unit are marked as busy. For instance, if a shifter takes one processor equivalent cycle to perform its function, and the delay of three rows represents this cycle, the two columns above the one already occupied by the shift instruction will also be marked as busy.

Using the same information of group given from the 1st stage, the following changes are necessary:

4th)

b) Besides updating the column *C* in row *R* of the resource table as busy, also update the subsequent columns *C* in row *R + 1 ... RD*, where *RD* is *N – 1*.

N, in turn, is the number of rows necessary for the functional unit in that group to perform its operation. *R* equals to *RD*.

5th)

e) In the write table, write the value of *W* in the row *R*, column *C*. Note that now, *R* could be changed in the previous step according to the delay of the functional unit.

5.2.8 Load/Store Instructions

Memory accesses are allowed. The allocation of these instructions in the array is based on the assumption that the stores can access the same address as previous loads. This way, the allocation is conservative: stores are always allocated after loads. In a more advanced version, however, advanced memory alias analysis can be performed (for instance, the mechanism could have an equivalent of the write bitmap table for memory addresses). The delay of these functional units can be configured according to the number of cycles necessary to access the main memory or cache. If a cache is used, a special mechanism is provided to re-execute the instructions in the array from the beginning, after that cache miss was resolved.

There is a major difference between load/store allocation when comparing other functions with variable delay: they need to start/end exactly in the bounds of the processor equivalent cycle: since memory accesses are still synchronous. For instance, if the current incoming instruction is a load, and three levels in the array are equivalent to a processor cycle, the instructions can only be allocated at the first, fourth, seventh (and so on) rows. This way, the following steps are modified:

1st) Decoding of LD/ST instructions. It is necessary to separate these instructions of the other ones with variable delay, as previously explained.

3rd) If row *R* not equals to *RP,* with $RP = (mod\ P) + 1$, where P is the number of rows in the array which the total delay is equivalent to a processor cycle; *R* equals to *RP*. Column C is not changed.

5.2.9 Write backs in different cycles

An extension of the context table is made. Before, the context table had just two rows: start and current contexts. Now, there is still the start row, but there is a copy of the current context for each row *RP* of the array (the row that corresponds to the beginning of each processor equivalent cycle). This small table says what registers will be written back in that level. The number of simultaneous writes in the register bank is the same as the number of ports that it has. If there is more writes in the current row than it supports, these writes are forwarded to the next level. To make it possible, besides the BT algorithm, multiplexers had to be added to the array's structure. If there are two ports available, two multiplexers in each row *RP* are added, receiving the whole context bus and presenting as output the value to be written back. The steps are added as follows:

5th)

f) Depending on the step 4c – if the pointer *W* is being written in the table, the row *RP* of the context table is updated with *W*. If there is no more slots available in the row *RP*, increment *RP* until finding a slot available for that write.

With that, if there are two writes in the same register within a given configuration, both will be performed, although just the last one is necessary. This way, There is also a comparison in the previous tables: if there is the same W marked to be written in previous rows, it is erased from them.

5.2.10 Handling False Dependencies

Let us consider an example to better illustrate this approach, with the follow sequence of instructions:

```
add r7, r5, r6
sub r5, r9, r6
mul r5, r8, r6
```

Between the *add* and the *sub*, there is a false dependence, named WAR (Write After Read). In this case, the processor could not execute the *sub* instruction in parallel to the *add* because of data coherence: the value of r5 cannot be changed at the same time it is read. Between the same *sub* and the next instruction, *mul*, another type of false dependence occurs, known as WAW (Write After Write), again, with R5. Because of the same reason as before, data coherence, both instructions cannot be executed in parallel because r5 cannot be written at the same time. They are declared as false dependencies because one can apply techniques to avoid them, such as Register Renaming (HENNESSY; PATTERSON, 2003), which is a very expensive process and it has a high cost in any design of a superscalar processor (BURNS; GAUDIOT, 2002).

In the proposed BT algorithm, the context table is altered to easily handle with false dependencies. It has a pointer indicating the last operator included in the context table. When an operation needs this operator, the search occurs from the right to the left, beginning at this pointer. Each new destination operator that it is included, no matter if it is the same, has a new entry in the current table. In the example above, r5 would have 3 entries in the context table. If one considers that between the *add* and *sub* there would have other instructions that would read R5, they would use the first entry (included because of the *add*). When the *sub* instruction itself is found, a new entry with R5 is added. Any instruction between the *sub* and *mul* instruction would use this last R5 entry in the array, because the search occurs from right to the left (from the last to the first). In the same way, any instruction executed after the *mul* would read the last entry of R5, and so on. As a circular buffer is employed, the previous operators that are not used anymore can be overwritten by new ones, when an "overflow" of the context table occurs.

The step 4c need to be changed:

4^{th})

c) Search in the current context table from right to left beginning at *FP*, where *FP* is the pointer indicating the end of the context table. For the *W* point, always generate the signal for including a new entry in the next stage. For each new entry, increment *FP*.

5.2.11 Speculative Execution

Before its implementation, two different ways of performing speculation were analyzed. The first one was the speculative execution of all possibilities inside the array (all possible basic blocks are executed), just writing back the results of the correct path

at the end. To gather the results, the estimator built in the Simplescalar toolset was extended. It has shown that, although performance improvements would be achieved, the area overhead would be huge: the array was growing too much in the horizontal direction. This way, in this case, speculative execution would be just worth if the array was big enough, otherwise a loss in performance could show up: instructions that would be executed but were worthless (in the sense that their results would not be written back) would take place of instructions that would be really valid.

The second approach (the one that was implemented), uses the same principle as trace scheduling: the configurations of the array are indexed by the PC register and the following basic blocks are executed speculatively (however, just one path is considered). If they are not miss predicted, the results are written back. If they are, the results are discarded and the control is given back to the processor, in order to execute these instructions, using its normal flow. The approach is illustrated in Figure 5.6. For this example, it is considered that the saturation point is 2. When the Basic Block 1 (Figure 5.6a) is found, it is allocated in the array as usual (Figure 5.6c). After that, the branch instruction can take two paths: to the Basic Block 2 or to the Basic Block 3. In this example, the path taken was to the Basic Block 3. This way, a variable responsible for that branch is incremented (equals to 1). Next time Basic Block 1 is found, the BT does not need to allocate its instructions (they have already been allocated previously). However, again, it is verified that the same branch has taken the same path as before: to the Basic Block 3. The variable was incremented once more, reaching the saturation point. Consequently, the instructions of the Basic Block 3 are also allocated in the same configuration (Figure 5.6c). On the other hand, if the path taken were to Basic Block 2, instead of 3, the variable would be decremented. Furthermore, if, during execution, the number of miss predictions in sequence equals to the saturation point for a given branch, the following basic block is removed from that configuration, starting the whole process of BT again.

A new group of functional unit in the array was created, composed by branch units. For the write back of results, new multiplexers in each row were added. Without speculation, the array would have a given number of multiplexers per level directly connected to the register bank, in order to write back the results from the context (as stated in section 5.2.9). Now, it has more multiplexers per row, divided in groups. Each group of multiplexers belongs to a given level of speculation in the array. The values of each group of multiplexers are saved in a buffer, waiting for a trigger, correspondent to the level of speculation. When a given branch unit executes the branch instruction relative to that level of speculation, the bit signal is sent, informing if the values waiting for that trigger can be written back or should be discarded.

A special control in each row keeps track of the values that need to be written back, for each group of multiplexers that represent each node (basic block), in the dataflow execution tree. If there is a miss speculation, the array finishes its execution and just writes back the results of the first basic block and the ones which previous branches were speculated right. Then, it sends information to the speculative control and, instead of returning the PC of the last instruction of that configuration, it returns the one correspondent to the beginning of that basic block that was miss speculated, in order to start the execution of the non translated instructions again (taking the right path for that branch).

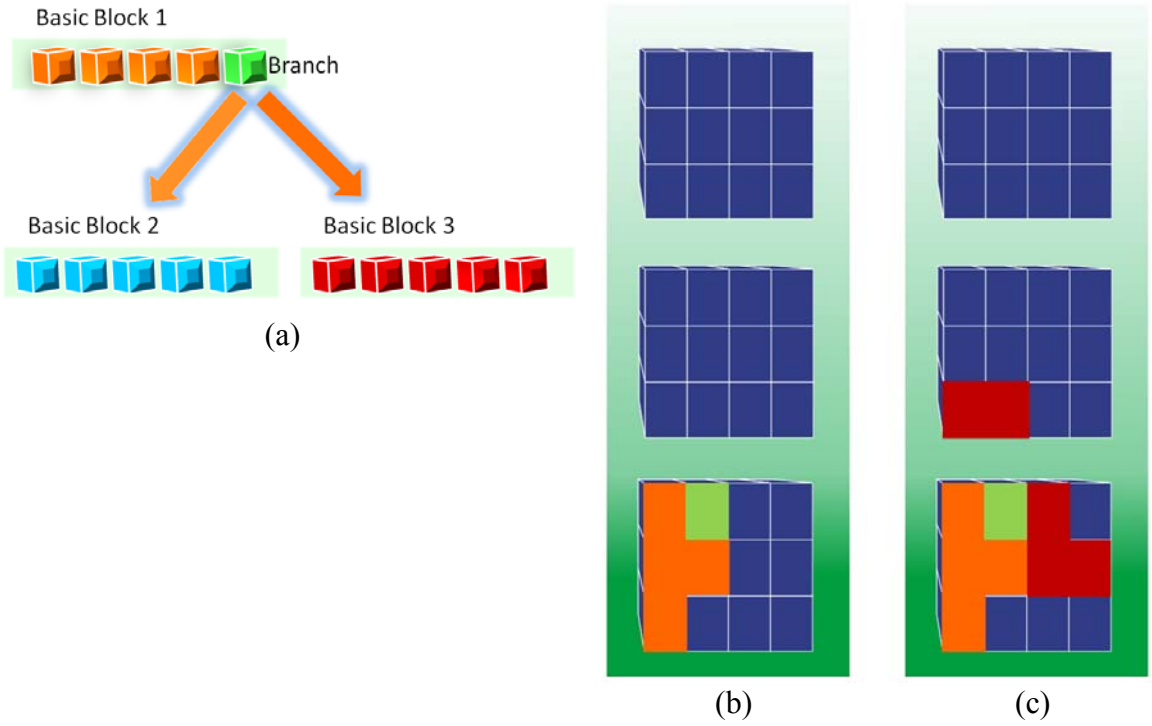


Figure 5.6: How the saturation point works during BT detection for future speculative execution

A new entry in the write table indicating the level of the result being written back was included. Note that the critical part in the implementation of speculative execution is the control of the reconfigurable array. The BT itself needs small changes. The following steps of the algorithms were added/modified:

1st) Also decode branch instructions. For each branch instruction, the variable *L* is incremented in one. *L* represents the level of speculation.

3rd) During the search, consider the branch group.

5th)

a) If *W* needs to be added, include *L* with it.

6 RESULTS

In the first part of this Chapter, results regarding the Java processor with two different versions of the Femtojava processor are demonstrated, executing a diverse range of benchmarks. Then, the results of the proposed approach in a RISC processor are shown, where both the Simplescalar Toolset and the MIPS R3000 are analyzed.

6.1 Java Processors

In this section, it is presented the results and the methodology used for the measurements when coupling the binary translation hardware and the reconfigurable array with the Femtojava Processor. Also, two different set of benchmarks were employed: one composed by simple benchmarks, besides a floating point sum emulation algorithm and a MP3 player; and a sub-set of the SPECJVM (SPEC, 2004). This sub-set was chosen because it represents the large range of different behaviors of the SPECJVM. Two versions of the Femtojava processors were employed – Low power and Multicycle, besides the VLIW version used for comparison purposes.

6.1.1 Femtojava Low-Power with simple benchmarks

The tool utilized to provide data on the energy consumption, memory usage and performance is a configurable compiled-code cycle-accurate simulator, called CACO-PS (BECK; CARRO, 2003). The CACO-PS is a SystemC like simulator that calculates the power consumed based on the switching activity of the system components. This way, it is possible to compare the dynamic power consumed by different versions of the Femtojava processor. In opposite to the next experiments, in these static power is not considered. The estimation for the memory consumption was done based on data available on (PUTTASWAMY et al., 2002). The area was evaluated using the Altera Quartus-II for Windows (ALTERA, 2008) and was computed in number of gates, after synthesis of the VHDL versions of them.

In this experiment (and all concerning the Femtojava processor), it was considered the cache organization as being fully associative, so any address can go to any place in the cache. Also, it is considered that there is no replacement of data at all. It means that, once the configuration is saved in the cache, it stays there until the end of program execution. This is a disadvantage for the proposed technique: even if the configuration is reused just a few times, it can take place of other one that could bring better results. In the future, replacement techniques will be used (it is important to note, however, that replacement techniques are used for RISC processors).

The group of benchmarks is composed by: Sin computation using the CORDIC method, as a representative of arithmetic libraries; sort – bubble, select and quick, in a array of 10 or 100 elements – and search (binary and sequential), used in schedulers; IMDCT (Inverse Modified Discrete Cosine Transformation), as an important part

present in various decompression algorithms, plus more three unrolled versions in order to expose the parallelism; a library to emulate sums of floating point numbers, since the Java processors do not support floating point operations yet; and finally a complete MP3 player that executes 4 frames of 40kbit, 22050Hz, joint stereo. The algorithm is divided in six parts, because of limitations of the simulation tool at that time.

Initially, in Table 6.1 the performance of all the benchmark set in the Low Power architecture and in the different versions of the VLIW is evaluated, and compared to the Java processor coupled to the reconfigurable array. As can be observed in this table, for the VLIW processor, better results are found when unrolled versions of the IMDCT are used (IMDCT u1, IMDCT u2 and IMDCT u3), compared to the non unrolled version. The reason for this is that there are less conditional branches, which reduces the number of cycles lost because of braches miss predictions, and (mainly) because there is more parallelism exposed. On the other hand, algorithms like the floating point sums emulation do not show performance improvements when the number of instructions available per packet in the VLIW grows. This occurs because there is no more ILP available to be explored. Thus, increasing the size of the VLIW packet does not matter at all.

Still in the Table 6.1, in the column Reconfigurable Array – Sequential, it is shown the greatest advantage of using an array with BT to explore every part of the algorithm. Even in algorithms that do not present a high level of parallelism to be explored like the floating point sums emulation, or in the sort or search ones, great gains are achieved. Furthermore, when the VLIW architecture shows good performance boosts in some algorithms, such as the unrolled versions of IMDCT with a high level of ILP exposed, the array presents even better results. Finally, there is the column entitled Reconfigurable Array – Parallel. This column shows results considering that the array also explores the ILP available, executing instructions in parallel. As in the VLIW version (although in different levels), significant improvements were obtained just in a few algorithms, mainly in the unrolled versions of IMDCT. This reinforces the idea of exploring any sequential part of the software, not being dependent just of the parallelism available.

In the second part of the table, data concerning the reconfigurable array is presented. In the first column of this second part, it is shown how many instruction sequences were saved to the cache and were reused in the array. In the second, the number of times that these sequences were reused is demonstrated. As a good example on how the reuse of code is important, let us discuss the sort family of algorithms. When versions that sort 100 elements are executed, more array configurations are reused, bringing an even better result with no area overhead: the number of different reconfigurations and cells in the array do not increase. The next column shows the maximum number of cycles necessary to reconfigure the array from the cache. The forth column exhibits the maximum number of cells that these sequences occupied considering that there is no parallelism available. On the other hand, when parallelism is explored in the array, sometimes more cells are necessary. The last column shows these values.

In Figure 6.1 the energy consumption in the ROM and RAM accesses of the Low-Power version with and without the array are compared against the 4 instruction/packet VLIW version, since the values of energy spent in memory accesses in this VLIW architecture are very similar to the 2 and 8 instructions/packet ones. As it can be

observed, the array saves energy in ROM accesses. Instructions that would be fetched in the memory are instead directly executed in the array, because the dataflow equivalent of this sequence is saved in the reconfiguration cache. In the same way, power consumed in the RAM memory and in the register bank is saved, because now there are a specific cache for loads of static values and the bypass of operands inside the array.

Regarding the energy spent just in the core, presented in Figure 6.2: even with the extra power spent because of the addition of the reconfiguration cache, there are still gains in terms of energy consumption in some algorithms. This occurs because a considerable amount of instructions that would use the five stages of the pipeline of the processor and its sequential logic are now being executed on combinational logic in the array.

Table 6.1: Comparison among different versions of the Femtojava with and without the reconfigurable array

Algorithm	Number of cycles						Data about the array				
	Low-Power	VLIW (instructions per packet)			Reconfigurable Array		#dif. reconf.	#Seq. reused	#max rec.	#max Seq. cells	#max Par. cells
		2	4	8	Sequencial	Parallel					
Sin	755	599	592	583	383	383	8	64	3	2	2
BubbleSort 10	2424	2013	1923	1923	712	600	7	177	3	4	4
SelectSort 10	1930	1689	1689	1689	532	514	8	182	3	3	6
QuickSort 10	1516	1246	1246	1246	496	496	13	132	3	2	2
BuubleSort 100	339797	268610	268610	268610	61541	47840	7	22458	3	4	6
SelectSort 100	134090	127466	127533	127533	30700	30502	8	15280	3	3	6
QuickSort 100	13239	10649	10649	10649	5007	5007	13	2804	3	2	2
Binary Search	403	369	365	365	176	176	5	33	3	2	2
Seq.l Search	1997	1776	1774	1774	658	658	2	253	3	2	2
IMDCT	40306	33128	33071	32994	9399	4287	7	2407	4	10	15
IMDCT u1	31500	18062	12191	9604	7624	2512	16	825	4	10	15
IMDCT u2	30372	17329	11546	9114	6972	2436	13	804	4	10	15
IMDCT u3	18858	11230	9838	7807	2852	2780	7	745	3	4	6
F. P. Sums	14531	12475	12314	12296	6760	6729	37	660	4	3	4
MP3 part 1	242153	210818	200721	183818	103549	102936	140	12317	5	4	6
MP3 part 2	109396	92735	92735	92735	65010	65010	11	8138	3	3	3
MP3 part 3	64488	49346	49346	49346	45525	45525	22	9190	3	2	2
MP3 part 4	41587	33860	34471	31436	22097	22097	5	2876	4	3	3
MP3 part 5	35895	34405	15905	8959	9016	9016	5	1212	3	3	3
MP3 part 6	159017	103441	73482	51124	36405	31485	53	6005	7	11	15

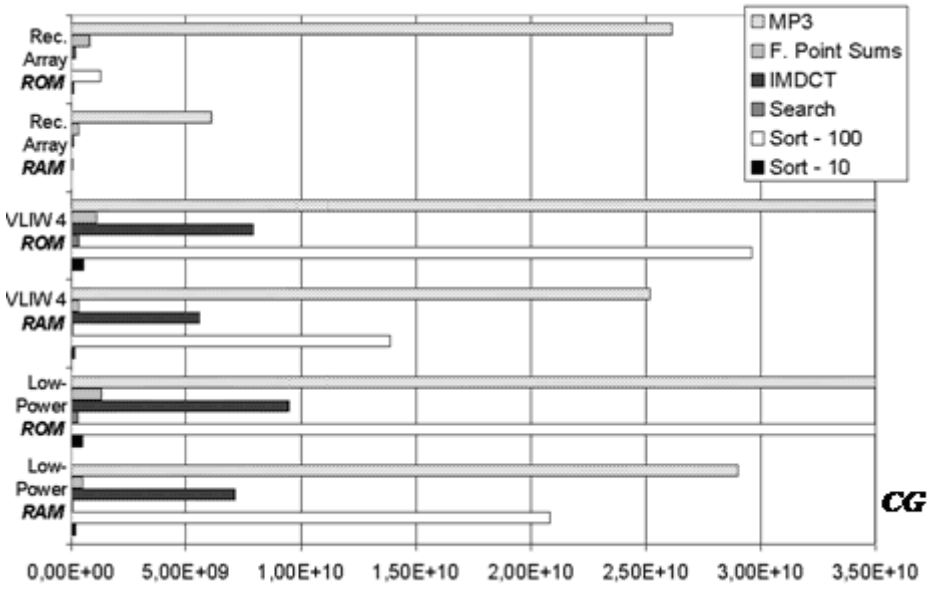


Figure 6.1: Energy spent by RAM and ROM accesses

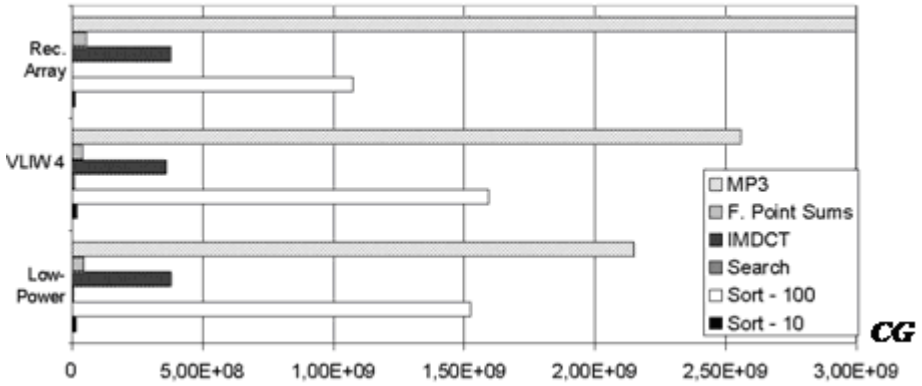


Figure 6.2: Energy spent in the core

Figure 6.3 shows the total energy consumption of the system considering the RAM, ROM, core and the additional BT hardware that makes the dynamic code analysis. It is important to note that great gains were achieved in energy consumption in all algorithms, proving the technique effectiveness.

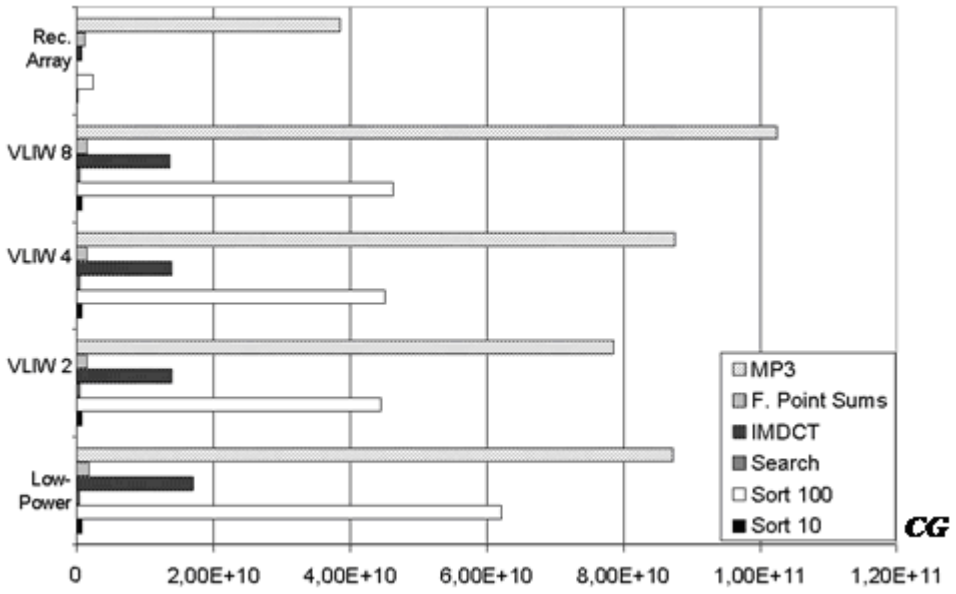


Figure 6.3: Total energy spent by the architectures

Table 6.2 shows the area occupied by the Low Power and VLIW versions of the Java processors. In Table 6.3 it is presented the area occupied by the Low-Power version with different configurations of the reconfigurable array (the maximum number of reconfigurations allowed versus the total number of cells available in the array), counting its cache and the BT hardware. As can be observed in this table, the reconfigurable array, when coupled to the Java processor, even in its simpler version, brings a significant area overhead when compared to the 8 instructions/packet VLIW architecture. However, this was expected, since reconfigurable arrays are very area-intensive due to their great number of functional units. Though counterintuitive, this extra area leads to energy savings, since fewer accesses to program memory and less iterations on the processor datapath are required. Table 6.4 shows the same information as Table 6.3, but in relative numbers.

Table 6.2: Area of the base processors

PROCESSOR	LOW POWER	VLIW (INSTRUCTIONS/ PACKET)		
		2	4	8
Area (gates)	131215	213850	367675	675395

Table 6.3: Area overhead due to the use of the reconfigurable array

# Cells / # Reconf.	2	3	4	7	10
5	757091	993999	1230907	1941630	2652353
10	1039631	1406301	1772971	2872981	3972990
15	1322172	1818604	2315036	3804332	5293628
20	1604712	2230906	2857100	4735683	6614265
40	2734873	3880116	5025358	8461087	11896815

Table 6.4: Relative Area overhead, comparing to the standalone Femtojava Low-Power Processor

# Reconf. \ # Cells	2	3	4	7	10
5	5.77	7.58	9.38	14.80	20.21
10	7.92	10.72	13.51	21.90	30.28
15	10.08	13.86	17.64	28.99	40.34
20	12.23	17.00	21.77	36.09	50.41
40	20.84	29.57	38.30	64.48	90.67

Finally, Table 6.5 compares the Java processor with the reconfigurable array against all other architectures, in terms of energy and performance, considering the best configuration of the array for each benchmark, according to the Table 6.1, Table 6.5 shows how faster the version with the reconfigurable array is, and how much energy it saves. As it can be observed, huge energy savings are achieved when compared to any architecture (10.89 times less energy against the low-power version on the average). There are also meaningful performance improvements even when comparing to the 8 instructions/packet VLIW version (2.77 times faster in the mean).

Table 6.5: Comparing the performance and energy consumption among all the architectures

	Energy				Performance			
Reconfigurable Array vs.	*Low Power*	*VLIW 2*	*VLIW 4*	*VLIW 8*	*Low Power*	*VLIW 2*	*VLIW 4*	*VLIW 8*
Sin	1.89	1.93	1.79	1.83	1.97	1.56	1.55	1.52
Sort - Bubble 10	3.98	4.35	4.21	4.29	4.04	3.35	3.21	3.21
Sort - Select 10	8.09	7.76	7.76	7.91	3.76	3.29	3.29	3.29
Sort - Quick 10	3.19	3.18	3.18	3.24	3.05	2.51	2.51	2.51
Sort - Bubble 100	34.59	19.95	20.04	1.60	7.10	5.61	5.61	5.61
Sort - Select 100	26.23	24.17	24.17	24.67	4.40	4.18	4.18	4.18
Sort - Quick 100	5.74	5.73	5.72	5.82	2.64	2.13	2.13	2.13
Search - Binary	2.00	2.31	2.31	2.35	2.29	2.10	2.08	2.08
Search – Seq.	15.04	16.44	16.45	16.71	3.03	2.70	2.69	2.69
IMDCT	28.33	24.14	24.15	24.65	9.40	7.73	7.71	7.70
IMDCT u1	19.89	16.34	15.70	15.02	12.54	7.19	4.85	3.82
IMDCT u2	21.72	17.93	17.04	16.35	12.47	7.11	4.74	3.74
IMDCT u3	26.68	20.76	21.38	20.24	6.78	4.04	3.54	2.81
F. Point Sums	1.53	1.26	1.25	1.27	2.16	1.85	1.83	1.83
MP3 Part 1	1.87	0.79	0.82	0.86	2.35	2.05	1.95	1.79
MP3 Part 2	3.42	2.74	2.99	3.05	1.68	1.43	1.43	1.43
MP3 Part 3	1.19	1.95	1.95	1.99	1.42	1.08	1.08	1.08
MP3 Part 4	2.71	3.00	2.80	2.84	1.88	1.53	1.56	1.42
MP3 Part 5	4.91	4.67	7.94	13.96	3.98	3.82	1.76	0.99
MP3 Part 6	4.71	5.39	6.34	8.36	5.05	3.29	2.33	1.62
Average	***10.89***	***9.24***	***9.40***	***8.85***	***4.60***	***3.43***	***3.00***	***2.77***

6.1.2 Femtojava Low-Power with SPEC JVM

For the SPECjvm98 benchmark set, a different approach was used, because of the benchmark code size. A trace was generated using the Kaffe Virtual Machine. For

performance and power measurements, an analyzer, which was modified and extended from a previous version used for measurements in the VLIW version of the processor, was employed (BECK; CARRO, 2004). The measurements were based on the same principle of the so-called instructions simulators (TIWARI et al., 1994), where there is a table with performance and power consumption data for each instruction.

The SPECjvm98 suite was developed to evaluate the hardware and software aspects of the JVM client platform providing different tests derived from real applications that are commercially available. For this simulation, all floating-point operations were converted to integer ones since the Femtojava does not support a floating-point unit. Additionally, the benchmarks can also be executed with three different input sizes. All evaluations here presented are based on the traces obtained from the smaller input size, the *s1* data set. It is important to note that this small amount of data is a disadvantaged scenario: it is very likely that there is less repeated code to be mapped to the array when comparing to the bigger input data sets.

Firstly, Figure 6.4 demonstrates the different performance improvements of the employed benchmark set when comparing the Femtojava Low-Power with the array against the standalone processor. Besides, it is considered that all configurations are available in the cache and that there is no limit to the number of cells that can be implemented (both in sequence as in parallel). This number represents the maximum theoretical speed up.

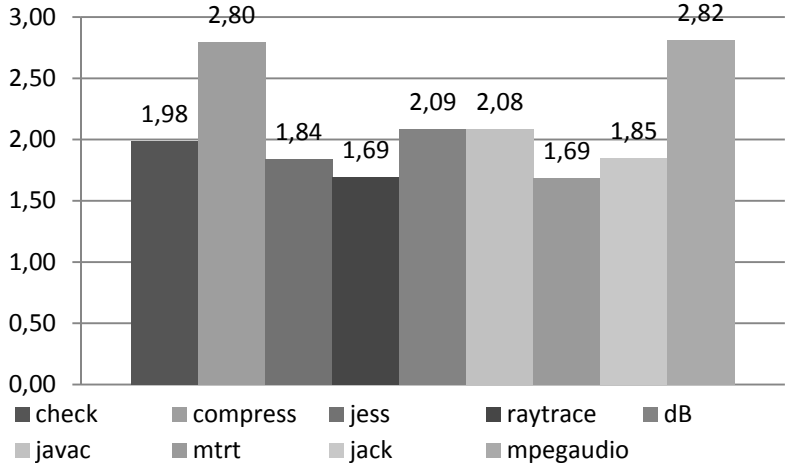


Figure 6.4: Performance improvements - JVMSPEC

Now, the analysis is limited to:

- The cache size was varied from 4 to 32 different configurations;
- The number of available cells implemented in the array was varied from 1 to 5 cells, which sums up a variation from 1 to 5 multipliers and 3 to 15 ALUs in series.

Increasing the number of cells or the cache size would not show a large improvement when comparing to the configuration with 32 cache slots and 5 cells, since a saturation in terms of optimization begins to occur.

In Figure 6.5 the same results presented in the previous figure are shown, only now the number of cells is limited to 5 and only the 32 most executed code sequences are saved in the cache. This figure shows as reference the best performance case with

the proposed restrictions. The consequence of varying the other parameters is analyzed in Figure 6.6 and Figure 6.7.

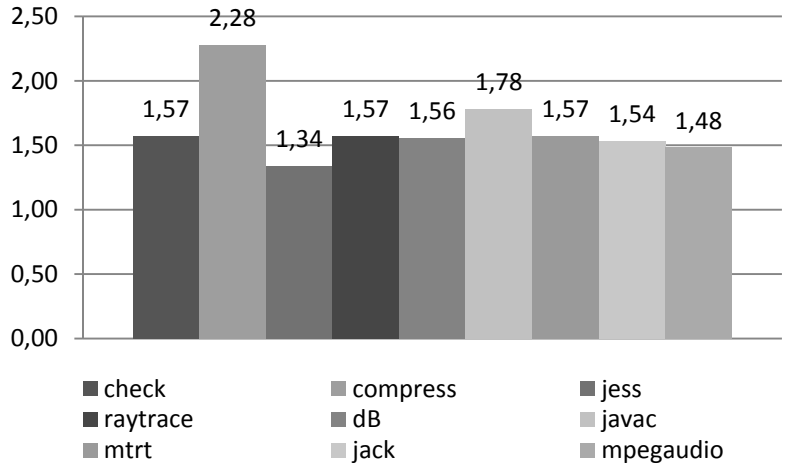


Figure 6.5: Performance improvements with restricted resources

As it can be observed from Figure 6.5 and comparing to Figure 6.4, the performance degradation is not significant in the majority of the whole set of algorithms – even with those restrictions and without using a cache policy that could improve results allowing more configurations to be reused. Nevertheless, it is notable a more important degradation of performance in the *mpegaudio* application. This can be attributed to the characteristics of how this code was programmed, as it will be discussed later. In Figure 6.6, the effect of varying the cache size while maintaining the number of implemented cells in 5 is analyzed. This graph clearly shows the dependence of performance improvements with the number of configurations available in the cache. For all algorithms, a greater number of available configurations contributed to performance increases. Using 32 configurations it is shown that, in the mean, almost 80% of the theoretical speedups is achieved.

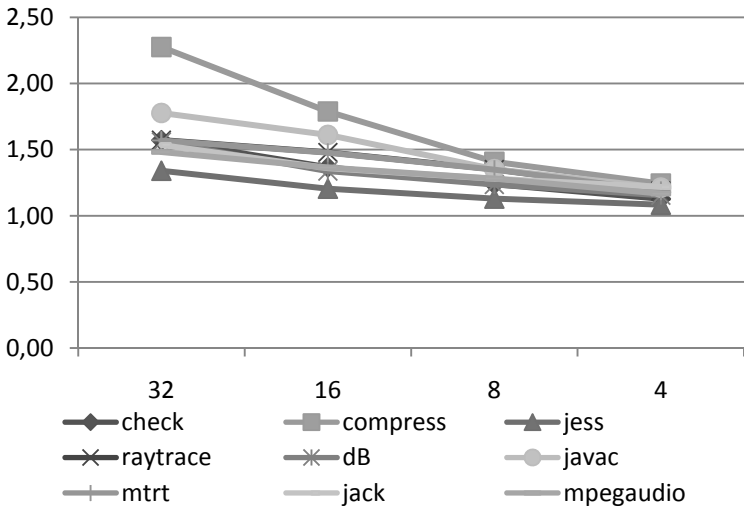


Figure 6.6: Performance improvements when varying the total number of configurations used

It has to be pointed out that algorithms such as *compress* do benefit from an increased number of used configurations. This demonstrates that this application has a

great number of code sequences that are highly reused. In the same manner, the effect of varying the number of available cells while maintaining the cache size holding 32 configurations is shown in Figure 6.7. While the previous figure showed a dependence of performance with the available number of configurations, this one shows that most algorithms do not actually benefit from having more than 2 cells in sequence. By consequence, this fact also demonstrates that normally the size of the most used operand blocks is not that large. Then again, the exception is made with the *mpegaudio* application. In this program, there exists big operand blocks that are highly executed. Therefore, the application would take benefit from having more available cells in the reconfigurable array.

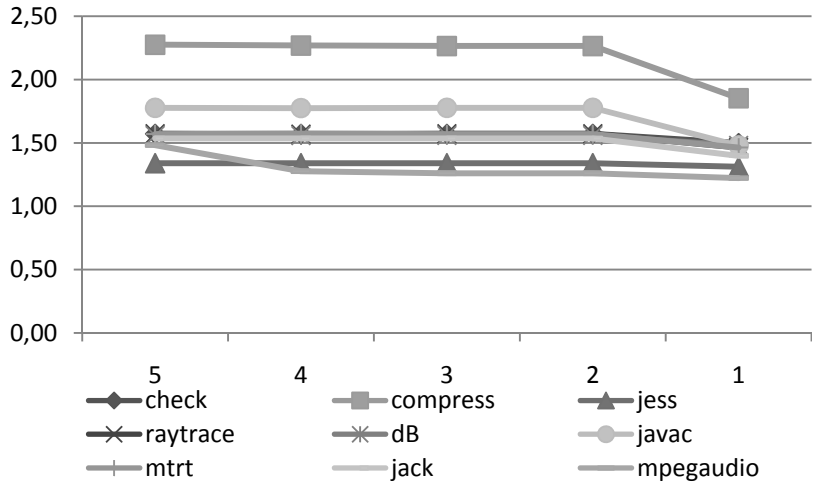


Figure 6.7: Performance improvements. Now, varying the total number of cells available in the array

When evaluating the impact of varying the parameters regarding energy consumption (array size and number of different configurations) it is important to observe that they directly influence the cache size. A bigger cache size could mean that more energy will be spent when reading and writing to this memory. On the other hand, increasing the cache size also could mean that more instructions and operands will be kept and, consequently, less reads to the RAM and ROM are necessary, thus saving energy. The energy tradeoff analysis regarding this discussion is primarily presented in Figure 6.8, while the table showing the energy gains for different configurations is shown in Table 6.6. As it can be seen, the impact of a bigger cache does not strongly influence the energy savings. The importance of saving RAM and ROM accesses is clear, as the increased number of available configurations increases energy savings. Once again, it is important to highlight that no cache policy is being used. Therefore, there is a potential for saving even more energy, since, with that, more configurations could be saved in the cache without increasing its size.

Figure 6.9 shows the effect of varying the number of implemented cells on the overall energy savings. Once again, having more than 2 cells also did not show much improvements for most benchmarks with the exception of the *mpegaudio*. It is also visible that energy gains start to decrease when implementing more than 3 cells for most algorithms. This happens due to the increased cache size that is indeed not being fully used.

Table 6.6: Energy savings with different configurations

#cells	#conf.	check	compress	jess	raytrace	dB	javac	mpegaudio	mtrt	jack
5	32	30,10%	45,97%	19,14%	39,65%	36,36%	44,51%	38,70%	39,81%	16,73%
5	16	23,92%	31,55%	11,26%	37,20%	26,87%	41,93%	28,74%	37,35%	12,09%
5	8	20,26%	25,17%	6,43%	27,78%	19,15%	24,88%	17,96%	27,91%	9,06%
5	4	14,53%	17,43%	3,15%	13,59%	12,81%	13,90%	15,77%	13,65%	6,16%
4	32	30,60%	45,96%	19,13%	39,65%	36,37%	44,54%	35,84%	39,72%	16,74%
4	16	24,12%	31,57%	11,25%	37,20%	26,87%	41,95%	26,42%	37,27%	12,11%
4	8	20,33%	25,16%	6,41%	27,78%	19,15%	24,91%	19,97%	27,83%	9,09%
4	4	14,55%	17,43%	3,13%	13,57%	12,81%	13,93%	13,74%	13,60%	6,18%
3	32	31,10%	46,00%	19,14%	39,65%	36,37%	44,51%	34,45%	39,81%	16,73%
3	16	24,32%	31,58%	11,26%	37,20%	26,87%	41,93%	26,10%	37,35%	12,09%
3	8	20,41%	25,18%	6,43%	27,78%	19,15%	24,88%	19,66%	27,91%	9,06%
3	4	14,57%	17,44%	3,15%	13,59%	12,81%	13,90%	13,43%	13,65%	6,16%
2	32	30,39%	45,55%	18,19%	37,58%	34,48%	38,09%	33,82%	37,62%	16,38%
2	16	23,89%	31,12%	11,27%	34,79%	26,89%	35,70%	26,72%	34,82%	12,10%
2	8	20,49%	24,71%	6,44%	27,83%	19,17%	24,60%	19,31%	27,85%	9,08%
2	4	14,59%	17,00%	3,16%	13,61%	12,83%	13,96%	13,08%	13,61%	6,18%
1	32	30,79%	37,47%	18,12%	34,44%	31,93%	35,69%	33,10%	34,52%	16,86%
1	16	24,60%	28,18%	12,04%	31,69%	25,64%	33,67%	26,61%	31,76%	12,16%
1	8	20,56%	18,73%	6,47%	27,70%	19,21%	24,20%	20,57%	27,76%	10,11%
1	4	14,60%	14,05%	3,19%	13,58%	12,86%	14,83%	13,12%	13,62%	7,58%

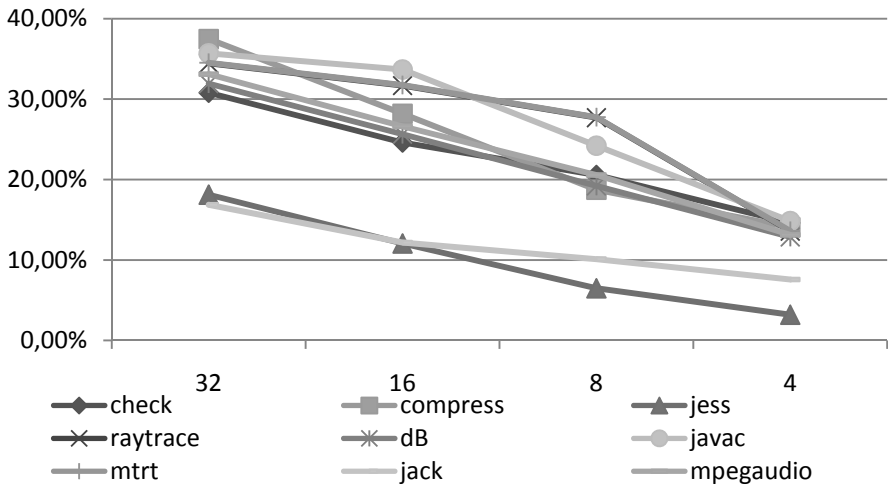


Figure 6.8: Energy savings varying the number of allowed configurations

The different characteristics found in the *mpegaudio* application can be justified on how it was programmed. It is a totally data-flow algorithm, which performs intensive mathematical operations that can easily be mapped to a great number of available units in the array. Moreover, this particular implementation of the *mpegaudio* application, as it is described in the SPEC documentation, has little amount of garbage collection. This demonstrates that the algorithm is not very object-oriented. It is interesting to note that less object orientation also favors the use of the reconfigurable array since operations such as *new* as well as method invocations are computational intensive and cannot be mapped. This same reason also explains more modest results for performance gains and energy savings in the *jess* and the *jack* algorithms: in these applications new objects are continuously allocated. The effect of object allocation operations and other Java particular operations that cannot be executed on the array will be analyzed in the next section.

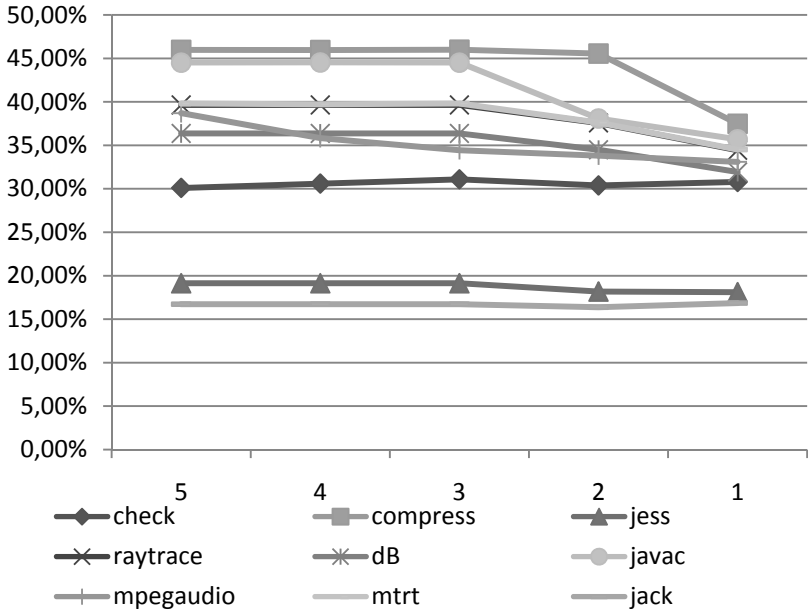


Figure 6.9: Energy savings when varying the maximum number of cells available in the array

Finally, to illustrate the effect of simultaneously varying both parameters (cache and size of the array), Figure 6.10 and Figure 6.11 show two 3D graphs for performance improvements and energy savings of the *compress* algorithm that represents the general behavior of most applications of the SPECjvm98 benchmark.

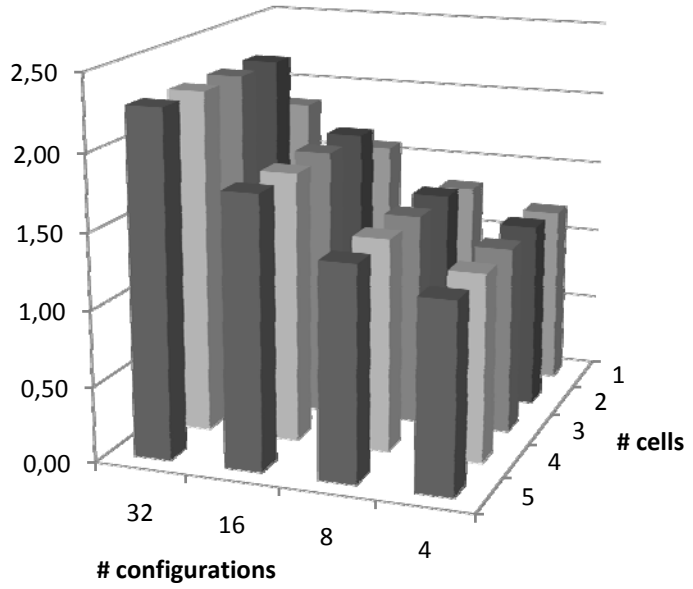


Figure 6.10: Performance improvements when varying both parameters for the compress algorithm

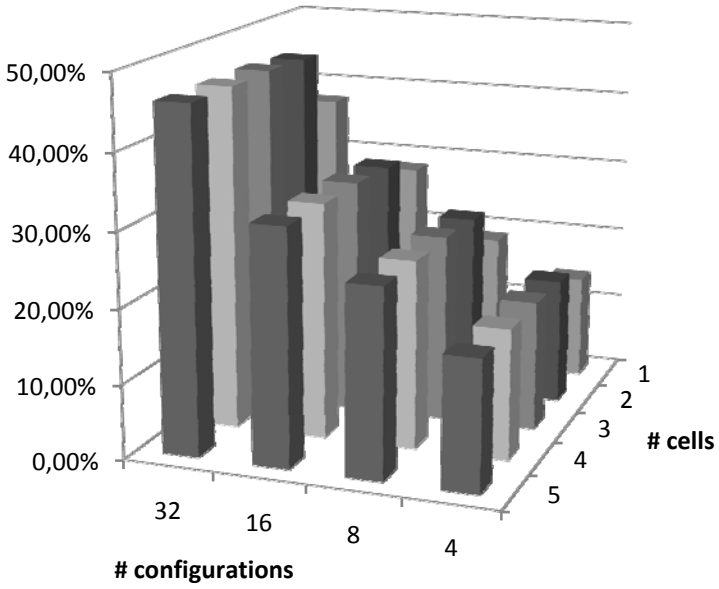


Figure 6.11: Energy savings achieved when varying both parameters for the compress algorithm

6.1.3 Femtojava Multicycle with SPEC JVM and others

Now, the Femtojava Multicycle with the reconfigurable array is compared against its standalone pipelined version. The objective here is to demonstrate the potential of the array when coupling it with a very simple processor: gains are shown even when comparing to a more powerful processor. This way, it is possible to save area and design time, concerning the main processor.

The results are supported by simulation, using the same methodology presented in the previous section. The only difference is that Synopsis Power Compiler (SYNOPSYS, 2006) was used for dynamic and leakage power consumption computation. All results were based on the TSMC 0.18 technology. Data about power consumption in the main memory comes from (PUTTASWAMY et al., 2002). A mix of the previous algorithms employed before was used for this experiment: sort and search algorithms, IMDCT (plus three unrolled versions) and the MP3 player. The second group is a subset of SPECjvm98 package.

In Figure 6.12 and Figure 6.13 the performance improvements in these two separated groups are shown. In the X-axis the number of cells available in the array is varied, while the Y-axis presents a normalized value that compares the results of the Femtojava Multicycle with the reconfigurable array against the Femtojava Low-Power, where the value one is its own performance. As can be observed, as more cells are available, more performance gains are achieved. Depending on the algorithm, however, as no speculative execution is done, there is a limit in the optimization. This limit is exactly the average size of the basic blocks of the application. In all the following presented results, in order to limit the design space, a reconfiguration cache of 8 slots is used, since increasing the number of slots available up to 512 would bring a performance increase of just 10%, on average.

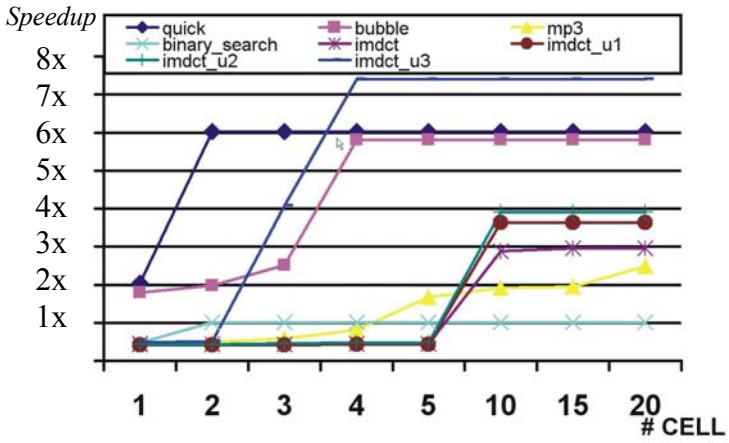


Figure 6.12: Performance improvements, in simple applications, when increasing the number of cells of the reconfigurable array

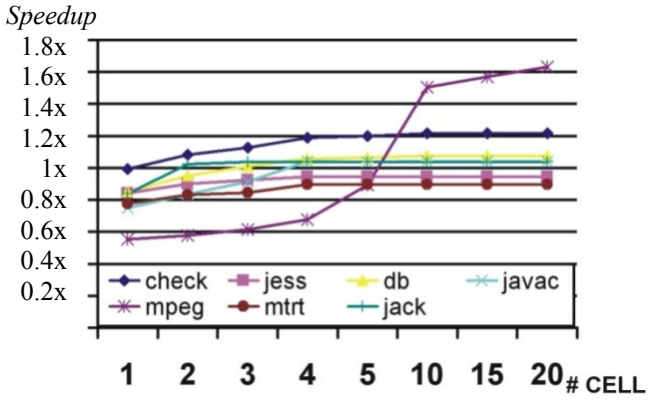


Figure 6.13: Same as the previous, but now executing a subset of the SPECjvm98

It is important to point out that Java is an object oriented language. This way, as already commented before, depending on the level of object orientation employed and style of programming, it is possible to find a huge number of instructions for the manipulation of these objects. As these instructions are complex and divided in several microoperations, taking more than one cycle to be executed, they cannot be optimized by the proposed approach. In Table 6.7 the percentage of cycles (Control Cycles) spent for this kind of instruction considering the whole program execution is shown. It is important to note that almost any instruction in RISC processors could be executed in the combinational logic because, opposite to CISC-based native Java bytecodes, RISC instructions are simple and do not have any kind of instruction that involves complex microcoding. The same table also explains the reason why some algorithms do not show any performance improvements in Figure 6.13. They are exactly those that have a huge percentage of instructions that cannot be optimized.

Figure 6.14 and Figure 6.15 present energy savings because of the reconfigurable system. As stated before, there are three main reasons for these: execution of large sequences of instructions in pure combinational logic, instead of using all the structure of the processor; the avoidance of repeating the dependence analysis again and again for the same sequence; and, since this information is kept inside the processor in a special

cache, the number of accesses to the instruction memory decreases considerably. This way, even if the size of the core increases, because of extra hardware for the detection, extra combinational hardware and the special cache memory, great energy advantages still appear.

Table 6.7: Percentage of cycles regarding instructions that cannot be optimized

	Total Cycles	Control Cycles	%
check	4 887 426	3 184 402	65.15
jess	39 394 726	28 252 774	71.72
db	7 751 709	5 036 323	64.97
javac	34 641 256	19 921 273	57.51
mpegaudio	327 158 420	82 163 257	25.11
mtrt	340 980 938	242 897 076	71.23
jack	465 479 393	354 870 998	76.24
quick	1 766	905	51.25
bubble	3 641	1 545	42.43
mp3	13 231 903	5 227 433	39.51
binary	518	294	56.76
imdct	82 187	43 224	52.59
Imdct1	73 123	28 538	39.03
Imdct2	69 658	26 644	38.25
Imdct3	36 383	14 922	41.01

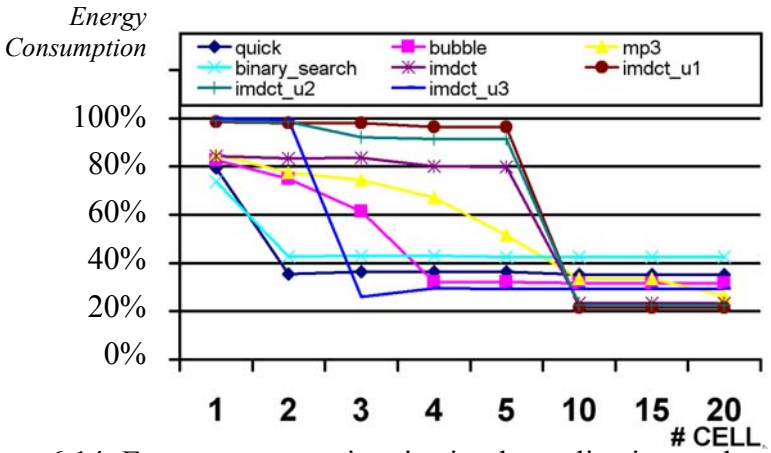


Figure 6.14: Energy consumption, in simple applications, when increasing the number of cells of the reconfigurable array

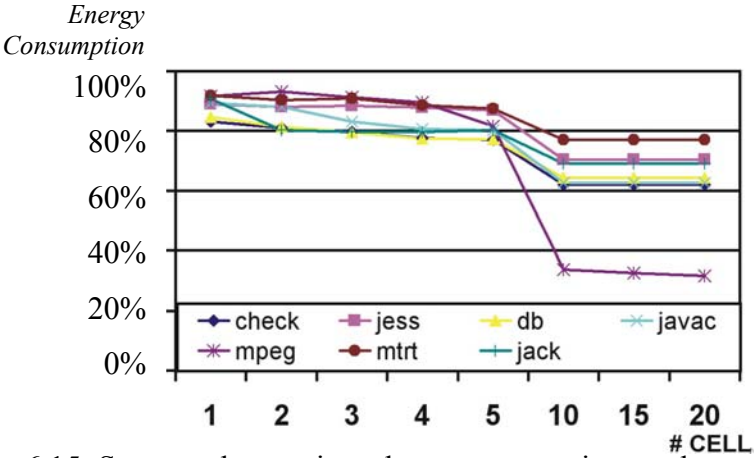


Figure 6.15: Same as the previous, but now executing a subset of the SPECjvm98

shows the area overhead when the Femtojava Low Power is compared against the Femtojava Multicyle with the reconfigurable architecture, considering different number of cells that compose the array and different configurations that is supported in the special cache. As can be observed, the reconfigurable system, even in its simpler version, shows a considerable area overhead.

Table 6.8: Additional area overhead, in number of gates, when compared to the Femtojava Low-Power processor

# Cells / # conf.	1	2	3	4	5
4	84 929	99 325	113 722	128 119	142 516
8	92 503	106 899	121 296	135 693	150 090
16	105 318	119 714	134 111	148 508	162 905
32	133 155	147 551	161 948	176 345	190 742
64	200 265	214 661	229 058	243 455	257 852

Finally, in Figure 6.16 a comparison is done between two very different benchmarks: one very control flow oriented (*mtrt*), and the other one mostly dataflow (*mpegaudio*), with 1 or 20 cells available in the array for optimization (x-axis). The objective is to show that, besides speeding up the execution of dataflow algorithms, as expected, it is also possible to increase the performance of control flow programs. The same Figure presents the execution time improvements when considering all instructions executed, and when considering just the set of instructions passive of optimization, as discussed before.

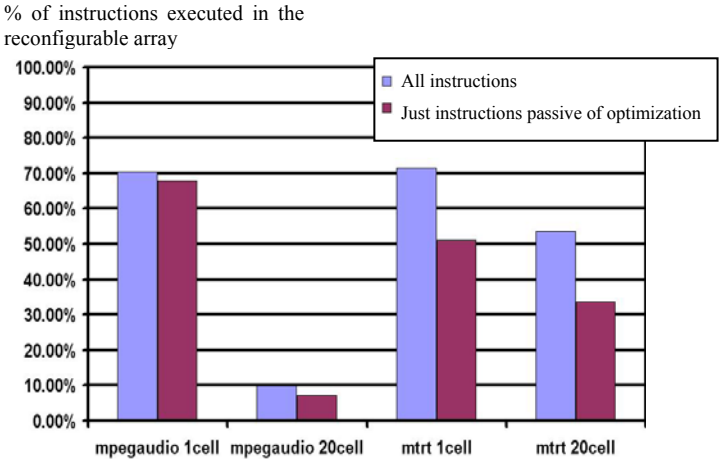


Figure 6.16 – Performance improvements in both control and data flow oriented algorithms

6.2 RISC Processors

In this section, it is analyzed the RISC implementation of the reconfigurable array. Firstly, it was simulated using the Simplescalar Toolset. This way, it was also possible to compare the performance of the reconfigurable system against an Out-of-order superscalar processor. Then, results considering the coupling of the BT and

reconfigurable array with a MIPS R3000 are shown – representing the embedded system market.

6.2.1 Simplescalar

For the performance analysis, an performance estimator was built and integrated to the simplest version of the Simplescalar toolset (BURGER; AUSTIN, 1997): sim-safe. It was implemented in C (the language was used because the Simplescalar itself also was programmed in C), and for performance measurements, it follows a very similar approach that Tiwari et al. (TIWARI et al., 1994) uses. This estimator analyses instructions at run time, during the execution of the software. As it is totally integrated to the Simplescalar toolset, one can change some parameters, as the number of functional units of the array, number of lines, columns, delay of each functional unit etc. In addition, it provides the statistics considering different reconfigurable cache sizes, and gives the average time spent by several operations, as execution of the sequences, context load time, write back time etc. It is important to point out that a low-level description of the algorithm has been developed, and it is aimed to be integrated with the most complicated version of the simplescalar toolset.

In this first experiment, the Simplescalar ToolSet was configured to use the PISA architecture (which is based on the MIPS IV ISA) and to behave like an ordinary in-order MIPS processor (very similar to the MIPS R3000 processor), executing a subset of the MiBench (GUTHAUS, 2001), with the follow algorithms: *Basicmath*, *Bitcount*, *Qsort*, *Tiffdither*, *Tiffmedian*, *Dijkstra*, *Patricia*, *Ghostscript*, *StringSearch*, *Sha*, *CRC*, *FFTinv* and *FFT*. This subset of benchmarks has a different average of branches per instruction. This way, the behavior of the approach in algorithms that are more control or dataflow oriented can be better analyzed. It is considered a memory where it is possible to make two reads and one write per cycle and a latency of one cycle to fetch values from the cache. This assumption is in somehow very pessimistic. For instance, the authors in (GONZALEZ et al., 1999) considered for trace reuse the capability to perform 16 reads+writes per cycle, including register and memory values. Developed architectures such as the Alpha 21264 (KESSLER, 1999) can perform up to 14 accesses per cycle (8 register reads, 4 register writes and 2 memory references). Therefore, the employed configuration could be easily implemented in nowadays memory systems.

Figure 6.17 shows the performance improvements over a in-order MIPS based processor, where the Y axis is the relative time spent by the algorithm according to the size of the reconfiguration cache, shown in the X axis (where zero means not using the reconfigurable array). It is considered that the array is always big enough to execute the largest configuration found. Analyzing the figure, one could notice that depending on the algorithm, a small number of cache slots is enough. As the cache replacement policy implemented for this analysis was FIFO (First In, First Out), this cache must be large enough to support all the basic blocks that are being executed inside a determined period of time in order to allow their reuse. For instance, consider that an algorithm is composed by a main loop and inside this loop there are five basic blocks. If there are four slots available in the cache, the first time the first basic block will be reused (in the second iteration of the loop), it will not be in the cache anymore. This was, all the detection process should be done again. Therefore, in this case, no optimization would be achieved.

Figure 6.18 shows the average gain (Y axis) concerning all algorithms for each different cache size (X axis). Depending on the this size, the algorithms can be executed

up to three times faster. To show the potential of the technique in this processor, the ideal curve represents the performance gain when considering 1 cycle per merged instruction executed. It is important to point out that in this experiment there was no exploration beyond basic blocks. It is common sense that in order to achieve higher performance improvements this exploration should be done, and that is way the overspread superscalar processors use aggressive speculation to increase even more the instruction level parallelism.

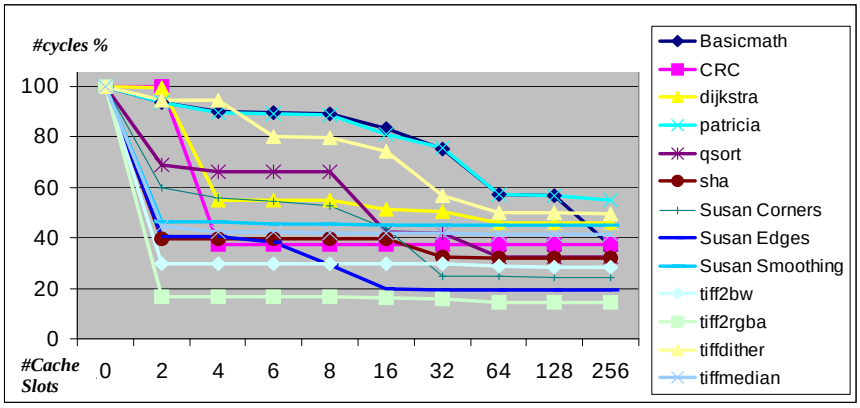


Figure 6.17: Performance Improvements using Dynamic Merging and the Reconfigurable Array

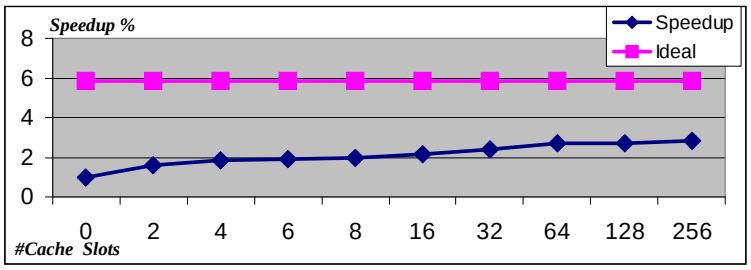


Figure 6.18: The average of the performance improvements considering the size of the cache

This way, the next experiment was adding the capability of performing speculative execution in the array. Moreover, the Simplescalar was configured to behave as close as possible to the Superscalar Out-Of-Order MIPS R10000 processor, for performance comparisons. Its configuration is summarized in Table 6.9. More details about it can be found in Appendix A, at the end of this thesis, where the configuration file employed is shown.

Table 6.9: Configurations of the superscalar processor

Out of Order	
Fetch, decode and commit	up to 4 instructions
Register Update Unit	16 entries
Load/Store Queue	16 entries
Functional Units	2 Integer ALU, 1 multiplier, 2 memory ports
Branch Predictor	Bimodal/512 entries

Table 6.10 shows three different configurations for the array that was employed in the experiments. The last configuration was used in order to try to figure out what is the real potential of the proposed technique. For each array configuration, the size of the reconfiguration cache is also varied: 2 to 512 slots, using the FIFO policy. The impact of doing speculation is evaluated considering optimization of up to three basic blocks ahead. Finally, the cache memory was increased in order to achieve almost no cache misses, so it was possible to evaluate the results without the influence of it. It is important to stand out that the impact of both misses of the reconfiguration cache as the speculation are considered for the simulation.

Table 6.10: Configurations of the array

	Reconfigurable Array		
	C #1	C #2	C #3
#Lines	27	54	99
#Columns	11	16	30
#ALU / line	8	8	11
#Multipliers /	1	2	3
#Ld/st / line	2	6	8

Table 6.11a shows the IPC of the out-of-order processor cited before. This table can be used to compare the IPC of this processor against the IPC of the instructions that are executed inside the array, in different configurations, which is shown in Figure 6.19. For each configuration, three different speculation policies are demonstrated: no speculation, 1 and 2 basic blocks ahead. The number of slots available in the reconfigurable cache was also changed (4, 16, 64, 128 and 512). The four benchmarks presented in this figure were chosen because they represent a very control-oriented algorithm, a dataflow one and a midterm between both, plus the CRC, which is the biggest benchmark in the subset. In Table 6.11b the benchmarks are classified according to the average number of branches per instructions.

Table 6.11: IPC in the Out-of-Order processor and the average BB size

(a)

Algorithm	IPC - Out-of-Order
Basicmath	1.43
CRC	2.13
dijkstra	1.76
Jpeg decode	1.86
patricia	1.40
qsort	1.79
sha	1.94
stringsearch	1.60
Susan Smoothing	1.64
Susan Corners	1.83
tiff2bw	1.90
tiff2rgba	1.92
tiffdither	1.56
tiffmedian	1.91

(b)

BB size
5.8751
7.9954
5.6011
6.2554
4.4255
4.6243
7.9381
4.8709
15.8098
13.4952
22.5567
13.4952
18.9188
30.686

As it is shown in Figure 6.19, it is possible to achieve a higher IPC when executing instructions in the reconfigurable array in comparison to the out-of-order superscalar processor, in almost all variations. However, the overall optimization when using the proposed technique depends on how many instructions are executed in the reconfigurable logic instead of using the normal flow of the processor. Table 6.12

shows the overall speedup obtained when coupling the reconfigurable array to the out-of-order processor against the standalone out-of-order. It is important to notice that reconfigurable systems in general can just show improvements when the programs are very dataflow oriented. The proposed technique, on the other hand, can optimize control and data oriented programs, as it can be observed by the results.

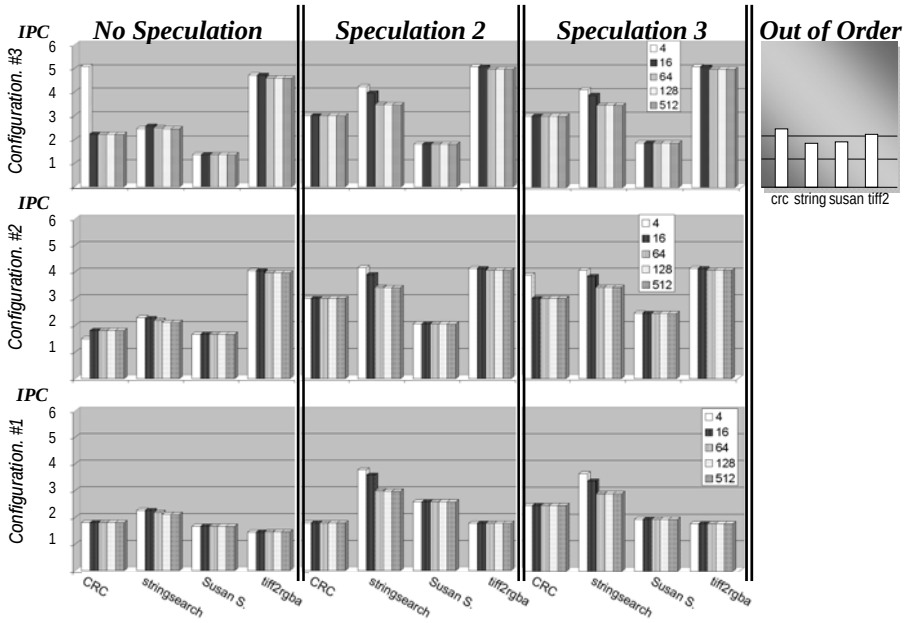


Figure 6.19: IPC of four different benchmarks being executed in the reconfigurable logic with different configurations

Table 6.12: Speedups using the reconfigurable array coupled to the out-of-order processor

Algorithm	#Cycles in the Out-Of-Order	% of Speed Up - Out-of-Order coupled to array with configuration 1									% of Speed Up - Out-of-Order coupled to array with configuration 3								
		No Speculation			Speculation 2			Speculation 3			No Speculation			Speculation 2			Speculation 3		
		4	64	256	4	64	256	4	64	256	4	64	256	4	64	256	4	64	256
Basicmath	111169924	5.03	13.75	17.85	3.52	14.49	21.79	3.40	15.22	23.31	5.76	19.27	26.40	4.63	19.83	30.33	4.86	20.52	32.14
CRC	399531928	-16.01	-16.03	-16.03	-5.20	-5.21	-5.21	9.03	9.03	9.03	3.97	3.97	3.97	8.12	8.14	8.14	20.75	20.77	20.77
dijkstra	31094638	-22.29	-24.31	-24.33	1.30	1.25	1.25	8.45	8.46	8.46	-21.96	-20.08	-20.04	1.00	4.34	4.36	4.13	7.65	7.67
Jpeg decode	3942226	-9.15	-9.72	-9.77	4.63	3.24	3.29	7.11	7.45	7.61	9.76	11.92	12.05	16.55	18.94	19.06	16.77	19.51	19.68
patricia	95927575	4.41	13.30	13.72	3.99	14.42	21.52	3.26	14.22	21.96	5.06	17.97	18.89	5.25	18.80	29.07	4.57	18.58	29.80
qsort	23435690	-8.76	-11.69	-11.69	-0.58	4.18	4.18	0.37	-30.41	-30.21	24.29	38.95	38.95	16.79	43.74	43.74	16.44	40.72	40.72
sha	6800950	11.56	13.07	13.07	27.22	33.45	33.45	26.30	31.29	31.29	22.57	25.48	25.48	39.91	48.66	48.66	41.27	50.28	50.28
stringsearch	115917	16.32	20.16	21.23	28.95	35.20	35.24	28.50	35.39	35.38	21.02	27.05	30.57	31.25	41.02	41.17	31.04	42.61	42.63
S. Smoothing	15628090	-0.94	-3.22	-3.22	0.31	-0.99	-1.00	2.13	1.59	1.59	25.35	35.66	35.69	26.87	37.95	37.96	23.73	32.05	32.04
S. Corners	533870	2.16	1.79	1.79	4.40	4.29	4.28	1.13	4.29	4.28	32.69	41.44	41.44	37.53	41.44	41.45	33.89	37.13	37.12
tiff2bw	27391803	-4.24	-4.38	-4.42	0.88	0.82	0.82	-0.20	-0.20	-0.20	-5.65	-5.42	-5.39	19.08	19.60	19.60	24.41	25.22	25.22
tiff2rgba	23796384	-10.94	-11.39	-11.40	-1.53	-1.75	-1.75	-1.19	-1.39	-1.40	57.19	57.83	57.83	58.29	59.69	59.69	47.30	48.87	48.87
tiffdither	188757828	1.48	8.88	8.92	6.65	9.34	9.41	4.47	-21.46	-23.52	4.33	18.15	18.30	10.73	19.33	19.57	7.95	14.31	14.60
tiffmedian	93254386	3.95	3.74	3.73	12.91	12.82	12.82	7.42	7.38	7.38	14.13	14.11	14.13	27.23	27.43	27.43	27.36	27.72	27.72

6.2.2 MIPS R3000 Processor

For this experiment, an improved VHDL version of the Minimips processor (MINIMIPS, 2008), which is based on the R3000 version, was employed. For area evaluation, again, it was used the Mentor Leonardo Spectrum (LEONARDO, 2008) and, for power estimations, Synopsis PowerCompiler (SYNOPSYS, 2006), both with the TSMC 0.18u library. Estimates on both processor and reconfigurable cache were

done using these tools. Data about power consumption in the main memory was taken from (PUTTASWAMY et al., 2002).The system was evaluated with the Mibench Benchmark Suite (GUTHAUS et al., 2001). All benchmarks with no representative floating point computations and that could be compiled successfully to the target architecture were utilized.

Firstly, the information given in the in the Chapter 3, section 3.2.1, is repeated in Figure 6.20 to reinforce two different concepts: algorithms can be control or dataflow oriented; and they can have few or a large number of distinct kernels subject of optimization. In Figure 6.20b it is characterized the algorithms regarding the number of instructions executed per branch (classifying them as control or dataflow oriented). Figure 6.20a shows the results of the investigation on the number of BBs responsible for a certain percentage of the total number of basic block execution figures. It is convenient to remember that, as more dataflow and as fewer distinct kernels an algorithm has, the better for conventional reconfigurable systems. This fact, on the other hand, is not a limiting factor for the proposed approach, as it will be shown later.

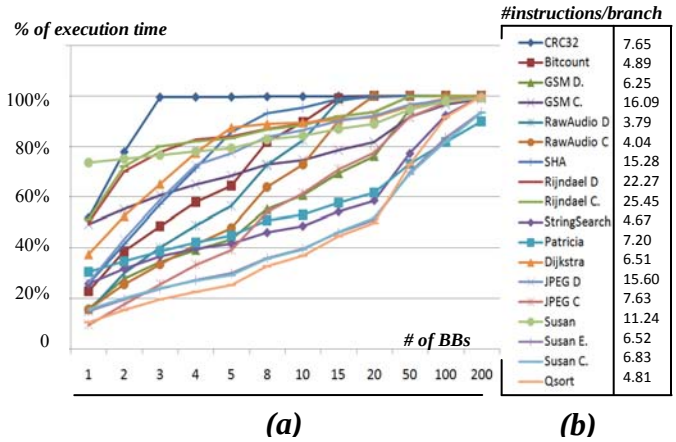


Figure 6.20: (a) How many BBs are necessary to cover a certain execution rate considering total execution time (b) Average size of the basic block

Table 6.13 shows three different configurations for the array used in the experiments. For each one it is varied the size of the reconfiguration cache: 16, 64 and 512 slots, and it is evaluated the impact of performing speculation, up to three basic blocks.

Table 6.13: Different configurations for the array, when coupling to the MIPS R3000

	C #1	C #2	C #3
#Lines	24	48	150
#Columns	11	16	20
#ALU / line	8	8	12
#Multipliers / line	1	2	2
#Ld/st / line	2	6	6

Table 6.14 demonstrates the speed up of the reconfigurable array for the same three configurations. It is ordered to show the most dataflow algorithms at the top and the most control flow ones at the bottom. In Configuration #3 with speculation, an average performance improvement of more than 2.5 times is achieved. Moreover, gains are shown regardless of the instruction/branch rate, even for very control oriented

algorithms such as *RawAudio Decoder* and *Quicksort*, as well as those which do not have distinct kernels, such as *Susan Corners*. Together with these results, there is an extra table at the right, demonstrating the overall optimization assuming infinite hardware resources for the array. As it can be observed, with the best configuration it is possible to get very close to this theoretical speedup in several algorithms: just in five of them there is a significant difference between the most aggressive configuration and the ideal. In fact, the algorithms that can most benefit from hardware infinite resources are exactly the dataflow ones, since they demand more lines in the array, mainly when speculation is used. They have as most executed kernels basic blocks with a huge number of instructions. On the other hand, in algorithms which have no distinct kernels, the most important resource to be increased is the number of slots available in the cache memory. Figure 6.21 summarizes the results Table 6.14.

Table 6.14: Speedups using the reconfigurable array coupled to the MIPS R3000 processor

Data Flow / Control Flow

Algorithm	Speed Up – Configuration #1						Speed Up – Configuration #2						Speed Up – Configuration #3						Ideal	
	No Speculation			Speculation			No Speculation			Speculation			No Speculation			Speculation			No Spec	Spec
	16	64	256	16	64	256	16	64	256	16	64	256	16	64	256	16	64	256		
Riijndael E.	1.05	1.20	1.21	1.05	1.24	1.24	1.05	1.71	1.73	1.06	1.55	1.55	1.05	3.46	3.60	1.06	2.68	2.68	5.10	8.05
Rijindael D.	1.07	1.21	1.21	1.07	1.25	1.25	1.07	1.63	1.64	1.07	1.55	1.55	1.07	3.32	3.33	1.07	2.32	2.32	4.68	7.42
GSM E.	1.63	1.65	1.68	2.01	2.05	2.13	1.63	1.65	1.68	2.03	2.07	2.17	1.63	1.65	1.69	2.03	2.07	2.19	1.70	2.19
JPEG E.	1.95	2.04	2.07	1.79	1.88	1.89	2.50	2.72	2.77	3.55	4.27	4.37	2.50	2.72	2.77	3.55	4.27	4.37	2.72	4.37
SHA	1.90	1.90	1.90	3.81	3.84	3.84	1.90	1.91	1.91	4.80	4.84	4.84	1.90	1.91	1.91	4.80	4.84	4.84	1.91	4.87
Susan Smothing	1.49	1.60	1.65	2.70	2.99	3.31	1.49	1.61	1.65	2.83	3.14	3.52	1.49	1.61	1.65	2.83	3.14	3.52	1.65	3.52
CRC	1.53	1.53	1.53	1.92	1.92	1.92	1.53	1.53	1.53	1.92	1.92	1.92	1.53	1.53	1.53	1.92	1.92	1.92	1.53	1.92
JPEG D.	1.92	2.03	2.04	1.64	1.78	1.78	2.05	2.21	2.22	2.02	2.54	2.55	2.05	2.21	2.22	2.03	2.62	2.63	2.77	4.39
Patricia	1.49	1.84	1.93	1.58	2.05	2.23	1.49	1.86	1.95	1.64	2.17	2.37	1.49	1.86	1.95	1.64	2.17	2.37	2.19	3.07
Susan Corners	1.22	1.49	1.72	1.31	1.47	1.91	1.38	1.79	2.17	1.56	1.79	2.64	1.38	1.79	2.17	1.56	1.79	2.64	2.17	2.66
Susan Edges	1.23	1.42	1.64	1.29	1.48	1.83	1.43	1.70	2.20	1.47	1.74	2.43	1.43	1.70	2.20	1.53	1.81	2.58	2.21	2.60
Dijkstra	1.59	1.71	1.71	2.03	2.21	2.22	1.59	1.72	1.72	2.04	2.24	2.24	1.59	1.72	1.72	2.04	2.24	2.24	1.72	2.25
GSM D.	1.28	1.28	1.29	1.27	1.28	1.29	1.62	1.62	1.65	1.48	1.50	1.52	2.79	2.79	2.93	2.37	2.49	2.58	3.31	3.68
Bitcount	1.76	1.76	1.76	1.83	1.83	1.83	1.76	1.76	1.76	1.83	1.83	1.83	1.76	1.76	1.76	1.83	1.83	1.83	1.76	1.83
Stringsearch	1.38	1.61	1.86	1.56	2.22	2.77	1.38	1.62	1.89	1.57	2.30	2.96	1.38	1.62	1.89	1.57	2.30	2.96	1.89	2.97
Quicksort	1.37	1.74	1.74	1.69	2.32	2.33	1.37	1.77	1.77	1.80	2.66	2.67	1.37	1.77	1.77	1.80	2.66	2.67	1.77	2.67
RawAudio E.	1.60	1.61	1.61	1.98	1.99	2.00	1.60	1.61	1.61	1.98	1.99	2.00	1.60	1.61	1.61	1.98	1.99	2.00	1.61	2.00
RawAudio D.	1.64	1.64	1.64	1.79	1.79	1.79	1.64	1.64	1.64	1.79	1.79	1.79	1.64	1.64	1.64	1.79	1.79	1.79	1.64	1.79
Average	*1.51*	*1.63*	*1.68*	*1.80*	*1.98*	*2.09*	*1.58*	*1.78*	*1.86*	*2.03*	*2.33*	*2.49*	*1.65*	*2.04*	*2.13*	*2.08*	*2.50*	*2.67*	*2.32*	*3.36*

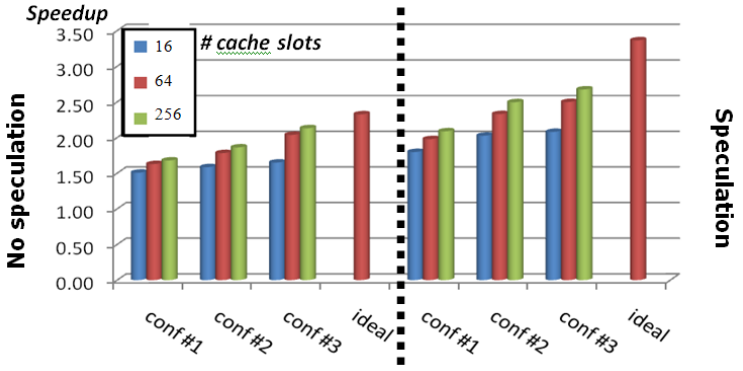


Figure 6.21: An overview of the average speed up presented with different configurations

At this moment, one can analyze the power and energy consumed by the system.

Figure 6.22 demonstrates the average power consumed by cycle in the Array coupled to the MIPS processor, with configurations #1 and #3 (shown as C#1 and C#3), considering 64 cache slots, and executing the algorithms *Rijndael E.*, *Rawaudio D.* and

JPEG E., the most control and data flow ones, and a mid-term, respectively. The same Figure also shows the MIPS processor without the reconfigurable array. The consumption is shown separated for the core, data and instruction memories, reconfigurable array and cache, and BT hardware. It is interesting to note that the major responsible for power consumption are the memory accesses. In third place comes the reconfigurable array. The power spent by this hardware depends on how much it is used during the program execution. The MIPS processor, reconfiguration cache and the BT hardware plays a minor role on this scenario.

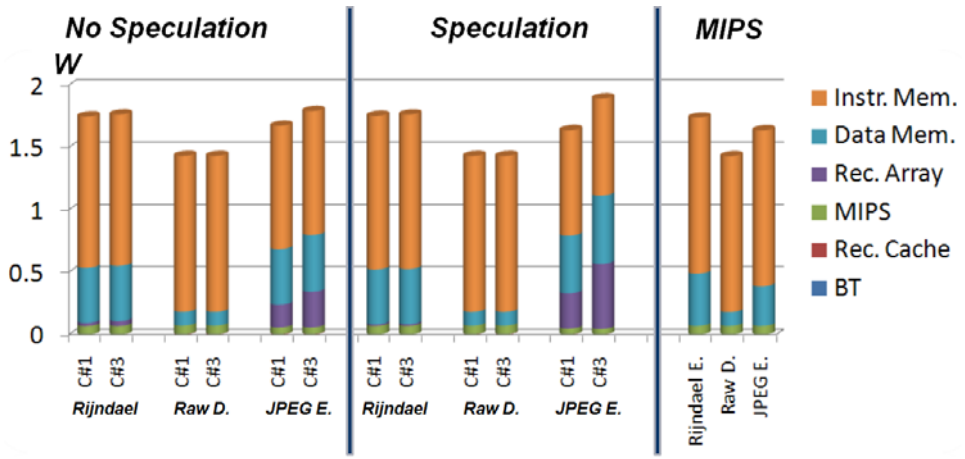


Figure 6.22: Power consumed by 3 different algorithms in conf. 1 and 3, with and without speculation, 64 cache slots

In Figure 6.23 the same experiment is repeated, but now analyzing the total energy consumption. As the power consumed per cycle is very similar when executing MIPS+array and just MIPS, but the number of cycles is reduced in the first case, energy savings are achieved. Making a deeper analysis, there are three main reasons for these savings:

- The execution of the instructions in a more effective way in combinational logic, instead of using the processor path.

Avoidance of repeated parallelism analysis. As commented before, there is no necessity of performing the analysis repeatedly for the same sequence of code, since DIM saves this information in its special cache. This is a very important characteristic, since, recalling again, almost half of the number of pipeline stages of the Pentium 4 processor (INTEL, 2008); and half of the power spent the Alpha 21264 processor are related to the extraction of dependence information among instructions (WILCOX; MANNE, 1999). As it can be observed in

- Figure 6.22, when using DIM, more power is spent in the core, because of the BT hardware, reconfigurable array and its cache. On the other hand, there is no need for fetching a great amount of instructions, since they reside in the reconfigurable cache, after their proper translation to an array's configuration.

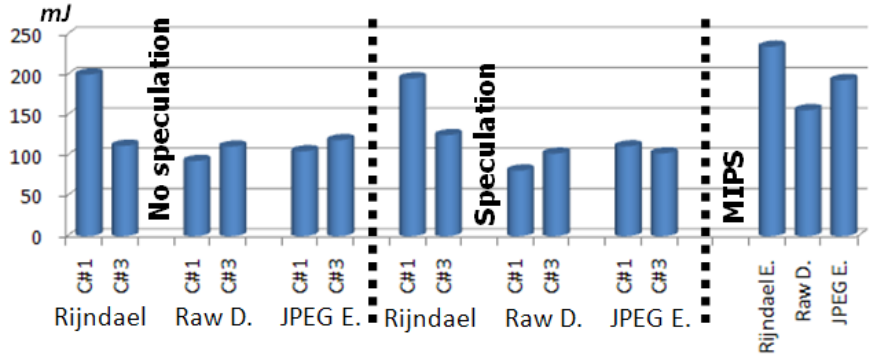


Figure 6.23: Repeating the data of the previous Figure, but now for Energy Consumption

For configuration #2, with 64 cache slots, the proposed system consumes 1.73 times less energy on average than the standalone MIPS core. Moreover, assuming that the MIPS itself would be enough to handle real time constraints necessary for a given application, one could reduce the system clock frequency to achieve exactly the same performance level of the processor - thus decreasing even more the power and energy consumptions.

In order to give an idea of the area overhead, Table 6.15a shows the number of functional units and multiplexers necessary to implement configuration #1 of Table 1, and what is the number of gates they consume. In the same table one can also find the area occupied by the DIM hardware. Table 6.15b shows the number of bits necessary to store one configuration in the reconfigurable cache. Note that, although 256 bits are necessary for the Write Bitmap Table, they are not added to the final total, since it is temporary and used just during detection. In Table 6.15c, the number of Bytes needed for different cache sizes is presented.

Table 6.15: Area evaluation

Unit	*#*	*Gates*
ALU	192	300,288
LD/ST	36	1,968
Multiplier	6	40,134
Input Mux	408	261,936
Output Mux	216	58,752
DIM Hardware	1,024	
Total	*664,102*	

(a)

Table	*#bits*
Write Bitmap	256
Resource	786
Reads Table	1,632
Writes Table	576
Context Start	40
Context	40
Immediate	128
Total	*3,202*

(b)

#Slots	*#Bytes*
2	833
4	1,601
8	3,300
16	6,404
32	13,012
64	25,616
128	51,304
256	102,464

(c)

Figure 6.24 represents the MIPS layout with the reconfigurable array. According to (YEAGER, 1996), the total number of transistors of core in the MIPS R10000 is 2.4 million. As presented in table 4a, the array together with the hardware detection occupies 664,102 gates. Considering that one gate is equivalent to 4 transistors, which would be the amount necessary to implement a NAND or NOR gates, the whole system would take nearly 2.66 million transistors to be implemented. This way, the reconfigurable array and DIM hardware would take nearly 2.66 million transistors to be

implemented. The area overhead is represented in Figure 6b. In this figure is also presented the area occupied by the reconfigurable cache, in number of different configurations it can support. This is an approximation, since it was not considered that the cache was fully associative (it is very likely that the area overhead will be slightly higher). The MimiMIPS, in turn, occupies 26,712 gates.

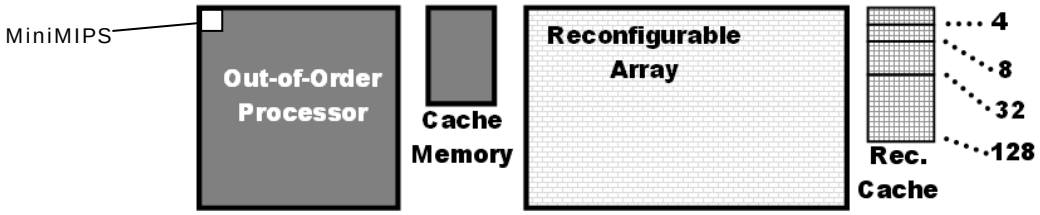


Figure 6.24: Area overhead presented by the reconfigurable array and its special cache

Finally, Table 6.16 and Table 6.17 show, respectively, the number of gates occupied by the functional units and the bits necessary to save each configuration in the cache, when varying the number of rows and columns of the reconfigurable array.

Table 6.16: Number of gates, varying the number of rows and columns of the array

#lines	*#columns*			*#functional units*					*#gates*					
	ALU	LD/ST	Mult	ALU	LD/ST	Mult	MUX in	MUX out	ALU	LD/ST	Mult	MUX in	MUX out	*Total*
12	8	2	1	96	8	4	192	96	150144	2624	26756	138672	29376	*347572*
	8	6	1	96	24	4	192	96	150144	7872	26756	159216	29376	*373364*
	12	6	2	144	24	8	288	96	225216	7872	53512	225984	29376	*541960*
24	8	2	1	192	16	8	384	192	300288	5248	53512	277344	58752	*695144*
	8	6	1	192	48	8	384	192	300288	15744	53512	318432	58752	*746728*
	12	6	2	288	48	16	576	192	450432	15744	107024	451968	58752	*1083920*
33	8	2	1	264	22	11	528	264	412896	7216	73579	381348	80784	*955823*
	8	6	1	264	66	11	528	264	412896	21648	73579	437844	80784	*1026751*
	12	6	2	396	66	22	792	264	619344	21648	147158	621456	80784	*1490390*
48	8	2	1	384	32	16	768	384	600576	10496	107024	554688	117504	*1390288*
	8	6	1	384	96	16	768	384	600576	31488	107024	636864	117504	*1493456*
	12	6	2	576	96	32	1152	384	900864	31488	214048	903936	117504	*2167840*
96	8	2	1	768	64	32	1536	768	1201152	20992	214048	1109376	235008	*2780576*
	8	6	1	768	192	32	1536	768	1201152	62976	214048	1273728	235008	*2986912*
	12	6	2	1152	192	64	2304	768	1801728	62976	428096	1807872	235008	*4335680*
150	8	2	1	1200	100	50	2400	1200	1876800	32800	334450	1733400	367200	*4344650*
	8	6	1	1200	300	50	2400	1200	1876800	98400	334450	1990200	367200	*4667050*
	12	6	2	1800	300	100	3600	1200	2815200	98400	668900	2824800	367200	*6774500*

Table 6.17: Number of bits necessary per cache slot, varying the number of rows and columns of the array

#lines	#columns			#functional units					#bits
	ALU	LD/ST	Mult	ALU	LD/ST	Mult	MUX in	MUX out	
12	8	2	1	96	8	4	192	96	*248*
	8	6	1	96	24	4	192	96	*268*
	12	6	2	144	24	8	288	96	*357*
24	8	2	1	192	16	8	384	192	*455*
	8	6	1	192	48	8	384	192	*495*
	12	6	2	288	48	16	576	192	*672*
33	8	2	1	264	22	11	528	264	*609*
	8	6	1	264	66	11	528	264	*664*
	12	6	2	396	66	22	792	264	*908*
48	8	2	1	384	32	16	768	384	*868*
	8	6	1	384	96	16	768	384	*948*
	12	6	2	576	96	32	1152	384	*1302*
96	8	2	1	768	64	32	1536	768	*1694*
	8	6	1	768	192	32	1536	768	*1854*
	12	6	2	1152	192	64	2304	768	*2562*
150	8	2	1	1200	100	50	2400	1200	*2623*
	8	6	1	1200	300	50	2400	1200	*2873*
	12	6	2	1800	300	100	3600	1200	*3979*

6.3 First studies about the ideal shape of the reconfigurable array

In (RUTZIG, 2008), studies about the ideal shape of the reconfigurable array considering a wide range of different applications were performed. Instead of using the rectangular shape, a tool was developed to execute several algorithms with different behaviors in order to find the best placement and usage of functional units in the array with the minimal performance loss possible. Significant results were achieved concerning the area occupied by the system. Figure 6.25a shows the original shape of the array. Figure 6.25b demonstrates it after the optimization analysis.

Figure 6.26 shows the average gain in performance considering all benchmarks and varying the cache memory size when comparing the reconfigurable array with different shapes to the architecture without the array. It is possible to observe in this figure that the new shape had a small performance loss (5.8% in average) when comparing to the data path 1 (original rectangular shape with a large amount of functional units). However, when comparing the new shape to the data path 2, which is also based on the traditional rectangular shape, but with a similar number of functional units as the new shape has, the new configuration has a performance improvement of 3%. Even though this relative gain appears to be low, it is important to point out that there is an area reduction of almost 15% over the data path 2. In other words, when considering the same number of functional units, the new shape presents a small performance improvement and a considerable area reduction when comparing to the traditional shape.

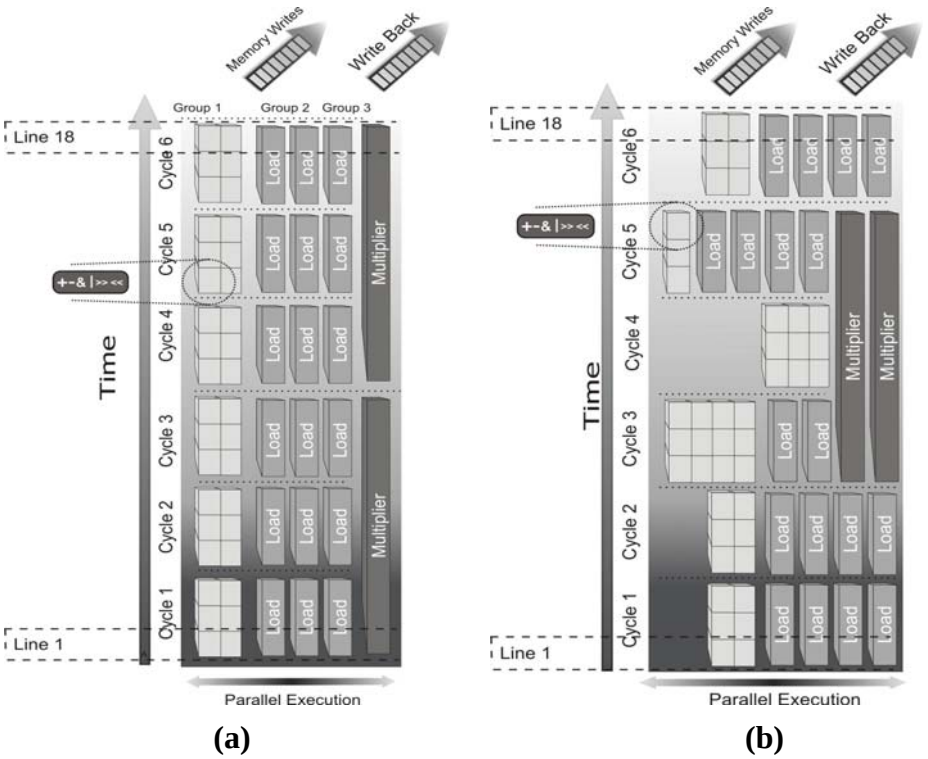


Figure 6.25: a) Original shape of the reconfigurable array b) Optimized shape

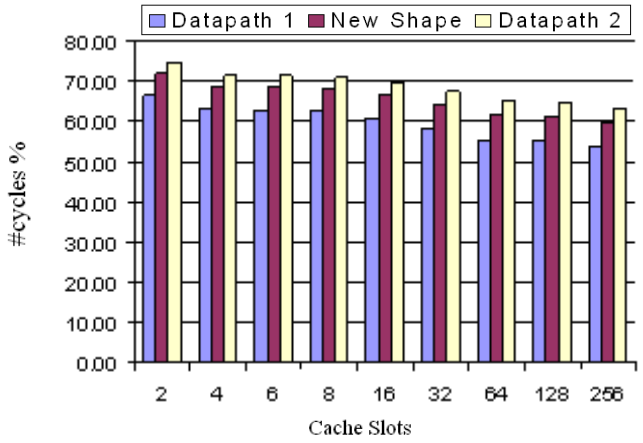


Figure 6.26: Performance comparison between different datapath shapes

6.4 Conclusions

Considering the different processors analyzed, although the significant area overhead and increase in power consumption, there was an important improvement on the performance and reduction in energy consumption – since parts of the code are executed in a more efficient mechanism.

Regarding the experiments with the Femtojava Low-Power processor, two main issues must be addressed in the future. The first one is the computation of the static power consumption (although it was considered in the other experiments, including the one with the Femtojava Multicycle). Moreover, the area was calculated in FPGA and, in this case, the cache memory was oversized: FPGAs, when using its logic, tend to

occupy a good number of resources to implement memories. Furthermore, considering the whole set of experiments with the Femtojava processor, the inclusion of a replacement policy for the reconfigurable cache must be done.

Considering all experiments, the performance numbers are mainly gathered through simulation. As these simulations are cycle accurate, the level of accuracy is high. All overheads, such as reconfiguration, context loading and write back were considered. The only exception was the simulations using sim-safe (section 6.2.1). In opposite to all other simulations performed, sim-safe works at the instruction level. This way, an average IPC was considered, based on several simulations previously performed in the cycle accurate simulator.

Regarding the area estimate, it depends on how the tool performs the place and routing. It is important to note that the area estimate, using the VHDL versions of the hardware, was done before place and routing. This way, depending on the tool methodology, more area will spent. In FPGA, for example, a great number of resources is spent with the routing mechanism. This way, the tradeoff presented by the array tends to be better on ASIC technologies. Moreover, as already previously stated, there is an error margin considering the cache area estimates: it was not considered that it would be fully associative. Furthermore, there is also the control mechanism for context input and output. Although it is a simple state machine, it will bring an additional area overhead.

Depending on the technology, more or less functional units can be allocated within a level in the array (processor equivalent cycle). Although this possibility is very unlikely to happen, because there is no wasted time with sequential logic as the ordinary pipeline stages of the processor presents, the worst case would be as follow: there is no room in the critical path to allocate more than one ALU per level in the array. In this case, one could think on decrease the frequency of operation so more ALU can be allocated per level. Although the array would not present performance gains, there would be huge energy savings.

Finally, the leakage power (one of the main responsible for the static power consumption) is still considered low in the employed technologies used in the experiments presented in this work. However, static power in future technologies will become more important. This way, studies have been done to take advantage of the structure of the array to decrease also static power.

7 CONCLUSIONS, FUTURE AND ON GOING WORKS

In this work we introduced a new approach that explores the potential of replacing the traditional execution flow for combinational logic. Employing similar techniques used by well-known dataflow architectures, but maintaining binary compatibility, good performance improvements and energy savings have been achieved. In Chapter 2 and 3 we contextualized our work, demonstrating from where the basic ideas come and how exactly our contribution fits in. We also demonstrated the potential of using reconfigurable systems and how the recent proposed ones just cover one niche of applications. In Chapter 4 and 5 the structure of the array and the BT algorithms were presented, considering the particularities of each architecture used as case study. Finally, in chapter 6, results for these architectures were shown, proving that is possible to optimize any kind of algorithm using pure combinational logic, and still allowing the reuse of binary code in a transparent process.

The list presented below summarizes the main contributions of this thesis:

- For Java processors (stack machines)
  - The development of the reconfigurable array structure;
  - The development of the BT algorithm;
  - Measurements in terms of area, performance and power/energy consumption, coupling the array to the Femtojava Low Power and Multicycle processors, and comparing the results against its VLIW version;
  - The VLIW analyzer, which was previously developed (BECK; CARRO, 2004c), was extended to support the array/BT simulation. Approximately 1800 lines in C code were written.
- For the MIPS and Simplescalar out-of-order Processors (RISC machines)
  - The development of the reconfigurable array structure;
  - The development of the BT algorithm;
  - A profiler to measure the potentials of the technique was developed to be used with the Simplescalar Toolset; more than 1700 lines in C code were written. Later, this profiler was adapted to be used together with the MIPS simulator (RUTZIG; CARRO, 2008);
    - Using this simulator, various tradeoffs analysis were done, such as the study of what would be the best speculation policy to be used.

  - Another profiler, working at a lower level, was also developed. It can be seen in Appendix B.
  - A MIPS core VHDL, called miniMIPS (MINIMIPS, 2008), was depurated and modified. The original version could not run even the simplest algorithms;
- First experiments simulating different reconfigurable arrays with different granularities have been done, in order to classify these reconfigurable systems according to their potentials and niche of applications. A simulator, coupled to the MIPS SystemC based description, was developed, with nearly 700 lines in C++

Besides, there are a large number of future and ongoing works, as it will be discussed in the next subsections.

7.1 Design space to be explored

There is still a huge design space to be explored, with a lot of open questions, such as: what is the ideal size of the array; how many instructions it can execute in parallel; how many results it can write back to the registers or memory per cycle; how many configurations the special cache can store, and what is the best replacement policy for it; how deep is the ideal speculation regarding basic blocks etc. Moreover, all these configurations can be different depending on the characteristics of the benchmark. That is why it is also planned to simulate other benchmarks, such as SPEC and Mediabench. Concerning specific Java Optimizations, first studies have already been done in order to optimize code at the object level (MATTOS, 2007).

7.2 Decreasing the routing area

The problem of the routing area in reconfigurable systems is a very well know issue. Although it is more evident in fine-grain reconfigurable systems based on FPGAs, it can also be observed in coarse grain ones. Depending on the configuration used in the array presented in this work, the area presented just by the multiplexers can reach half the total area of the reconfigurable system, as can be observed in Figure 6.24a. This way, different structures such as multistage networks have been tested in order to decrease the area (FERREIRA et al., 2008). It is important to stress that different routing techniques can affect the BT algorithm. By consequence, it is very likely that the BT algorithm will have to suffer changes.

7.3 Speculation of variable length

The speculation performed in the reconfigurable array has a fixed length, meaning that it will always speculatively execute a fixed number of basic blocks. However, with small modifications in the BT algorithm, it is possible to make the speculation with a variable number of basic blocks to be executed ahead. For instance, the BT could speculate until there are no more resources available in the array. Currently, it stops speculating if it reaches the pre determined depth in the tree, even if there is free room available in the array. It would also influence the reconfiguration cache, since each slot for each configuration should have more space. Another alternative would be the use of a cache slot with variable length, where some kind of flag could indicate the end of one configuration and the beginning of the other.

7.4 DSP, SIMD and other extensions

Following a trend that can be observed in nowadays reconfigurable systems, the study about adding DSP extensions in the array is a future work. This can be achieved by either adding new hardware or optimizing the routing mechanism (for instance, in the case of Multiply-and-Accumulate operations). The BT hardware should be adapted in order to detect these new instructions. This work should involve the search for the most common DSP instructions in a program's execution. This way, it would be possible to know the tradeoff of implementing such extensions.

In the case of SIMD instructions, the BT would detect in what set of instructions it could be applied, and configure the array according to it. Two different implementation alternatives must be evaluated: the first solution would be to increase word length of the functional units (so they could execute a SIMD instruction in one of these functional units, after it was detected and transformed), or optimize the routing between a given group of functional units to support multiple data with a single operation.

This approach could be extended to any kind of special instruction depending on the field of application desired. The greatest advantage of such extensions is that the array does not lose its generality: if it has any kind of extension and it is not used, the only drawback is the unutilized extra area of the array.

7.5 Study of the area overhead with technology scaling and future technologies

This study concerns the analysis of the area overhead according to future technologies. What is the impact of using the array with an even larger area available in a near future? Furthermore, what are the possibilities of implementing the array using other technologies instead of silicon? According to the roadmap (ITRS, 2006) these new technologies are slow but present huge integration possibilities, or the opposite. Considering the fact that the array is very regular and easily scalable, could this be an advantage?

7.6 Measuring the impact of the OS in reconfigurable systems

In conventional reconfigurable systems, the source code needs to be available, so it could be optimized to be executed on reconfigurable logic. However, some of the most used Operating Systems (OS) in the market do not have their source code available. This way, any part of the OS, such as system calls, could not be optimized at all. Some algorithms (even those that are part of benchmark sets) spend a considerable amount of time exactly with system calls (for instance, read/write from/to files). This way, a future work would be the analysis of this limiting factor on conventional reconfigurable systems, and the comparison of it against the proposed technique that, in turn, can optimize any part of the software, including OS parts that have already been compiled. This could answer the following question: what is the importance of optimizing the OS in the overall system speedup?

7.7 Array+BT to increase the Yield

One of the major problems that the industry faces nowadays is the yield rate. The number of processors that are produced is directly influenced by the faults that occurs during the manufacturing process. Considering general purpose processors, if one fault

occurs in its control or datapath, the whole processor cannot be used anymore. On the other hand, if the fault happens at the cache memory and the processor has, for instance, two separate banks, the processor maybe can still be used (the Intel Celeron processors are one of these examples) (INTEL, 2008).

Considering that the array can be coupled to a very simple processor, the percentage of area occupied by the reconfigurable logic would be enormous. If some fault occurs in a given part of the reconfigurable array (for instance, a given functional unit), that part would be isolated (it could be marked as always used in the resource table, for instance) and the array could be used without any kind of modification. A test could be done at the beginning of execution, using an algorithm to test all parts of the reconfigurable array, in order to mark the failed parts. As the array occupies the majority of the die area, it is very likely that any fail will occur at the reconfigurable part, increasing the overall yield rate.

7.8 Array+BT for fault tolerance

Several approaches replicate hardware in order to verify if the circuit is working properly. For instance, two processors can execute exactly the same software and compare their results to verify the validity of them. The reconfigurable array can also be used for this purpose. For example, instead of stalling the processor while executing sequences in the array, both can work together, comparing their results at the end of their execution. Another approach would be executing just certain parts of the software, which could be chosen randomly, so the execution overhead for the verification would be reduced. There are many opportunities in this field. For instance, only one array could be shared between various processors in a CMP to be used only for fault tolerance.

7.9 BT scheduling targeting to Low-Power

Considering the way the BT algorithm works now, it is target to achieve the highest level of parallelism possible. However, instead of trying to reach the maximum performance, the BT scheduler could try to place instructions in the array with the objective of keeping the largest possible number of functional units turned off – decreasing the power consumed by the system. For example, even if there is an opportunity of executing two instructions in parallel, but one of the functional units necessary for that operation was turned off in a previous configuration, another functional unit would be chosen, probably taking more time to execute the current configuration.

7.10 Comparison against Superscalar Architectures

Although in the results section a comparison with a superscalar architecture was demonstrated, the analysis lacks of a more theoretical background. For instance, one of the major problems about finding ILP in superscalar architectures is the number of comparators needed in the instruction queue. On the other hand, the complexity of this comparison when using the proposed approach diminishes, since the BT takes advantage of the hierarchy of the array: information about data dependency is summarized for each row. Moreover, the proposed reconfigurable system can be compared to complex superscalar architectures with different characteristics of the one

evaluated in this work, such as the number of pipeline stages, number of functional units, size of the instruction window etc.

7.11 Comparison against a Fine-Grain reconfigurable system

Is a coarse grain reconfigurable system faster than a FPGA based one? If one considers that the granularity of the first is coarser than the second, a simple operation would be executed faster in a coarse grain array. However, at bit manipulation, FPGAs tend to obtain an advantage. The routing, on the other hand, depends on the level of manipulation in the array. If words are computed, FPGA will spend more area and time to connect the bits to form this word.

Another particular issue when comparing specifically fine grain arrays with the proposed system: FPGA synthesis tools are more intelligent and have more time to build a configuration. This would be an advantage that could overcome some of the routing and allocation problems cited before. The BT system, on the other hand, needs to use a fixed structure and does not have any time to optimize it: the routing algorithm should be as simple as possible. Another interesting comparison would be using fixed configurations with a fine grain array, generated by a specific tool; against a coarse grain array exactly in the same conditions: fixed configurations also produced by a synthesis tool – so a fair environment would be available.

FIRST STUDIES WITH MOLEN

First studies about this comparison have already been performed, using Molen, a fine grain reconfigurable architecture that was introduced in chapter 2. To transform parts of code to be executed in Molen hardware, the C2VHDL, a tool developed by the same group, was employed. This tools aims at automatically transform a given C code in VHDL. Using the MiBench benchmark set, GSM has shown an improvement of 1.8x. Four more examples besides this one were executed: ADPCM, dijkstra, quick and bubble sort. However, some of them do not show performance improvements, while in others the speedup is very small, even if they are dataflow oriented algorithms.

Furthermore, these results consider that the additional hardware generated to be executed as the Molen reconfigurable system do not increase the critical path of the processor (it would take just one processor's equivalent cycle to execute the whole operation). It means that it would have the same processor's frequency of operation, which is very likely to not happen as previous implemented examples (VASSILIADIS et al., 2001) have already shown. This can be explained because of the code transformation tool, which are at its first stages of development: it does not consider critical paths while generating the VHDL cores. For instance, it is considered that a multiply operation would take the same time to be executed as an addition. Furthermore, it does not explore parallel operations at all. Therefore, for a better comparison and analysis, applications kernels need to be generated by hand.

This way, after the kernels were proper implemented, it will also be considered characteristics of the bench set (control or dataflow oriented; number of distinct kernels), which study has already begun, as one can see in chapter 2. This comparison must consider the three axis (power, performance and area). There is no study between two different reconfigurable architectures with different implementation strategies using exactly the same environment and peripheral components, and still considering the different behaviors of the benchmark set.

The toolset is already ready to be used. There is the SystemC like MIPS simulator adapted both to work with DIM and Molen as a GPP. For Molen, the simulator shows the hot spots considering basic blocks, or loops, or functions. Furthermore, the VHDL code of the MIPS processor and both reconfigurable architectures is available. The power consumption would follow the same methodology presented before for the BT/array, and the Xilinx XPower (XILINX, 2008b) could be used to figure the power spent by the FPGA based Molen system.

7.12 Attacking Different levels of granularity

MULTITHREADING

The search for processing power in a reduced design space has also been modifying the whole paradigm of parallelism exploitation. The focus has been changed, where complex and superpipelined superscalar processors are giving space to multiprocessors sometimes composed by simpler processors, increasing the scalability. The parallelism grain is not explored just at the instruction level anymore, but also at threads and processes. Figure 7.1 shows the difference between superscalar, multithreaded and simultaneous multithreading processors. Each square represents one functional unit of the processor. Each line (horizontal set of functional units) represents the processor state at each cycle. If one square is filled, it means that the correspondent functional unit was used. When it is empty, that functional unit was idle at that time.

Not using functional units through the time can be characterized as horizontal or vertical waste. Horizontal waste occurs when one or more functional units were not used within a cycle. Vertical waste means that all units within a cycle were not used (the whole cycle was wasted). Figure 7.1a demonstrates the execution sequence on a superscalar processor. In such processor, there is just the execution of one thread at a time. When, for some reason (as data dependences), there are no enough instructions to feed the functional units, there is horizontal waste. If there is no instruction at all to be issued to the functional units at a given cycle (this could happen when there is a cache miss, for example), vertical waste is characterized. The first kind of waste can be observed in cycles 1, 2, 5, 7 e 9, while the second kind in cycles 3, 4, 6, 8 and 10, in Figure 7.1a.

In Figure 7.1b and Figure 7.1c the behavior of the multithread and SMT processors are shown, respectively. The Multithread architecture can fetch instructions from different threads in different cycles, avoiding the vertical waste. For instance, while the processor is treating a cache miss of one thread, instructions can be fetched from another thread. The SMT architecture, on the other hand, can fetch instructions from different threads within the same cycle. This way, if there is a limit in the ILP of one thread, the functional units can be fed from others. By consequence, the horizontal waste is also dramatically reduced. In this architecture, the processor dynamically allocates the instructions from different threads in the functional units. For example, let us consider that in a processor there are eight functional units available, and in a given cycle the thread ILP is of five. The processor can fetch the 3 left instructions at run time from other threads to issue to the reminiscent functional units. Examples of SMT implementations are: Intel Pentium 4 (which technology is called Hyperthreading), Alpha EV8, IBM Power 5 and Sun Microsystems' UltraSPARC T1.

It is important to note that there is horizontal and vertical waste in reconfigurable architectures, as there is in superscalar processors. In the literature there is no study

about the possibility of using reconfigurable systems based on the SMT paradigm. Considering the proposed approach, the functional characteristics may facilitate the implementation of this technique. As it is composed by a large number of functional units, these can be fed of instructions from different threads, with a control mechanism for the input context and write back of results – and a special concern about maintaining data coherence and allowing communication between threads.

To became a reality, it is necessary the implementation of the control mechanism, which must be responsible for separating the input and output contexts, originated from different threads that are being executed simultaneously in the array. Other analysis can be performed, such as the study on the influence of the amount and kind of communication between threads on the performance and size of the array. It is important to highlight that the study of ideal size and other tradeoffs will be changed because of the fact that the reconfigurable array will execute more than one thread at a time.

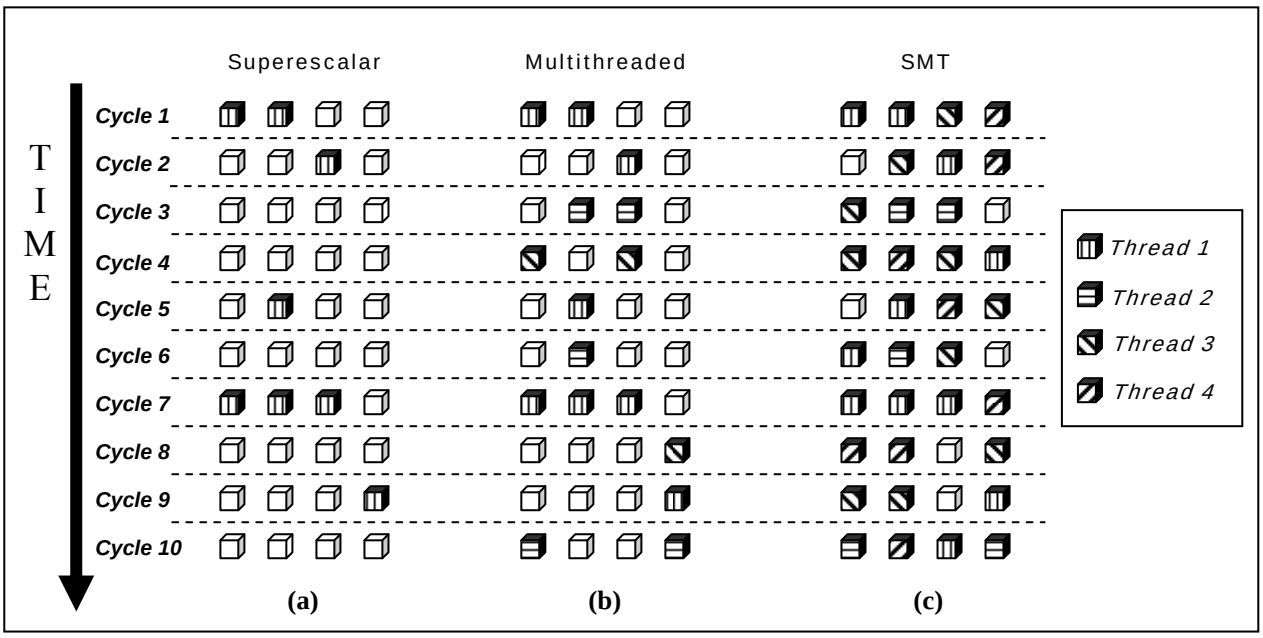


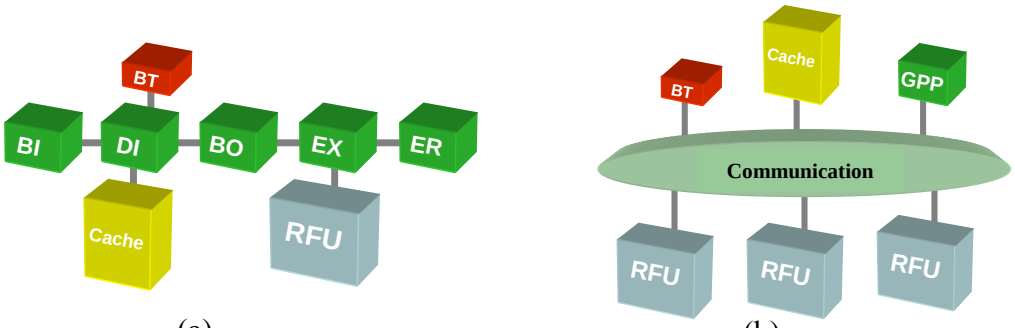
Figure 7.1: Different models and theirs functional units executing various threads

CMP

Processors such as Core2Duo (Intel) and AMD Athlon 64 X2 (AMD) are examples of another trend in the development of architectures: multiple processors in a chip, also known as CMP (Chip Multiprocessor). One of the main reasons that motivate designers to use CMP is the reduced design time necessary for its development, since the processors employed are usually already validated (because of the reuse of existent designs). This way, all the effort is focused on the communication between the components.

Reconfigurable systems have been used to optimize a single threaded software. This way, using these systems following the CMP strategy cited before can be a good focus of research. Considering the proposed approach, Figure 7.2a shows its current implementation: the communication between the components of the architecture and the

processor is done using dedicated buses, which makes its implementation not scalable considering the increment on the number of available RFUs. Figure 7.2b, in turn, illustrates how a CMP model could be implemented. With a new communication mechanism, it would be possible to increase the number of RFUs.

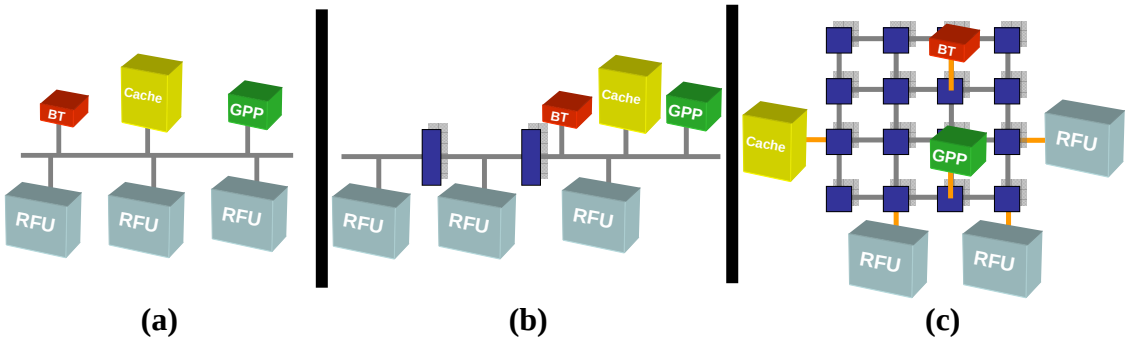


(a) (b)

Figure 7.2: a) Current implementation b) reconfigurable architecture based on CMP

There are a great number of open questions concerning reconfigurable CMP architectures, such as: how much parallelism is available, energy consumption, scalability, testability, fault tolerance, reusability, communication etc. Moreover, a new BT algorithm need to be developed, in order to analyze the partitioning of processes at run time, so they can be executed on different reconfigurable architectures, following the same premise of maintaining software compatibility.

Furthermore, it is necessary to analyze means of communication between the components, as well as memory sharing, such as monolithic buses (Figure 7.3a) or segmented (Figure 7.3b); use of a crossbar or even intrachip networks (Figure 7.3c). Finally, the possibility of implementing a heterogeneous architecture, composed by different reconfigurable units that can be used according to the process requirements at a given moment, can be evaluated. Similar studies using ordinary processors were done in (OLUKOTUN et al., 1996).



(a) **(b)** **(c)**

Figure 7.3: Communication alternatives. a) Monolithic bus b) Segmented bus c) Intra chip network

PUBLICATIONS

BECK FILHO, A. C. S.; MATTOS, J. C.; WAGNER, F. R.; CARRO, L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003, São Paulo. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 349-354.

BECK FILHO, A. C. S.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 12., 2003, Darmstadt. **Proceedings…** Darmstadt: Technische Universitat Darmstadt, 2003. p. 239-244.

BECK FILHO, A. C. S.; CARRO, L. A VLIW Low Power Java Processor for Embedded Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 17., 2004, Porto de Galinhas. **Proceedings…** New York: ACM Press, 2004. p. 157-162.

BECK FILHO, A. C. S.; CARRO, L. Um Processador Java VLIW com Baixo Consumo de Potência para Sistemas Embarcados. In: WORKSHOP IBERCHIP, 10., 2004, Cartagena de Indias. **Proceedings…** Cartagena de Indias: Universidad de los Andes, 2004. p. 114.

BECK FILHO, A. C. S.; CARRO, L. Dynamic Reconfiguration with Binary Translation: Breaking the ILP barrier with Software Compatibility. In: DESIGN AUTOMATION CONFERENCE, DAC, 42., 2005, Anaheim. **Proceedings…** New York: ACM Press, 2005, p. 732-737.

BECK FILHO, A. C. S.; CARRO, L.. Application of Binary Translation to Java Reconfigurable Architectures. In INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM - WORKSHOP 3 (RAW), IPDPS, 2005. **Proceedings…** Washington: IEEE Computer Society, 2005, p. 156.2.

BECK FILHO, A. C. S.; GOMES, V. F.; CARRO, L. Exploiting Java Through Binary Translation for Low Power Embedded Reconfigurable Systems. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 18., 2005, Florianópolis. **Proceedings…** Washington: IEEE Computer Society, 2005. p. 92-97.

BECK, A. C. S. ; CARRO, Luigi . Low Power Java Processor for Embedded Applications. In: Glesner,M.; Reis, R.; Indrusiak, L.; Eveking, H... (Org.). VLSI-SOC: From Systems to Chips. 1 ed. Boston: Springer, 2006, v. 1, p. 213-228.

BECK FILHO, A. C. S.; RUTZIG, M. B.; CARRO, L. Cache Performance Impacts for Stack Machines in Embedded Systems. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 19., 2006, Ouro Preto. **Proceedings…** New York: ACM Press, 2006. p. 155-160.

BECK FILHO, A. C. S.; GOMES, V. F.; CARRO, L. Automatic Dataflow Execution with Reconfiguration and Dynamic Instruction Merging. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 14., 2006, Nice. **Proceedings…** Nice: University of Grenoble, 2006, p. 30-35.

BECK, A. C. S.; GOMES, V. F.; CARRO, L. Dynamic Instruction Merging and a Reconfigurable Array: Dataflow Execution with Software Compatibility. In: INTERNATIONAL WORKSHOP ON APPLIED RECONFIGURABLE COMPUTING, ARC, 2006, Delft. **Proceedings…** Berlin/Heidelberg: Springer, 2006, p. 449-454.

BECK, A. C. S.; RUTZIG, M. B.; CARRO, L. Advantages of Java Processors in Cache Performance and Power for Embedded Applications. INTERNATIONAL WORKSHOP ON SYSTEMS, ARCHITECTURES, MODELING, AND SIMULATION, SAMOS, 2006, Samos. **Proceedings…**, Berlin/Heidelberg: Springer, v. 4017, 2006, p. 321-330.

BECK FILHO, A. C. S.; CARRO, L. Transparent Acceleration of Data Dependent Instructions for General Purpose Processors. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 15., 2007, Atlanta. **Proceedings…** Atlanta: Georgia Institute of Technology, 2007, p. 66-71.

BECK, A. C. S.; RUTZIG, M. B.; GAYDADJIEV, G.; CARRO, Luigi. Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2008, Munique. **Proceedings…** Washington: IEEE Computer Society, 2008. p. 1208-1213.

BECK, A. C. S.; RUTZIG, M. B.; GAYDADJIEV, G.; CARRO, Luigi. Run-time Adaptable Architectures for Heterogeneous Behavior Embedded Systems. In: INTERNATIONAL WORKSHOP ON APPLIED RECONFIGURABLE COMPUTING, ARC, 2008, Londres. **Proceedings…** Berlin/Heidelberg: Springer, 2008, p. 111-124.

FERREIRA, R., MARCONE, L, RUTZIG, M., BECK, A.C. S., CARRO, L. Reducing Interconnection Cost In Coarse-Grained Dynamic Computing Through Multistage Network. In: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 2008, Heidelberg. To be published.

GOMES, V. F.; BECK, A. C. S.; CARRO, L. A VHDL Implementation of a Low Power Pipelined Java Processor for Embedded Applications In: WORKSHOP IBERCHIP, 10., 2004, Cartagena de Indias. **Proceedings…** Cartagena de Indias: Universidad de los Andes, 2004. p. 102.

GOMES, V. F.; BECK, A. C. S.; CARRO L. Trading Time and Space on Low Power Embedded Architectures with Dynamic Instruction Merging. **Journal of Low Power Electronics**, Estados Unidos, v. 1, n. 3, 2005, p. 249-258.

GOMES, V. F.; BECK, A. C. S.; CARRO, L. Advantages of Java Machines in the Dynamic ILP Exploitation for Low-Power Embedded Systems. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 13., 2005, Perth. **Proceedings…** Perth: Edith Cowan University, 2005, p. 1 – 6.

MATTOS, Julio C. B.; BECK, A. C. S.; CARRO, L. Object-Oriented Reconfiguration. In: IEEE/IFIP INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, RSP, 18., 2007, Porto Alegre. **Proceedings...** Washington: IEEE Computer Society, 2007, p. 69-72.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. Transparent Dataflow Execution for Embedded Applications. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, ISVLSI, 2007, Porto Alegre. **Proceedings…** Washington: IEEE Computer Society, 2007, p. 47-54.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. Balancing Reconfigurable Data Path Resources According to Application Requirements. In INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM - WORKSHOP RAW, IPDPS, 2008. **Proceedings…** Washington: IEEE Computer Society, 2008, p. 1-8.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. Measuring the Efficiency of Cache Memory on Java Processors for Embedded Systems. **Journal of Integrated Circuits and Systems**, São Paulo, v. 2, n. 1, p. 7-13, Mar. 2007.

REFERENCES

ALTERA Quartus-II Homepage. Available at: <http://www.altera.com/products/software/quartus-ii/subscription-edition/qts-se-index.html>. Visited on: Aug. 4th, 2008.

ALTMAN, E. R.; KAELI, D.; SHEFFER, Y. Welcome to the Opportunities of Binary Translation. **Computer**, Los Alamitos, v.33, n.3, p. 40-45, Mar. 2000.

ALTMAN, K. E. et al. Advances and Future Challenges in Binary Translation and Optimization. **Proceedings of the IEEE,** Los Alamitos, CA, v.89, n.11, p. 1710-1722, Nov. 2001.

ANANIAN, C.S. **Reconfigurable Cryptography**: A Hardware Compiler for Cryptographic Applications. [S.1.]: Massachusetts Institute of Technology (MIT), CS Department, 1997. Technical Report.

ATHANAS, P. M.; SILVERMAN, H. F. Processor reconfiguration through instruction-set metamorphosis. **Computer**, Los Alamitos, CA, v.36, n.3, p. 11-18, Mar. 1993.

AUSTIN, T. et al. Mobile Supercomputers. **Computer**, Los Alamitos, v.37, n.5, p. 81-83, May 2004.

BALA, E. D. V.; DUESTERWALD, E.; BANERJIA, S. Dynamo: a transparent dynamic optimization system. In: CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2000, Vancouver. **Proceedings…** New York: ACM Press, 2000. p. 1-12.

BARAT, F.; LAUWEREINS, R. Reconfigurable Instruction Set Processors: A Survey. In: INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, RSP, 11., 2000, Paris. **Proceedings…** Washington: IEEE Computer Society, 2000. p. 168-173.

BECK FILHO, A. C. S.; MATTOS, J. C.; WAGNER, F. R.; CARRO, L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003, São Paulo. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 349-354.

BECK FILHO, A. C. S.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 12., 2003, Darmstadt. **Proceedings…** Darmstadt: Technische Universitat Darmstadt, 2003. p. 239-244.

BECK FILHO, A. C. S.; CARRO, L. A VLIW Low Power Java Processor for Embedded Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 17., 2004, Porto de Galinhas. **Proceedings…** New York: ACM Press, 2004. p. 157-162.

BECK FILHO, A. C. S.; CARRO, L. Um Processador Java VLIW com Baixo Consumo de Potência para Sistemas Embarcados. In: WORKSHOP IBERCHIP, 10., 2004, Cartagena de Indias. **Proceedings…** Cartagena de Indias: Universidad de los Andes, 2004. p. 114.

BECK FILHO, A. C. S. **Uso da Técnica VLIW para Aumento de Performance e Redução do Consumo de Potência em Sistemas Embarcados Baseados em Java**. 2004. 130 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BECK FILHO, A. C. S.; CARRO, L. Dynamic Reconfiguration with Binary Translation: Breaking the ILP barrier with Software Compatibility. In: DESIGN AUTOMATION CONFERENCE, DAC, 42., 2005, Anaheim. **Proceedings…** New York: ACM Press, 2005, p. 732-737.

BECK FILHO, A. C. S.; CARRO, L. Application of Binary Translation to Java Reconfigurable Architectures. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDPS, 19., 2005, RECONFIGURABLE ARCHITECTURES WORKSHOP, RAW, 3, 2005, Denver. **Proceedings…** Los Alamitos: IEEE Computer Society, 2005. p. 156.2.

BECK FILHO, A. C. S.; GOMES, V. F.; CARRO, L. Exploiting Java Through Binary Translation for Low Power Embedded Reconfigurable Systems. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 18., 2005, Florianópolis. **Proceedings…** Washington: IEEE Computer Society, 2005. p. 92-97.

BECK FILHO, A. C. S.; GOMES, V. F.; CARRO, L. Automatic Dataflow Execution with Reconfiguration and Dynamic Instruction Merging. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 14., 2006, Nice. **Proceedings…** Grenoble: Tima, 2006. p. 30-35.

BECK, A. C. S.; GOMES, V. F.; CARRO, L. Dynamic Instruction Merging and a Reconfigurable Array: Dataflow Execution with Software Compatibility. In: INTERNATIONAL WORKSHOP ON APPLIED RECONFIGURABLE COMPUTING, ARC, 2006, Delft. **Revised Selected Papers.** Berlin: Springer, 2006. p. 449-454.

BECK FILHO, A. C. S.; CARRO, L. Transparent Acceleration of Data Dependent Instructions for General Purpose Processors. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 15., 2007, Atlanta. **Proceedings…** New York: IEEE, 2007. p. 66-71.

BECK, A. C. S.; RUTZIG, M. B.; GAYDADJIEV, G.; CARRO, L. Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. In: DESIGN,

AUTOMATION AND TEST IN EUROPE, DATE, 2008, Munique. **Proceedings…** New York: ACM SIGDA, 2008. p. 1208-1213.

BECK, A. C. S.; RUTZIG, M. B.; GAYDADJIEV, G.; CARRO, L. Run-time Adaptable Architectures for Heterogeneous Behavior Embedded Systems. In: INTERNATIONAL WORKSHOP ON APPLIED RECONFIGURABLE COMPUTING, ARC, 2008, London. **Revised Selected Papers.** Berlin/Heidelberg: Springer, 2008. p. 111-124.

BURGER, D.; AUSTIN, T. M. The SimpleScalar Tool Set, Version 2.0. **ACM SIGARCH Computer Architecture News**, New York, v.26, n.3, p. 13-25, June 1997.

BURNS, J.; GAUDIOT, J. L. SMT Layout Overhead and Scalability. **IEEE Transactions on Parallel and Distributed Systems**, Piscataway, v.13, n.2, p.142-155, Feb. 2002.

CARDOSO, J. M. P.; WEINHARDT, M. XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS: RECONFIGURABLE COMPUTING IS GOING MAINSTREAM, 12., 2002, Montpellier. **Proceedings…** Londres: Springer-Verlag, 2002. p.864-874.

CLARK, N.; ZHONG, H.; MAHLKE, S. Processor Acceleration Through Automated Instruction Set Customization. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 36., 2003, San Diego. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 129-140.

COMPTON, K.; HAUCK, S. Reconfigurable Computing: A Survey of Systems and Software. **ACM Computing Surveys,** New York, v.34, n.2, p. 171-210, June 2002.

CONTE, G. The long and winding road to high-performance image processing with MMX/SSE. In: IEEE INTERNATIONAL WORKSHOP ON COMPUTER ARCHITECTURES FOR MACHINE PERCEPTION, CAMP, 5., 2000, Padova. **Proceedings...** Washington: IEEE Computer Society, 2000. p. 1368–1372.

COSTA, A. T.; FRANÇA, F. M. G.; CHAVES FILHO, E. M. The Dynamic Trace Memoization Reuse Technique. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, PACT, 9., 2000, Philadelphia. **Proceedings…** Washington: IEEE Computer Society, 2000. p. 92-99.

CRONQUIST, D. C. et al. Specifying and Compiling Applications for RaPiD. In: IEEE SYMPOSIUM ON FPGAS FOR CUSTOM COMPUTING MACHINES, FCCM, 1998, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1998. p. 116-125.

DEHNERT, J. C. et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In: INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION: FEEDBACK-DIRECTED AND RUNTIME OPTIMIZATION, 2003, San Francisco. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 15-24.

EBCIOGLU, E.; ALTMAN, E. R. DAISY: Dynamic compilation for 100% architectural compatibility. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE , ISCA, 24., 1997, Denver. **Proceedings…** New York: ACM Press, 1997. p. 26-37.

FERREIRA, R.; MARCONE, L.; RUTZIG, M.; BECK, A. C. S.; CARRO, L. Reducing Interconnection Cost In Coarse-Grained Dynamic Computing Through Multistage Network. In: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 2008, Heidelberg. To be published.

FLYNN, M. J.; HUNG, P. Microprocessor Design Issues: Thoughts on the Road Ahead. **IEEE Micro**, Los Alamitos, v.25, n.3, p. 16-31, May 2005.

GAJSKI, D. et al. SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. **IEEE Transactions on Very Large Scale Integration Systems**, New York, v.6, n.1, p. 84-100, 1998.

GOLDSTEIN, S. et al. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 26., Atlanta, 1999. **Proceedings…** Washington: IEEE Computer Society, 1999. p. 28-39.

GOMES, V. F.; BECK, A. C. S.; CARRO, L. A VHDL Implementation of a Low Power Pipelined Java Processor for Embedded Applications In: WORKSHOP IBERCHIP, 10., 2004, Cartagena de Indias. **Proceedings…** Cartagena de Indias: Universidad de los Andes, 2004. p. 102.

GOMES, V. F.; BECK, A. C. S.; CARRO L. Trading Time and Space on Low Power Embedded Architectures with Dynamic Instruction Merging. **Journal of Low Power Electronics**, USA, v. 1, n. 3, p. 249-258, 2005.

GOMES, V. F.; BECK, A. C. S.; CARRO, L. Advantages of Java Machines in the Dynamic ILP Exploitation for Low-Power Embedded Systems. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, VLSI-SOC, 13., 2005, Perth. **Proceedings…** Perth: IFIP, 2005. p. 1 – 6.

GONZÁLEZ, A.; TUBELLA, J.; MOLINA, C. Trace-Level Reuse. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, ICPP, 28., 1999, Wakamatsu. **Proceedings…** Washington: IEEE Computer Society, 1999. p. 30-37.

GSCHWIND, K. E. M.; ALTMAN, E.; SATHAYE, S. Binary Translation and Architecture Convergence Issues for IBM System/390. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, ICS, 2000, Santa Fe. **Proceedings…** Washington: IEEE Computer Society, 2000. p. 336–347.

GUPTA, R. K.; MICHELI G. D. Hardware-Software Co-Synthesis for Digital Systems. **IEEE Design and Test of Computers,** Los Alamitos, v.10, n.3, p. 29-41, July 1993.

GUTHAUS, M. R. et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: WORKSHOP ON WORKLOAD CHARACTERIZATION, WWC, 4., 2001, Texas. **Proceedings…** Washington: IEEE Computer Society, 2001. p. 3-14

HARTENSTEIN, R. W. Coarse Grain Reconfigurable Architecture. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 6., 2001, Yokohama. **Proceedings…** New York: ACM Press, 2001. p. 564-570.

HAUCK, S. The Chimaera Reconfigurable Functional Unit. In: IEEE SYMPOSIUM ON FPGA-BASED CUSTOM COMPUTING MACHINES, FCCM, 5., 1997, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1997. p. 87-96.

HAUSE, J. R.; WAWRZYNEK, J. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In: IEEE SYMPOSIUM ON FPGA-BASED CUSTOM COMPUTING MACHINES, FCCM, 5., 1997, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1997. p. 12-21.

HENKEL, J. A Low Power Hardware/Software Partitioning Approach for Core-Based Embedded Systems. In: DESIGN AUTOMATION CONFERENCE, DAC, 36., 1999, New Orleans. **Proceedings…** New York: ACM Press, 1999. p. 122-127.

HENKEL, J.; ERNST, R. A Hardware/Software Partitioner using a Dynamically Determined Granularity. In: DESIGN AUTOMATION CONFERENCE, DAC, 34., 1997, Anaheim. **Proceedings…** New York: ACM Press, 1997. p. 691-696.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: A Quantitative Approach. 3rd ed. Amsterdam: Morgan Kaufmann, 2003.

HENNING, J. L. SPEC CPU2000: Measuring CPU Performance in the New Millennium. **Computer**, Los Alamitos, v.33, n.7, p. 28-35, July 2000.

HEYSTERS, P.; SMIT, G.; MOLENKAMP, E. A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems. **Journal of Supercomputing**, Hingham, v.26, n.3, p. 283-308, Nov. 2003.

HOMPSON, S. E. et al. In search of "forever," continued transistor scaling one new material at a time. **IEEE Transactions on Semiconductor Manufacturing**, New York, v. 18, n.1, p. 26-36, Feb. 2005.

HON, A. **Reconfigurable Accelerators.** [S.l.]: Massachusetts Institute of Technology (MIT), Artificial Intelligence Laboratory, 1996. (Technical Report 1586).

HUANG, J.; LILJA, D. Exploiting Basic Block Value Locality with Block Reuse. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE, HPCA, 5., 1999, Orlando. **Proceedings…** Washington: IEEE Computer Society, 1999. p. 106-114.

INTEL Pentium 4 Homepage. Available at: <http://www.mentor.com/sysnthesis>. Visited on: June 5th, 2008.

ITO, S. A.; CARRO, L.; JACOBI, R. P. Making Java Work for Microcontroller Applications. **IEEE Design & Test of Computers**, Los Alamitos, v. 18, n. 5, p. 100-110, Sept. 2001.

SEMICONDUCTOR Industry Association Home Page. Available at: <http://www.itrs.net>. Visited on: July 31th, 2008.

JAIN, M. K.; BALAKRISHNAN, M.; KUMAR, A. ASIP Design Methodologies : Survey and Issues. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 4., 2001, Bangalore. **Proceedings…** Washington: IEEE Computer Society, 2001. p. 76-81.

KESSLER, R. E. The Alpha 21264 microprocessor. **IEEE Micro,** Los Alamitos, v. 19, n. 2, p. 24-36, Mar. 1999.

KIM, N. S. et al. Leakage Current: Moore's Law Meets Static Power, **Computer**, Los Alamitos, v. 36, p. 68-75, Dec. 2003.

KLAIBER A. **The technology behind Crusoe processors**. [S.l.]: Transmeta Corporation, 2003. Technical Report.

KOOPMAN JUNIOR, P. K. **Stack Computers, the new wave.** Chichester: Ellis Horwood, 1989.

KOUFATY, D.; MARR, D. T. Hyperthreading technology in the netburst microarchitecture. **IEEE Micro**, Los Alamitos, v. 23, n. 2, p. 56-65, Mar. 2003.

LAWTON, G. Moving Java into Mobile Phones. **Computer**, Los Alamitos, v.35, n.6, p. 17-20, 2002.

LEE, M. Design and Implementation of the MorphoSys Reconfigurable ComputingProcessor. **Journal of VLSI Signal Processing Systems**, Hingham, v. 24, n. 2-3, p. 147-164, Mar. 2000.

LEONARDO Spectrum Homepage. Available at: <http://www.mentor.com>. Visited on: July 25th, 2008.

LIPASTI, M. H.; WILKERSON, C. B.; SHEN, P. S. Value locality and load value prediction. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, ASPLOS, 7., 1996, Cambridge. **Proceedings…** New York: ACM Press, 1996. p. 138-147.

LODI, A. et al. A VLIW Processor With Reconfigurable Instruction Set for Embedded Applications. **IEEE Journal of Solid-State Circuits**, Los Alamitos, v. 38, n. 11, p. 1876–1886, Nov. 2003.

LYSECKY, R.; VAHID, F. A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2004, Paris. **Proceedings…** Washington: IEEE Computer Society, 2004. p. 10480-10485.

LYSECKY, R.; VAHID, F. A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2005, Munique. **Proceedings…** Washington: IEEE Computer Society, 2005. p. 18-23.

MATTOS, J. C. B.; BECK, A. C. S.; CARRO, Luigi. Object-Oriented Reconfiguration. In: IEEE/IFIP INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, RSP, 18., 2007, Porto Alegre. **Proceedings...** Los Alamitos: IEEE Computer Society, 2007. p. 69-72.

MEI, B. et al. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In: FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 2003, Lisbon. **Proceedings...** Berlin: Springer, 2003. p. 61-70.

MINIMIPS VHDL. Available at: <http://www.opencores.org>. Visited on: July 31st, 2008.

MIYAMORI, T.; OLUKOTUN, K. REMARC: Reconfigurable Multimedia Array Coprocessor. In: SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, 6., 1998, Monterey. **Proceedings…** New York: ACM Press, 1998. p. 261-265.

MULCHANDANI, D. Java for Embedded Systems. **IEEE Internet Computing**, Los Alamitos, v. 31, n. 10, p. 30-39, 1998.

OR-BACH, Z. (When) will FPGAs kill ASICs? In: DESIGN AUTOMATION CONFERENCE, DAC, 38., 2001, Las Vegas. **Proceedings…** New York: ACM Press, 2001. p. 321-322.

OLUKOTUN, K. et al. The Case for a Single-Chip Multiprocessor. In: INTERNATIONAL SYMPOSIUM ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, ASPLOS, 7., 1996, Cambridge. **Proceedings…** New York: ACM Press, 1996. p. 2-11.

PATEL, S. J.; LUMETTA, S. S. rePLay: A Hardware Framework for Dynamic Optimization. **IEEE Transactions Computers**, Washington, v. 50, n. 6, p. 590-608, June 2001.

PILLA, M.; NAVAUX, P.; COSTA, A.; FRANCA, F.; CHILDERS, B.; SOFFA, M. The limits of speculative trace reuse on deeply pipelined processors. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, SBAC-PAD, 15., 2003, São Paulo. **Proceedings…** Los Alamitos: IEEE Computer Society, 2003. p. 36-43.

PILLA, M. L.; CHILDERS, B. R.; COSTA, A. T. DA; FRANCA, F. M. G.; NAVAUX, P. O. A. A Speculative Trace Reuse Architecture with Reduced Hardware Requirements. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, SBAC-PAD, 18., 2006, Ouro Preto. **Proceedings…** Los Alamitos: IEEE Computer Society, 2006. p. 47-54.

PUTTASWAMY, K. et al. System Level Power-Performance Trade-Offs in Embedded Systems Using Voltage and Frequency Scaling of Off-Chip Buses and Memory. In: INTERNATIONAL SYMPOSIUM ON SYSTEMS SYNTHESIS, ISSS, 15., 2002, Kyoto. **Proceedings…** New York: ACM Press, 2002. p. 225-230.

RAZDAN, R.; SMITH, M. D. A High-Performance Microarchitecture With Hardware-Programmable Functional Units. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 27., 1994, San Jose. **Proceedings…** New York: ACM Press, 1994. p. 172-180.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. Transparent Dataflow Execution for Embedded Applications. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, ISVLSI, 2007, Porto Alegre. **Proceedings…** Los Alamitos: IEEE Computer Society, 2007. p. 47-54.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. Balancing Reconfigurable Data Path Resources According to Application Requirements. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDPS, 22.; RECONFIGURABLE ARCHITECTURES WORKSHOP, RAW, 6., 2008, Miami. **Proceedings…** Los Alamitos: IEEE Computer Society, 2005. p. 1-8.

SANKARALINGAM, K. et al. Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 30., 2003, San Diego. **Proceedings…** New York: ACM Press, 2003. p. 422-433.

SHANKLAND S. **Transmeta shoots for 700 MHz with new chip**. 2000. Available at: <http://cnet.com/news/0-1003-200-1 526 340.html?tag=st.ne.ni.rnbot.rn.ni>. Visited on: July 31th, 2008.

SIMA, D.; FALK, H. Decisive aspects in the evolution of microprocessors. **Proceedings of the IEEE**, Los Alamitos, v. 92, n. 12, p. 1896-1926, Dec. 2004.

SODANI, A.; SOHI, G. S. An Empirical Analysis of Instruction Repetition. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, ASPLOS, 8., 1998, San Jose. **Proceedings…** New York: ACM Press, 1998. p. 35-45.

SODANI, A., SOHI, G. S. Understanding the Differences Between Value Prediction and Instruction Reuse. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 31., 1998, Dallas. **Proceedings…** Washington: IEEE Computer Society, 1998. p. 205-215.

SPEC Jvm98 Benchmarks. Available at: < http://www.spec.org/jvm98/>. Visited on: July 31th, 2008.

STITT, G.; LYSECKY, R.; VAHID, F. Dynamic Hardware/Software Partitioning: A First Approach. In: DESIGN AUTOMATION CONFERENCE, DAC, 40., 2003, Anaheim. **Proceedings…** New York: ACM Press, 2003. p. 250-255.

STITT, G.; VAHID, F. Hardware/Software Partitioning of Software Binaries. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 2002, San Jose. **Proceedings…** New York: ACM Press, 2002. p. 164-170.

STITT, G.; VAHID, F. The Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic. **IEEE Design and Test of Computers**, Los Alamitos, v. 19, n. 3, p. 36-43, July 2002.

SWANSON, S. et al. WaveScalar. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 36., 2003, San Diego. **Proceedings…** Washington: IEEE Computer Society, 2003. p. 291-302.

SYNOPSYS Power Compiler Homepage. Available at: <http://www.synopsys.com/products/power/power_ds.html>. Visited on: July 31th, 2008.

TATAS, K.; SIOZIOS, K.; SOUDRIS D. A Survey of Existing Fine-Grain Reconfigurable Architectures and CAD tools. **Basic Definitions, Critical Design Issues and Existing Coarse-grain Reconfigurable Systems**. Berlin: Springer, 2007. p. 3-87.

THEODORIDIS, G.; SOUDRIS, D.; VASSILIADIS, S. A Survey of Coarse-Grain Reconfigurable Architectures and Cad Tools. Basic Definitions, Critical Design Issues and Existing Coarse-grain Reconfigurable Systems. In: VASSILIADIS, S.; SOUDRIS, D. (Ed.). **Fine and Coarse Grain Reconfigurable Computing.** Berlin: Springer, 2007. p. 89-149.

TIWARI, V.; MALIK, S.; WOLFE, A. Power Analysis of Embedded Software: a First Step Towards Software Power Minimization. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, 1994, San Jose. **Proceedings…** Washington: IEEE Computer Society, 1994. p. 384-390.

VAHID, F. et al. Highly configurable platforms for embedded computing systems. **Microelectronics journal**, Amsterdam, v. 34, n. 11, p. 1025-1029, 2003.

VASSILIADIS, N. et al. A RISC Architecture Extended by an Efficient Tightly Coupled Reconfigurable Unit. **International Journal of Electronics**, London, v. 93, n. 6, p. 421-438, 2006.

VASSILIADIS, S. et al. The Molen Polymorphic Processor. **IEEE Transactions on Computers**, Washington, v. 53, n. 11, p. 1363-1375, Nov. 2004.

VASSILIADIS, S. et al. The Molen Programming Paradigm. In: INTERNATIONAL WORKSHOP ON SYSTEMS, ARCHITECTURES, MODELING, AND SIMULATION, SAMOS, 3., 2004, Samos. **Proceedings…** Berlin: Springer, 2004. p. 1-10. (Lecture Notes in Computer Science, v. 3133).

VASSILIADIS, S.; WONG, S.; COTOFANA. S. The MOLEN ρμ-coded processor. In: FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 11., 2001, Belfast. **Proceedings...** Berlin: Springer, 2001. p. 275-285.

VENKATARAMANI G. et al. A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE AND SYNTHESIS FOR EMBEDDED SYSTEMS, CASES, 2001, Atlanta. **Proceedings…** New York: ACM Press, 2001. p. 116-125.

WALL D. W. Limits of instruction-level parallelism. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, ASPLOS, 4., 1991, Santa Clara. **Proceedings…** New York: ACM Press, 1991. p. 176-188.

WAZLOWSKI M., M. et al. PRISM-II compiler and architecture, In: IEEE SYMPOSIUM ON FPGA'S FOR CUSTOM COMPUTING MACHINES, FCCM, 3., 1993, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1993. p. 9-16.

WILCOX, K.; MANNE, S. Alpha processors: A history of power issues and a look to the future. In: COOLCHIPS TUTORIAL AN INDUSTRIAL PERSPECTIVE ON LOW POWER PROCESSOR DESIGN, 1999, Monterey. **Proceedings…** Washington: IEEE Computer Society, 1999.

WIRTHLIN, M. J.; HUTCHINGS, B. L. A Dynamic Instruction Set Computer. In: IEEE SYMPOSIUM ON FPGA'S FOR CUSTOM COMPUTING MACHINES, FCCM, 3., 1995, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1995. p. 99-107.

WIRTHLIN, M. J.; HUTCHINGS, B. L.; GILSON, K. L. The Nano Processor: A Low Resource Reconfigurable Processor. In IEEE WORKSHOP ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1994, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1994. p. 23-30.

WITTIG R. D.; CHOW P. OneChip: an FPGA processor with reconfigurable logic. In IEEE WORKSHOP ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1996, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1996. p. 126-135.

XILINX, The Programmable Logic Data Book. Available at: <http://*www*.xilinx.*com/*partinfo/databook.htm>. Last access: June 10th, 2008.

XILINX Xpower. Available at: <http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm>. Last access: 10 jun. 2008.

YEAGER, K. C. The MIPS R10000 Superscalar Microprocessor. **IEEE Micro**, Los Alamitos, v. 16, n. 2, p. 28-40, April 1996.

YU, P.; MITRA, T. Characterizing Embedded Applications for Instruction-Set Extensible Processors. In: DESIGN AUTOMATION CONFERENCE, DAC, 41., 2004, San Diego. **Proceedings…** New York: ACM Press, 2004. p. 723-728.

ZHANG, H. et al. A 1-V Heterogeneous Reconfigurable DSP IC for Wireless Baseband Digital Signal Processing. **IEEE Journal of Solid-State Circuits**, Los Alamitos, v. 35, n. 11, p. 1697-1704, Nov. 2000.

APPENDIX A CONFIGURATION FILE FOR SIMPLESCALAR

The configuration file employed for simulations using the Simplescalar toolset, regarding section 6.2.1, is shown below. This configuration was made so the simulator could behave as close as possible to the MIPS R10000 processor

```
################################################################################
##  GENERAL
################################################################################

# random number generator seed (0 for timer seed)
-seed                              1

# initialize and terminate immediately
# -q                           false

# restore EIO trace execution from <fname>
# -chkpt                      <null>

# redirect simulator output to file (non-interactive only)
# -redir:sim                  <null>

# redirect simulated program output to file
# -redir:prog                 <null>

# simulator scheduling priority
-nice                              0

# maximum number of inst's to execute
-max:inst                          0

# number of insts skipped before timing starts
-fastfwd                           0

# generate pipetrace, i.e., <fname|stdout|stderr> <range>
# -ptrace                     <null>

# profile stat(s) against text addr's (mult uses ok)
# -pcstat                     <null>

# operate in backward-compatible bugs mode (for testing only)
-bugcompat                     false


################################################################################
##  BRANCH PREDICTOR
################################################################################

# extra branch mis-prediction latency
-fetch:mplat                       3

# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred                         bimod
```

```
# bimodal predictor config (<table size>)
-bpred:bimod           512

# 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:2lev            1 1024 8 0

# combining predictor config (<meta_table_size>)
-bpred:comb            1024

# return address stack size (0 for no return stack)
-bpred:ras                      8

# BTB config (<num_sets> <associativity>)
-bpred:btb             512 1

##############################################################################
##  CARACTERISTICAS GERAIS - SUPERESCALAR
##############################################################################

# speed of front-end of machine relative to execution core
-fetch:speed                    1

# instruction fetch queue size (in insts)
-fetch:ifqsize                  4

# speculative predictors update in {ID|WB} (default non-spec)
# -bpred:spec_update         <null>

# instruction decode B/W (insts/cycle)
-decode:width                   4

# instruction issue B/W (insts/cycle)
-issue:width                    4

# run pipeline with in-order issue
-issue:inorder                false

# issue instructions down wrong execution paths
-issue:wrongpath               true

# instruction commit B/W (insts/cycle)
-commit:width                   4

# register update unit (RUU) size
-ruu:size                      16

# load/store queue (LSQ) size
-lsq:size                      16

##############################################################################
##  UNIDADES FUNCIONAIS
##############################################################################

# total number of integer ALU's available
-res:ialu                       2

# total number of integer multiplier/dividers available
-res:imult                      1

# total number of memory system ports available (to CPU)
-res:memport                    2

# total number of floating point ALU's available
-res:fpalu                      2

# total number of floating point multiplier/dividers available
-res:fpmult                     1
```

```
###################################################################################
##  CACHE
###################################################################################

############## Level 1

# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1             il1:16384:32:1:l

# l1 instruction cache hit latency (in cycles)
-cache:il1lat                     1

# l1 data cache config, i.e., {<config>|none}
-cache:dl1             dl1:16384:32:4:l

# l1 data cache hit latency (in cycles)
-cache:dl1lat                     1

############## Level 2

# l2 data cache config, i.e., {<config>|none}
-cache:dl2             ul2:16384:64:4:l

# l2 data cache hit latency (in cycles)
-cache:dl2lat                     1

# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                      dl2

# l2 instruction cache hit latency (in cycles)
-cache:il2lat                     1

##############

# flush caches on system calls
-cache:flush                  false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress              false

###################################################################################
##  MEMORIA
###################################################################################

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat                        1 1

# memory access bus width (in bytes)
-mem:width                        8

###################################################################################
##  TLB
###################################################################################

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:4096:4096:4:l

# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:4096:4096:4:l

# inst/data TLB miss latency (in cycles)
-tlb:lat
```

APPENDIX B FIRST VERSION OF THE LOW-LEVEL DIM ALGORITHM FOR SIMPLESCALAR

```
#include <stdio.h>
#include <string.h>
#include "machine.h"
#include "regs.h"
//#include "memory.h"
#include "simple_array_hardware.h"

//#define DEBUG_ON
//#define CACHE_DEBUG
//#define DEBUG_RECONFIGURABLE_ARRAY_ON


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// GENERAL DEFINITIONS
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

// Definicao das caracteristicas do array

#define REC_ARRAY_LINES 30

struct groups_specification {
   int start;
   int length;
} context_group[10];


// Resources
#define REC_ARRAY_RESOURCE_LD 0
#define REC_ARRAY_RESOURCE_ALU 1

void set_resources () {
   context_group[REC_ARRAY_RESOURCE_LD].start = 0;
   context_group[REC_ARRAY_RESOURCE_LD].length = 2;

   context_group[REC_ARRAY_RESOURCE_ALU].start =
context_group[REC_ARRAY_RESOURCE_LD].length;
   context_group[REC_ARRAY_RESOURCE_ALU].length = 3;

   return;
}

#define REC_ARRAY_RESOURCE_TOTAL 5
#define REC_ARRAY_RESOURCE_TOTAL_READ (REC_ARRAY_RESOURCE_TOTAL * 2)
//////////////////////////////

#define REC_ARRAY_CONTEXT 24
#define REC_ARRAY_IMMED 24
#define REC_ARRAY_CONTEXT_TOTAL 48  // REC_ARRAY_CONTEXT + IMEDIATO


////////////////////////////////////////////////////////////////////////
```

```
// Tables

// Bitmap writes
int bitmap_writes[REC_ARRAY_LINES];

// Resources
// int resource_table[REC_ARRAY_LINES][REC_ARRAY_RESOURCE_TOTAL];
int resource_table_bitmap[REC_ARRAY_LINES][REC_ARRAY_RESOURCE_TOTAL];
int resource_table_function[REC_ARRAY_LINES][REC_ARRAY_RESOURCE_TOTAL];

// reads write
int read_table[REC_ARRAY_LINES][REC_ARRAY_RESOURCE_TOTAL_READ];
int write_table[REC_ARRAY_LINES][REC_ARRAY_CONTEXT];

// context
int context_table[REC_ARRAY_CONTEXT_TOTAL];
int context_table_start[REC_ARRAY_CONTEXT_TOTAL];
int context_table_flag[REC_ARRAY_CONTEXT_TOTAL];  // 0 - free, 1 - busy, 2 - write

#define CONTEXT_FLAG_FREE  0
#define CONTEXT_FLAG_BUSY  1
#define CONTEXT_FLAG_WRITE 2

int context_table_regpointer_r1_write;
int context_table_regpointer_r2_write;
int context_table_regpointer_w_write;
int context_table_next_free = 0;
int context_table_next_immed_free = REC_ARRAY_CONTEXT; // comeca no final da tabela
"normal"

//////////////////////////////////////////////////////////////////////
// Start tables

void DTM_start_tables () {
   int i,j;

   for (j=0;j<REC_ARRAY_CONTEXT_TOTAL;j++) {
      context_table[j] = -1;
      context_table_start[j] = -1;
   }

   for (j=0;j<REC_ARRAY_CONTEXT;j++) {
      context_table_flag[j] = CONTEXT_FLAG_FREE;
   }


   for (i=0;i<REC_ARRAY_LINES;i++) {

      bitmap_writes[i] = 0;
      for (j=0;j<REC_ARRAY_RESOURCE_TOTAL;j++) {
         resource_table_bitmap[i][j] = 0;
         resource_table_function[i][j] = 0;
      }

      for (j=0;j<REC_ARRAY_RESOURCE_TOTAL_READ;j++)
         read_table[i][j] = 0;

      for (j=0;j<REC_ARRAY_CONTEXT;j++)
         write_table[i][j] = 0;

   }

   context_table_next_free = 0;
   context_table_next_immed_free = REC_ARRAY_CONTEXT;

   set_resources ();

   return;
}


//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
```

```
// CACHE
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

#define N_REC_CACHE 16

struct array_cache {

   int read_table[REC_ARRAY_LINES][REC_ARRAY_RESOURCE_TOTAL_READ];
   int write_table[REC_ARRAY_LINES][REC_ARRAY_CONTEXT];
   int resource_table_function[REC_ARRAY_LINES][REC_ARRAY_RESOURCE_TOTAL];

   int context_table[REC_ARRAY_CONTEXT_TOTAL];
   int context_table_start[REC_ARRAY_CONTEXT_TOTAL];
   int context_table_flag[REC_ARRAY_CONTEXT_TOTAL];  // 0 - free, 1 - busy, 2 - write

   md_addr_t start_pc, end_pc;

   int fifo_position;

} rec_cache[N_REC_CACHE];

void start_cache_DTM () {
   int i;

   for (i=0;i<N_REC_CACHE;i++) {
      rec_cache[i].fifo_position = -1;
      rec_cache[i].start_pc = -1;
   }


   return;
}

void debug_DTM(void);

void add_configuration (int slot, int pc, int pc_final) {

   int i,j;

   // Grava a configuracao atual neste slot
   for (j=0;j<REC_ARRAY_CONTEXT_TOTAL;j++) {
      rec_cache[slot].context_table[j] = context_table[j];
      rec_cache[slot].context_table_start[j] = context_table_start[j];
   }

   for (j=0;j<REC_ARRAY_CONTEXT;j++) {
      rec_cache[slot].context_table_flag[j] = context_table_flag[j];
   }

   #ifdef DEBUG_ON
   printf("aaaaaaaa SLOT %i\n", slot);
   #endif

   for (i=0;i<REC_ARRAY_LINES;i++) {

      for (j=0;j<REC_ARRAY_RESOURCE_TOTAL;j++)  {
         rec_cache[slot].resource_table_function[i][j] = resource_table_function[i][j];
      }

      for (j=0;j<REC_ARRAY_RESOURCE_TOTAL_READ;j++)
         rec_cache[slot].read_table[i][j] = read_table[i][j];

      for (j=0;j<REC_ARRAY_CONTEXT;j++)
         rec_cache[slot].write_table[i][j] = write_table[i][j];
   }
   ////////////////////////

   rec_cache[slot].start_pc = pc;
   rec_cache[slot].end_pc = pc_final;

   #ifdef DEBUG_ON
   debug_DTM();
```

```
   #endif

   return;
}

int cache_DTM (md_addr_t pc, md_addr_t pc_final, int rw) {
   int i,where_hit = -1,j;
   int k;
   static int fifo_pointer = 0;


   for (i=0;i<N_REC_CACHE;i++) {
      if (rec_cache[i].start_pc == pc) {
         where_hit = i;
      }
   }

#ifdef CACHE_DEBUG
   printf("-*-*-*-*-*-*-*-*-* CACHE CONFIGURATION -*-*-*-*-*-*-*-*-*\n");
   for (i=0;i<N_REC_CACHE;i++) {
      printf("%8x - %i\n", rec_cache[i].start_pc, rec_cache[i].fifo_position);
/*
      if (rec_cache[i].start_pc == 0x402980) {

            printf("SLOT %i\n\n", i);


   printf("\n*************************************************************************
***\n");
            printf("Tabela de contexto inicial de reads\n\n");

            for (j=0;j<REC_ARRAY_CONTEXT_TOTAL;j++)
               printf("%3i ",rec_cache[i].context_table_start[j]);

            printf("\n\nTabela de contexto atual\n\n");

            for (j=0;j<REC_ARRAY_CONTEXT_TOTAL;j++) {
               printf("%3i ",rec_cache[i].context_table[j]);
            }

            printf("\n");

            for (j=0;j<REC_ARRAY_CONTEXT;j++) {
               printf("%3i ",rec_cache[i].context_table_flag[j]);
            }

            printf("\n");

   printf("*************************************************************************
*\n");
            printf("Bitmap *    Tabela de    *        Tabela de Reads         *
Tabela de  \n");
            printf("Writes *     Recursos    *                                *
Writes   \n");

   printf("*************************************************************************
*\n");

            for (k=0;k<REC_ARRAY_LINES;k++) {

               printf("0000 ");
               printf("  * ");


               for (j=0;j<REC_ARRAY_RESOURCE_TOTAL;j++)
                  //if (resource_table_bitmap[k][j])
                     printf("%2x ", rec_cache[i].resource_table_function[k][j]);

               printf(" * ");

               for (j=0;j<REC_ARRAY_RESOURCE_TOTAL_READ;j++)
                  printf("%2i ", rec_cache[i].read_table[k][j]);

               printf(" * ");
```

```
               for (j=0;j<REC_ARRAY_CONTEXT;j++)
                  printf("%2i ", rec_cache[i].write_table[k][j]);

               printf("\n");

            }

            printf("\n\n\n");
      }
*/
   }
   printf("-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-\n");
#endif

   if (rw) {
      // Nao achou, inclui
      if (where_hit == -1) {
            add_configuration(fifo_pointer,pc,pc_final);
            fifo_pointer++;
            if (fifo_pointer == N_REC_CACHE) fifo_pointer = 0;
/*

         for(i=0;i<N_REC_CACHE;i++) {
            if ( (rec_cache[i].fifo_position == (N_REC_CACHE - 1)) ||
(rec_cache[i].start_pc == -1) ) {
               add_configuration(i,pc,pc_final);
               where_hit = i;
               break;
            }
         }
*/
      }
   }
/*
   if (where_hit != -1) {
      rec_cache[where_hit].fifo_position = 0;

      for(j=0;j<N_REC_CACHE;j++) {
         if ( (j != where_hit) && (rec_cache[j].fifo_position != -1 ) ) {
            rec_cache[j].fifo_position++;
         }
      }
    }
*/
   return where_hit;

}

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// ARRAY HARDWARE
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

// array components

int mux (int entrada[], int controle) {
   return entrada[controle];
}

int alu (int entrada_A, int entrada_B, int controle) {
   int function, type;

   function = (controle >> 4) & 0xf;
   type = controle & 0xf;

   switch (function) {
      case 0:
         // INSTR  ADD
         // INSTR  ADDU
         // INSTR  ADDI
```

```
   // ADDIU  ADDIU

   //if (type == 0)
      return entrada_A + entrada_B;
   //else
   //addu...

break;

case 1:

   // INSTR  SUB
   // INSTR  SUBU

   //if (type == 0)
      return entrada_A - entrada_B;
   //else
   //subu...

break;

case 2:

   // INSTR  AND_
   // INSTR  ANDI

   return entrada_A & entrada_B;
break;

case 3:

   // INSTR  OR_
   // INSTR  ORI

   return entrada_A | entrada_B;
break;

case 4:

   // INSTR  XOR
   // INSTR  XORI

   return entrada_A ^ entrada_B;
break;

case 5:

   // INSTR  NOR
   // Conferir

   return ~(entrada_A | entrada_B);
break;

case 6:



   // INSTR  SLL
   if (type == 0) {
      return entrada_A << ( entrada_B & 0xff);
   }
   // INSTR  SLLV
   else if (type == 1) {
      return entrada_A << entrada_B;
   }
   // INSTR  SRL
   else if (type == 2) {
      int partial_res = entrada_A, i;
      for (i= (entrada_B & 0xff) ;i>0;i--)
         partial_res = (partial_res >> 1) & 0x7fffffff; // right sem signal
      return partial_res;

   }
   // INSTR  SRLV
   else if (type == 3) {
```

```
            int partial_res = entrada_A, i;
            for (i= entrada_B;i>0;i--)
               partial_res = (partial_res >> 1) & 0x7fffffff;
            return partial_res;
         }
         // INSTR  SRA
         else if (type == 4) {
            int partial_res = entrada_A, i;
            for (i= (entrada_B & 0xff) ;i>0;i--)
               partial_res = (partial_res >> 1) + (entrada_A & 0x80000000);

            return partial_res;
         }
         // INSTR  SRAV
         else if (type == 5) {
            int partial_res = entrada_A, i;
            for (i= entrada_B ;i>0;i--)
               partial_res = (partial_res >> 1) + (entrada_A & 0x80000000);

            return partial_res;
            }
      break;


      // set less than
      case 7:

         // INSTR  SLT
         if (type == 0) {
            if (entrada_A < entrada_B) return 1;
         }
         // INSTR  SLTU
         else if (type == 1) {
            if ( (unsigned) entrada_A < entrada_B) return 1;
         }
         // INSTR  SLTI
         else if (type == 2) {
            if (entrada_A < entrada_B) return 1;
         }
         // INSTR  SLTIU
         else if (type == 3) {
            if ( (unsigned) entrada_A < entrada_B) return 1;
         }



         else return 0;

      break;

   }

   return 0;
}


int ld (int entrada_A, int entrada_B, int controle) {
   int function, type, temp;

   function = (controle >> 4) & 0xf;
   type = controle & 0xf;

   switch (function) {
      case 0:
         if (type == 0) {        // byte     LB
            temp = (read_memory(entrada_A + entrada_B)) & 0xff;
            if (temp & 0x80) temp |= 0xffffff00;
            return temp;
         }
         else if (type == 3) {      // byte  unsigned LBU
            return (read_memory(entrada_A + entrada_B)) & 0xff;
         }

         else if (type == 1)  {     // half     LH
            temp = (read_memory(entrada_A + entrada_B)) & 0xffff;
```

```
            if (temp & 0x8000) temp |= 0xffff0000;
            return temp;
         }

         else if (type == 5)       // half  unsigned LHU
            return (read_memory(entrada_A + entrada_B)) & 0xffff;

         else if (type == 2)       // word     LW
            return read_memory(entrada_A + entrada_B);
         //else
            //{ printf("naaaaah\n"); exit(0); }
            //MEM_WRITE_WORD(mem, addr, *((word_t *)p));
      break;

      case 1:
         return entrada_B << 16;
      break;

   }

   return 0;
}

// context loading

void load_context(int contexto_atual[], int GPR_regs[], int slot) {
   int j,i;

   if (slot != -1) {  // Trabalha apenas com o contexto atual
      for (j=0;j<REC_ARRAY_CONTEXT_TOTAL;j++) {
         context_table[j] = rec_cache[slot].context_table[j];
         context_table_start[j] = rec_cache[slot].context_table_start[j];
      }

      for (j=0;j<REC_ARRAY_CONTEXT;j++) {
         context_table_flag[j] = rec_cache[slot].context_table_flag[j];
      }

      for (i=0;i<REC_ARRAY_LINES;i++) {

         for (j=0;j<REC_ARRAY_RESOURCE_TOTAL;j++)
            resource_table_function[i][j] =
rec_cache[slot].resource_table_function[i][j];

         for (j=0;j<REC_ARRAY_RESOURCE_TOTAL_READ;j++)
            read_table[i][j] = rec_cache[slot].read_table[i][j];

         for (j=0;j<REC_ARRAY_CONTEXT;j++)
            write_table[i][j] = rec_cache[slot].write_table[i][j];
      }
   }

   ///////////////////////

   for (j=0;j<REC_ARRAY_CONTEXT;j++) {
      if (context_table_start[j] != -1) {
         contexto_atual[j] = GPR_regs[context_table_start[j]];
      }
      else
         contexto_atual[j] = -1;
   }

   for (j=REC_ARRAY_CONTEXT;j<REC_ARRAY_CONTEXT_TOTAL;j++) {
      contexto_atual[j] = context_table_start[j];
   }

   return;
}

int resultado_contexto_atual[REC_ARRAY_CONTEXT_TOTAL];

int reconfigurable_array (int GPR_regs[], int slot){
   int linha, coluna, contexto;
   int saida_unidade_funcional[REC_ARRAY_RESOURCE_TOTAL];
   int (*unidade_funcional_corrente) ();
```

```
   load_context(resultado_contexto_atual, GPR_regs, slot);

   for (linha = 0; linha < REC_ARRAY_LINES; linha++) {

      #ifdef DEBUG_RECONFIGURABLE_ARRAY_ON

         printf("\n\n\n");
         printf("Contexto atual\n");
         printf("------------------------------------------------\n");

         for (contexto = 0; contexto < REC_ARRAY_CONTEXT; contexto ++) {
            if (context_table[contexto] != -1)
               printf("r%i =\t %8x \t\t%i\n", context_table[contexto],
resultado_contexto_atual[contexto], context_table_flag[contexto]);
         }

         printf("------------------------------------------------\n");
         printf("\n");
         for (contexto = 0; contexto < REC_ARRAY_CONTEXT_TOTAL; contexto ++) {
               printf("%4x ", resultado_contexto_atual[contexto]);
         }

         printf("\n");
         printf("------------------------------------------------\n");

      #endif


      for (coluna = 0; coluna < REC_ARRAY_RESOURCE_TOTAL; coluna++) {

         if (coluna >= context_group[REC_ARRAY_RESOURCE_LD].start &&
            coluna < context_group[REC_ARRAY_RESOURCE_LD].length) {

            unidade_funcional_corrente = ld;
         }

         else if (coluna >= context_group[REC_ARRAY_RESOURCE_LD].length &&
            coluna < (context_group[REC_ARRAY_RESOURCE_ALU].start +
context_group[REC_ARRAY_RESOURCE_ALU].length) )

            unidade_funcional_corrente = alu;
         /////

         saida_unidade_funcional[coluna] =
         unidade_funcional_corrente (  mux(resultado_contexto_atual, read_table[linha][
coluna * 2]),
               mux(resultado_contexto_atual, read_table[linha][(coluna  * 2) + 1]),
               resource_table_function[linha][coluna]
            );

      }

      #ifdef DEBUG_RECONFIGURABLE_ARRAY_ON
         printf("Entradas\n");
         for (coluna = 0; coluna < REC_ARRAY_RESOURCE_TOTAL; coluna++) {

            printf("%8x %8x --",
               mux(resultado_contexto_atual, read_table[linha][coluna * 2]),
               mux(resultado_contexto_atual, read_table[linha][(coluna * 2) + 1])
            );
         }
         printf("\n");
         printf("------------------------------------------------\n");

         printf("Saidas\n");
         for (coluna = 0; coluna < REC_ARRAY_RESOURCE_TOTAL; coluna++)
            printf("    %8x      --", saida_unidade_funcional[coluna] );


         printf("\n");
         printf("------------------------------------------------\n");
      #endif
```

```
      // Todos menos o campo dos imediatos
      for (contexto = 0; contexto < REC_ARRAY_CONTEXT; contexto ++) {
         if (write_table[linha][contexto] > 0)
            resultado_contexto_atual[contexto] =
mux(saida_unidade_funcional,write_table[linha][contexto] - 1);
      }
   }

   // write back dos resultados

   for (contexto = 0; contexto < REC_ARRAY_CONTEXT; contexto ++) {
      if ( (context_table[contexto] != -1) && (context_table_flag[contexto] ==
CONTEXT_FLAG_WRITE) )
         regs.regs_R[context_table[contexto]] = resultado_contexto_atual[contexto];
   }

   // retorna o novo pc
   return rec_cache[slot].end_pc;
}



////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// DTM
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////
// Warning

void BUM (int error) {

   printf("BUUUUUUUUUUUUUUUUM\n\n");

   switch (error) {
      case 0: printf("Number of lines\n"); break;
   }

   exit(0);
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// 5th stage

void DTM_fill_tables_2st_step (
      int op_r1, int op_r2, int op_w, int resource_line, int resource_columm,
      int context_table_pointer_r1,int context_table_pointer_r2,int
context_table_pointer_w,
      int context_table_regpointer_r1_write,int context_table_regpointer_r2_write,int
context_table_regpointer_w_write,
      int immed, int immed_use) {


   int pointer_r1, pointer_r2, pointer_w;

   if (context_table_regpointer_r1_write == 1) {
      pointer_r1 = context_table_next_free++;
      context_table[pointer_r1] = op_r1;
   }
   else
      pointer_r1 = context_table_pointer_r1;

   if (context_table_regpointer_r2_write == 1) {
      if (immed_use) {
         pointer_r2 = context_table_next_immed_free++;
```

```
         context_table[pointer_r2] = immed;
      }
      else{
         pointer_r2 = context_table_next_free++;
         context_table[pointer_r2] = op_r2;
      }
   }
   else

   if (context_table_regpointer_w_write == 1) {
      pointer_w = context_table_next_free++;
      context_table[pointer_w] = op_w;
      context_table_flag[pointer_w] = CONTEXT_FLAG_WRITE;
   }
   else
      pointer_w = context_table_pointer_w;


   if (context_table_regpointer_r1_write == 1)
      context_table_start[pointer_r1] = op_r1;

   if (context_table_regpointer_r2_write == 1) {
      if (immed_use)
         context_table_start[pointer_r2] = immed;
      else
         context_table_start[pointer_r2] = op_r2;
   }


   read_table[resource_line][ resource_columm * 2     ] = pointer_r1;
   read_table[resource_line][(resource_columm * 2) + 1] = pointer_r2;

   write_table[resource_line][pointer_w] = resource_columm + 1;

}

//////////////////////////////////////////////////////////////////////
// 4th stage
void DTM_fill_tables_1st_step (int op_r1, int op_r2, int op_w, int resource_line, int
resource_columm, int immed, int immed_use, int function, int type) {

   int i;
   int context_table_pointer_temp_r1 = 0;
   int context_table_pointer_temp_r2 = 0;
   int context_table_pointer_temp_w  = 0;

   int op_w_bits;
   op_w_bits = 1 << op_w;
   bitmap_writes[resource_line] |= op_w_bits;

   resource_table_bitmap[resource_line][resource_columm] = 1;

bits
   resource_table_function[resource_line][resource_columm] = (function << 4) | (type &
0xf);

   context_table_regpointer_r1_write = context_table_regpointer_r2_write =
context_table_regpointer_w_write = 1;

   for (i=0;i<REC_ARRAY_CONTEXT;i++) {
      if (context_table[i] == op_r1) {
         context_table_pointer_temp_r1 = i;
         context_table_regpointer_r1_write = 0;
      }

      if (context_table[i] == op_w) {
         context_table_flag[i] = CONTEXT_FLAG_FREE;
      }

   }

   if (!immed_use) {
      for (i=0;i<REC_ARRAY_CONTEXT;i++) {
         if (context_table[i] == op_r2) {
            context_table_pointer_temp_r2 = i;
```

```
            context_table_regpointer_r2_write = 0;
         }
      }
   }


   #ifdef DEBUG_ON
      printf("*********************************************************\n");
      printf("** FOURTH STAGE\n");
      printf("*********************************************************\n");
      printf("Context pointers\n");
      printf("op_r1 = %i  w =
%i\n",context_table_pointer_temp_r1,context_table_regpointer_r1_write);
      printf("op_r2 = %i  w =
%i\n",context_table_pointer_temp_r2,context_table_regpointer_r2_write);
      printf("op_w  = %i  w = %i\n",context_table_pointer_temp_w
,context_table_regpointer_w_write);
   #endif

   DTM_fill_tables_2st_step (
      op_r1, op_r2, op_w, resource_line, resource_columm,

   context_table_pointer_temp_r1,context_table_pointer_temp_r2,context_table_pointer_tem
p_w,

   context_table_regpointer_r1_write,context_table_regpointer_r2_write,context_table_reg
pointer_w_write,
      immed, immed_use
   );

}

////////////////////////////////////////////////////////////////////////
// 3rd stage

void DTM_check_resource_table (int group, int op_r1, int op_r2, int op_w, int
dependence_line, int immed, int immed_use, int function, int type) {
   int resource_line, resource_columm = 0;
   int found = 0;


   for (resource_line = dependence_line;resource_line < REC_ARRAY_LINES;resource_line++)
{
      for (resource_columm = context_group[group].start;
         resource_columm < (context_group[group].start + context_group[group].length);
         resource_columm++) {
         if (resource_table_bitmap[resource_line][resource_columm] == 0) {
            found = 1;
            break;
         }
      }
      if (found) break;
   }

   if (!found) BUM(0);

   DTM_fill_tables_1st_step (op_r1, op_r2, op_w, resource_line, resource_columm, immed,
immed_use, function, type);

}

////////////////////////////////////////////////////////////////////////
// 2nd stage

void DTM_check_dependences (int group, int op_r1, int op_r2, int op_w, int immed, int
immed_use, int function, int type) {
   int dependence_line = 0;
   int i;

   int op_r1_bits = 0, op_r2_bits = 0, op_bits;
   if (op_r1 != -1) op_r1_bits = 1 << op_r1;
   if (op_r2 != -1) op_r2_bits = 1 << op_r2;

   op_bits = op_r1_bits | op_r2_bits;
```

```
   for (i=0;i<REC_ARRAY_LINES;i++)
      if ((bitmap_writes[i] & op_bits) != 0) dependence_line = i + 1;

   #ifdef DEBUG_ON
   printf("*******************************************************\n");
   printf("** SECOND STAGE\n");
   printf("*******************************************************\n");

   printf("dependence line = %i\n", dependence_line);
   printf("group = %2i  op_r1 = %2i  op_r2 = %2i  op_w = %2i  immed =
%i\n",group,op_r1,op_r2,op_w,immed);
   #endif

   if (dependence_line >= REC_ARRAY_LINES) BUM(0);

   DTM_check_resource_table(group, op_r1, op_r2, op_w, dependence_line, immed,
immed_use, function, type);

}

////////////////////////////////////////////////////////////////////////
// Primeiro Estagio
int DTM_instrution_decoder (md_inst_t inst, int verifica_existe) {

   // Separa operadores de read e write
   int op_r1, op_r2, op_w, immed, immed_use;

   int group;
   int function = 0, type = 0;

   int rs,rt,rd;

   rs = (inst.b >> 24) & 0xff;
   rt = (inst.b >> 16) & 0xff;
   rd = (inst.b >> 8) & 0xff;
   immed = inst.b & 0xffff;

   immed_use = 0;


   switch(inst.a) {

      ////////////////////////////////////////////////////////////////////////////
      // Loads/Stores
      ////////////////////////////////////////////////////////////////////////////

      // A extensao de sinal do IMMED pode ser feita na propria unidade funcional!
      // Pensar em hardware. Da pra dar um merge em tudo e apenas mudar o type. IF
dentro de IF
      // Load - type = 0

      case LB:  // 0x20:
         // extensao de sinal
         if (immed & 0x8000) immed |= 0xffff0000;
         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;

         group = REC_ARRAY_RESOURCE_LD;
         function = 0; type = 0;
         break;
      case LBU: // 0x22:
         // extensao de sinal
         if (immed & 0x8000) immed |= 0xffff0000;

         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;

         group = REC_ARRAY_RESOURCE_LD;
         function = 0; type = 3;
         break;
```

```
      case LH:  // 0x24:
         // extensao de sinal
         if (immed & 0x8000) immed |= 0xffff0000;

         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;

         group = REC_ARRAY_RESOURCE_LD;
         function = 0; type = 1;
         break;

      case LHU: // 0x26:
         // extensao de sinal
         if (immed & 0x8000) immed |= 0xffff0000;

         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;

         group = REC_ARRAY_RESOURCE_LD;
         function = 0; type = 5;
         break;

      case LW:  // 0x28:
         // extensao de sinal
         if (immed & 0x8000) immed |= 0xffff0000;

         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;

         group = REC_ARRAY_RESOURCE_LD;
         function = 0; type = 2;
         break;

      case LUI:
         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;

         group = REC_ARRAY_RESOURCE_LD;
         function = 1; type = 0;
         break;

      // Store - function = 2
/*
      case SW:  // 0x28:
         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;
         group = REC_ARRAY_RESOURCE_LD;
         function = 0;
         type = 1;
         break;
*/
      ////////////////////////////////////////////////////////////////////////////
      // arithmetic and logic
      ////////////////////////////////////////////////////////////////////////////

      // function

      // 0 - add
      // 1 - sub
      // 2 - and
      // 3 - or
      // 4 - xor
      // 5 - nor
      //
      //    type até aqui
```

```
//       0 - signed
//       1 - unsigned
//
// 6 - shift
// 7 - set


case  ADD:  // 0x40:
   op_w  = rd;
   op_r1 = rs;
   op_r2 = rt;
   group = REC_ARRAY_RESOURCE_ALU;

   function = 0;
   type = 0;

   break;

case  ADDU: // 0x42:
   op_w  = rd;
   op_r1 = rs;
   op_r2 = rt;
   group = REC_ARRAY_RESOURCE_ALU;

   function = 0;
   type = 1;

   break;

// Imediato

case ADDI:  // 0x41:
   op_w  = rt;
   op_r1 = rs;
   op_r2 = -1;
   immed_use = 1;
   group = REC_ARRAY_RESOURCE_ALU;
   function = 0;
   type = 0;

   break;

case ADDIU: // 0x43:
   op_w  = rt;
   op_r1 = rs;
   op_r2 = -1;
   immed_use = 1;
   group = REC_ARRAY_RESOURCE_ALU;

   function = 0;
   type = 1;

   // extensao de sinal
   if (immed & 0x8000) immed |= 0xffff0000;

   break;

//////////////////////////////////////////////////////////////////////

case  SUB:  // 0x44:
   op_w  = rd;
   op_r1 = rs;
   op_r2 = rt;
   group = REC_ARRAY_RESOURCE_ALU;

   function = 1;

   break;

case  SUBU: // 0x45:
   op_w  = rd;
   op_r1 = rs;
   op_r2 = rt;
   group = REC_ARRAY_RESOURCE_ALU;
```

```
         function = 1;
         type = 1;

         break;

      ////////////////////////////////////////////////////////////////////

      case  AND_:  // 0x4e:
         op_w  = rd;
         op_r1 = rs;
         op_r2 = rt;
         group = REC_ARRAY_RESOURCE_ALU;

         function = 2;

         break;

      // Imediato

      case ANDI:  // 0x4f:
         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;
         group = REC_ARRAY_RESOURCE_ALU;

         function = 2;

         break;

      ////////////////////////////////////////////////////////////////////

      case  OR:   // 0x50:
         op_w  = rd;
         op_r1 = rs;
         op_r2 = rt;
         group = REC_ARRAY_RESOURCE_ALU;

         function = 3;

         break;

      // Imediato

      case ORI:   // 0x51:
         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;
         group = REC_ARRAY_RESOURCE_ALU;

         function = 3;

         break;

      ////////////////////////////////////////////////////////////////////

      case  XOR:  // 0x52:
         op_w  = rd;
         op_r1 = rs;
         op_r2 = rt;
         group = REC_ARRAY_RESOURCE_ALU;

         function = 4;

         break;

      // Imediato

      case XORI:  // 0x53:
         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;
         group = REC_ARRAY_RESOURCE_ALU;
```

```
   function = 4;

   break;
////////////////////////////////////////////////////////////////////////

case  NOR:  // 0x54:
   op_w  = rd;
   op_r1 = rs;
   op_r2 = rt;
   group = REC_ARRAY_RESOURCE_ALU;

   function = 5;

   break;

////////////////////////////////////////////////////////////////////////
//shifts

case  SLLV: // 0x56:
   op_w  = rd;
   op_r1 = rt;
   op_r2 = rs;
   group = REC_ARRAY_RESOURCE_ALU;

   function = 6;
   type = 1;

   break;

case  SRLV: // 0x58:
   op_w  = rd;
   op_r1 = rt;
   op_r2 = rs;
   group = REC_ARRAY_RESOURCE_ALU;

   function = 6;
   type = 3;

   break;

case  SRAV: // 0x5a:
   op_w  = rd;
   op_r1 = rt;
   op_r2 = rs;
   group = REC_ARRAY_RESOURCE_ALU;

   function = 6;
   type = 5;

   break;

// SHAMT (= immed & 0ff)

case SLL: // 0x55:
   op_w  = rd;
   op_r1 = rt;
   op_r2 = -1;
   group = REC_ARRAY_RESOURCE_ALU;

   immed_use = 1;
   function = 6;
   type = 0;

   break;

case SRL: // 0x57:
   op_w  = rd;
   op_r1 = rt;
   op_r2 = -1;
   group = REC_ARRAY_RESOURCE_ALU;

   immed_use = 1;
   function = 6;
   type = 2;
```

```
         break;

      case SRA: // 0x59:
         op_w  = rd;
         op_r1 = rt;
         op_r2 = -1;
         group = REC_ARRAY_RESOURCE_ALU;

         immed_use = 1;
         function = 6;
         type = 4;

         break;

      ////////////////////////////////////////////////////////////////////
      // set less than
      case  SLT:  // 0x5b:
         op_w  = rd;
         op_r1 = rs;
         op_r2 = rt;
         group = REC_ARRAY_RESOURCE_ALU;

         function = 7; type = 0;

         break;

      case  SLTU: // 0x5d:
         op_w  = rd;
         op_r1 = rs;
         op_r2 = rt;
         group = REC_ARRAY_RESOURCE_ALU;

         function = 7; type = 1;

         break;

      // Imediato

      case SLTI:  // 0x5c:
         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;
         group = REC_ARRAY_RESOURCE_ALU;
         function = 7; type = 2;

         // extensao de sinal
         if (immed & 0x8000) immed |= 0xffff0000;

         break;

      case SLTIU: // 0x5e:
         op_w  = rt;
         op_r1 = rs;
         op_r2 = -1;
         immed_use = 1;
         group = REC_ARRAY_RESOURCE_ALU;

         function = 7; type = 3;

         break;

      default:
         #ifdef DEBUG_ON
         printf("*************** \n\nNao achei %x\n", inst.a);
         #endif
         if (verifica_existe) return 0;

   }

   if (verifica_existe) return 1;

   // Chama o segundo estagio
   DTM_check_dependences(group, op_r1, op_r2, op_w, immed, immed_use, function, type);
```

```
}


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////
// DTM

int DTM (md_inst_t inst, md_addr_t pc) {

   DTM_instrution_decoder(inst, 0);

   return 0;

}


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// DEBUG
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

int debug_confere_resultado (int GPR_regs[]) {
   int j;

   for (j=0;j<REC_ARRAY_CONTEXT;j++) {
      if (context_table_flag[j] == CONTEXT_FLAG_WRITE) {
         if (resultado_contexto_atual[j] != GPR_regs[context_table[j]] )  {
            printf("\n\n");
            printf("r%i \t array = %x   registrador = %x  \n",context_table[j],
resultado_contexto_atual[j], GPR_regs[context_table[j]]);
            return context_table[j];
         }
      }
   }

   return -1;

}

void debug_DTM () {
   int i,j;

   printf("\n*************************************************************************
***\n");
   printf("Tabela de contexto inicial de reads\n\n");

   for (j=0;j<REC_ARRAY_CONTEXT_TOTAL;j++)
      printf("%3i ",context_table_start[j]);

   printf("\n\nTabela de contexto atual\n\n");

   for (j=0;j<REC_ARRAY_CONTEXT_TOTAL;j++) {
      printf("%3i ",context_table[j]);
   }

   printf("\n");

   for (j=0;j<REC_ARRAY_CONTEXT;j++) {
      printf("%3i ",context_table_flag[j]);
   }

   printf("\n");
   printf("*************************************************************************
*\n");
```

```
   printf("Bitmap *    Tabela de    *        Tabela de Reads         *    Tabela de
\n");
   printf("Writes *     Recursos    *                                *      Writes
\n");
   printf("*********************************************************************************
*\n");

   for (i=0;i<REC_ARRAY_LINES;i++) {

      printf("%4x ", bitmap_writes[i]);
      printf("  * ");

/*    for (j=0;j<REC_ARRAY_RESOURCE_TOTAL;j++)
         printf("%2i ", resource_table_bitmap[i][j]);
*/

      for (j=0;j<REC_ARRAY_RESOURCE_TOTAL;j++)
         //if (resource_table_bitmap[i][j])
            printf("%2x ", resource_table_function[i][j]);

      printf(" * ");

      for (j=0;j<REC_ARRAY_RESOURCE_TOTAL_READ;j++)
         printf("%2i ", read_table[i][j]);

      printf(" * ");

      for (j=0;j<REC_ARRAY_CONTEXT;j++)
         printf("%2i ", write_table[i][j]);

      printf("\n");

   }

   printf("\n\n\n");

   return;
}

#ifdef DEBUG_ON

int main_DTM() {

   md_inst_t inst[50];
   int i = 0, j = 0;

   // rs, rt, rd

   //ADD
   //0x  op_r1 op_r2 op_w     xx
   //LB
   //0x  op_r1 op_w     xx    xx

   // add r7,r5,r6
   inst[j  ].a = ADD;
   inst[j++].b = 0x050607ff;
   // add r7,r7,r6
   inst[j  ].a = ADD;
   inst[j++].b = 0x070608ff;
   // add r2,r8,r6
   inst[j  ].a = ADD;
   inst[j++].b = 0x080609ff;
   // add r1,r2,r7
   inst[j  ].a = ADD;
   inst[j++].b = 0x020701ff;
   // lb r7,r2
   inst[j  ].a = ADD;
   inst[j++].b = 0x020704ff;

   inst[j  ].a = -1;

   DTM_start_tables ();

   do {
```

```
      printf("\n\n\n");
      md_print_insn(inst[i], 0, stdout); printf("\n");
      printf("%s %s\n", MD_OP_NAME(inst[i].a), MD_OP_FORMAT(inst[i].a) );
      printf("----------------------------------\n");

      DTM (inst[i],0);

      debug_DTM();
     } while (inst[++i].a != -1);

   //reconfigurable_array();

   return 0;
}

#endif
```

APPENDIX C UMA ARQUITETURA RECONFIGURÁVEL TRANSPARENTE PARA APLICAÇÕES HETEROGÊNEAS

A possibilidade se adicionar cada vez mais e mais transistores dentro de um circuito integrado, de acordo com a lei de Moore, faz com que o aumento de desempenho atinja o mesmo patamar de crescimento. Entretanto, esta lei poderá mudar em um futuro não muito distante, por uma simples razão: os limites físicos do silício estão sendo alcançados. Outro aspecto visível do limite tecnológico atual é o aumento da potência dissipada pelos circuitos integrados (CI), graças às correntes de fuga e ao chaveamento natural de bilhões de transistores. Este fato tem um impacto diferente em computadores de propósito geral, onde sistemas de refrigeração cada vez mais robustos têm de ser usados; ou em sistemas embarcados móveis, onde a energia gasta é o fator principal para aumentar o tempo de utilização do aparelho sem a necessidade de recarga. Além do mais, novas tecnologias que irão substituir completa ou parcialmente o silício estão surgindo. De acordo com o ITRS Roadmap (SEMICONDUCTOR, 2008), estas tecnologias possuem um alto grau de densidade e são lentas comparadas à CMOS, ou o oposto: novos dispositivos podem atingir altas velocidades, mas com uma grande ocupação de área e consumo de potência, mesmo quando levadas em conta tecnologias CMOS futuras.

Em paralelo com a questão tecnológica, arquiteturas tradicionais de alto desempenho, como os difundidos processadores superescalares, estão atingindo seus limites. Como demonstrado em (FLYNN; HUNG, 2005) e (SIMA; FALK, 2004), não há nenhuma novidade arquitetural em tais sistemas nos últimos anos. Ademais, recentes incrementos de desempenho foram ocasionados apenas pelo aumento na freqüência de operação. Entretanto, este recurso também está chegando a um ponto de estagnação. Por exemplo, a freqüência de operação do processador Pentium IV da Intel aumentou apenas de 3,06 para 3,8GHz entre 2002 e 2006 (INTEL, 2008).

Em (OR-BACH, 2001), é discutido o futuro dos processos de fabricação usando novas tecnologias. De acordo com este trabalho, células padrão (*standard cells*), como são utilizadas atualmente, não existirão mais. Como os métodos de fabricação estão mudando, circuitos regulares logo se tornarão uma necessidade. Entenda-se por circuitos regulares aqueles que apresentam uma grande repetição de uma mesma e simples estrutura, seja no nível das portas, células, blocos etc. É também um consenso que a liberdade hoje oferecida para os projetistas, representada pela irregularidade do projeto, será mais cara no futuro. Desta maneira, utilizando lógica regular, as companhias irão reduzir custos, como também a possibilidade da diminuição do número de falhas do circuito, já que a confiabilidade da impressão de geometrias utilizadas hoje em 65 nanômetros já é considerada um grande problema.

Desta maneira, por diferentes razões anteriormente discutidas, tanto no âmbito de sistemas embarcados quanto no de computação de propósito geral, a redução no aumento constante de freqüência junto com os novos limites impostos pelas recentes tecnologias são novos desafios arquiteturais que precisam ser tratados.

Alternativas Arquiteturais

Sistemas Reconfiguráveis

As várias abordagens de exploração do paralelismo utilizadas atualmente recaem no mesmo problema: o paradigma de programação. O modelo Von Neumann traz consigo uma limitação nesta exploração, causada pelo seu modelo orientado a controle (*control-driven*), que tem a sua execução conduzida pelo contador de programa. Máquinas *dataflow*, entretanto, exploram o máximo paralelismo da aplicação utilizando um modelo orientado a dados (*data-driven*). Basicamente, a execução de certa operação se dá quando os dados requisitados para tal estiverem disponíveis. Neste cenário, arquiteturas reconfiguráveis aparecem como uma solução muito atrativa. O fato que motiva a utilização deste tipo de arquitetura é que estas se localizam entre os dois modelos citados anteriormente. Assim, consegue-se obter uma arquitetura implementável atualmente, que explora um alto grau de paralelismo, ainda utilizando compiladores, ferramentas e métodos baseados em Von Neumann já existentes.

Ao mesmo tempo que a computação baseada em reconfiguração pode explorar o paralelismo entre as instruções, ela também pode diminuir o tempo de processamento de instruções dependentes entre si. Esta é a sua maior vantagem em relação às arquiteturas tradicionais utilizadas atualmente. Usando a mesma idéia de reutilização de instruções, transforma-se uma seqüência de operações (parte do código de programa) em um equivalente implementado em circuito combinacional, que executa exatamente as mesmas funções. Assim, aumenta-se o desempenho do sistema (HENKEL, ERNST, 1997) (VENKATARAMANI et al., 2001), reduzindo drasticamente o seu consumo de energia (STITT; VAHID, 2002) – pelo preço do incremento de área ocupada.

A Figura 1 ilustra este processo de forma simplificada. Geralmente, uma arquitetura reconfigurável é formada por uma Unidade Funcional Reconfigurável (Reconfigurable Function Unit - RFU); uma unidade capaz de realizar a reconfiguração da RFU; e um processador de propósito geral (*General Purpose Processor* - GPP). Outra vantagem na utilização de sistemas reconfiguráveis é que estes são altamente regulares, formados por replicações de estruturas idênticas, enquadrando-se exatamente como uma arquitetura a ser utilizada para solucionar os problemas de fabricação e produção, citados anteriormente.

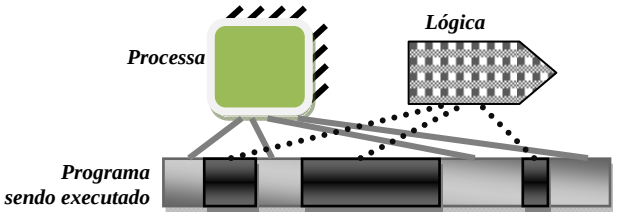


Figura 1 – O funcionamento básico de um sistema reconfigurável

Como exemplo de sistemas reconfiguráveis, podem ser citados os sistemas Chimaera (HAUCK, 1997) e ConCISe (RAZDAN; SMITH, 1994), que possuem uma unidade reconfigurável fortemente acoplada ao núcleo do processador, formada totalmente por lógica combinacional. Esta unidade é, de fato, uma unidade funcional adicional no *pipeline* do processador, compartilhando os mesmos recursos das outras unidades. Esta técnica faz com que o controle da unidade seja simples, diminuindo o gasto adicional (tanto em desempenho quanto potência) requerido na comunicação entre a unidade reconfigurável e o resto do sistema.

Por sua vez, o processador GARP (HAUSE; WAWRYNEK, 1997) é baseado na arquitetura MIPS, e possui uma unidade fracamente acoplada ao processador. Assim, a comunicação com esta unidade é feita através de instruções *move* dedicadas. Outro exemplo é a arquitetura Molen (VASSILIADIS et al., 2001), que possui uma unidade reconfigurável externa, utilizada para fazer otimizações nos núcleos principais do programa.

A técnica de reconfiguração também foi aplicada em outros níveis arquiteturais, impondo mudanças radicais no paradigma de programação, envolvendo o desenvolvimento de novos compiladores e ferramentas. Como um exemplo significativo, o processador TRIPS (SANKARALINGAM et al., 2003) é baseado na execução de blocos (ou conjuntos) de instruções em lógica combinacional. São utilizados pequenos núcleos de granularidade grossa (compostos por nodos, que por sua vez são formados por uma pequena memória com os operandos, uma unidade lógica e aritmética e um roteador). Da maneira pelo qual foram projetados, estes núcleos podem ser agrupados para explorar um vasto conjunto de diferentes tipos de paralelismo, que vão desde o no nível de dados até *threads*. Todavia, toda a análise e alocação destas partes do software nos pequenos núcleos são feitas totalmente pelo compilador. Colocando este conceito ao extremo, pode-se também citar como exemplo representativo o processador Wavescalar (SWANSON et al., 2003), uma arquitetura totalmente *dataflow*.

Tradução Binária

Outra técnica para aumentar o desempenho do sistema e que também pode trazer um baixo consumo de energia é o uso de Tradução Binária (Binary Translation) (ALTMAN; SHEFFER, 2000). Nesta técnica, o sistema por si próprio monitora o código binário do programa que está sendo executado e detecta os núcleos mais executados do software, com o intuito de otimizá-los. Dentre as otimizações existentes relacionadas a esta técnica, estão a recompilação dinâmica e a gravação do resultado de traduções anteriores em uma cache especial. Como exemplo, o processador Crusoe da Transmeta (KLAIBER, 2000) é baseado em um processador VLIW (*Very Long Instruction Word*) onde a aplicação é analisada em tempo de execução, com o objetivo de achar as melhores partes de software para melhor explorar o paralelismo disponível entre as instruções. Uma das grandes vantagens no uso desta técnica é que o processo é transparente, o que significa que não é necessária mudança alguma no código fonte ou binário da aplicação (envolvendo, por exemplo, recompilação), o que consequentemente não causa rompimento algum no fluxo padrão de desenvolvimento de software.

Reutilização de Seqüências de Instruções

O primeiro estudo sobre a reutilização de seqüências de instruções foi feito em (GONZALEZ et al., 1999). A idéia básica é, após a primeira vez que uma seqüência de instruções for executada, guardar o seu contexto (valores de entrada e saída de registradores, endereços de memória, contador de programa etc.) em uma memória especial e, da próxima vez que esta mesma seqüência for encontrada, reutilizá-la através de seu contexto salvo, ao invés de executá-la novamente.

Contudo, o tamanho deste contexto e da seqüência de instruções (chamada de *trace*) pode se tomar enorme, limitando o campo de ação de tal técnica, e incrementando a complexidade do algoritmo responsável pela detecção das instruções e pela sua reutilização. Bons resultados apenas são alcançados quando suposições muito otimistas são feitas, como a reutilização de uma seqüência levar apenas um ciclo de relógio.

Motivações

Como discutido anteriormente, arquiteturas reconfiguráveis aparecem como sérias candidatas para se tornarem uma destas soluções. Entretanto, é necessário ter muito cuidado quando se propõe novas possibilidades arquiteturais, já que há um claro requisito de se manter a compatibilidade de software e paradigmas tradicionais de programação. Estes são fatores chave para reduzir o ciclo de desenvolvimento de um produto, permitindo que ele possa ser lançado o mais rápido possível no mercado. E é este exatamente o maior problema de arquiteturas reconfiguráveis atualmente: são necessários compiladores ou ferramentas especiais, que claramente não sustentam o conceito de portabilidade de software, principalmente no campo de computação de propósito geral. Além do mais, outra restrição faz com que estas arquiteturas não sejam ainda amplamente utilizadas: somente partes específicas de um programa são otimizadas. Assim, apenas programas bem específicos, como aqueles que fazem processamento digital de sinais, são beneficiados por esta técnica – o que não reflete a realidade dos sistemas de propósito geral nem da nova geração de sistemas embarcados.

Solução Proposta

Unificando todas as idéias citadas na seção anterior, em (STITT; VAHID, 2002) (LYSECKY; VAHID, 2004) foram apresentados primeiros estudos sobre os benefícios e a possibilidade de implementação do particionamento dinâmico usando lógica reconfigurável, movendo dinamicamente, em tempo de execução, núcleos do software para a unidade reconfigurável – técnica esta chamada de *Warp Processing*. Entretanto, esta técnica ainda é limitada apenas às partes mais criticas do software, como os laços mais executados, já que sua unidade reconfigurável é implementada em FPGA: há um alto grau de complexidade no algoritmo de detecção e reconfiguração, já que há um enorme número de configurações possíveis que pode ser feita em tal estrutura. Desta maneira, altos ganhos são apenas atingidos em algoritmos que possuem poucas instruções de controle, onde seus núcleos são bastante distintos do resto do programa, como filtros. Algoritmos que possuem mais estruturas de controle ou que possuem um comportamento misto não conseguem tirar proveito de tal técnica. Ademais, o fato da obrigatoriedade do uso de FPGA faz com que a migração do *Warp Processing* para processadores convencionais utilizados atualmente seja mais complicada.

Este trabalho, entretanto, propõe o uso de uma unidade reconfigurável de granularidade grossa, fortemente acoplada ao processador, composta por unidades funcionais simples e multiplexadores que, por ser independente de qualquer tecnologia, não está limitada à alta complexidade de configurações de granularidade fina. Esta unidade reconfigurável é usada em conjunto com uma técnica de Tradução Binária chamada *Dynamic Instruction Merging*, que é usada para detectar seqüências de instruções em tempo de execução para serem enviadas para a unidade reconfigurável. Transformando em lógica combinacional qualquer seqüência de instruções, não ficando limitado às partes críticas do programa, aumentos no desempenho são obtidos mesmo em programas que possuem muito controle ou em programas que não apresentam um alto nível de paralelismo entre suas instruções. É exatamente a natureza de granularidade grossa que faz com que isso seja possível: o algoritmo de detecção dinâmica e de configuração da unidade reconfigurável torna-se mais simples, e uma quantidade menor de memória para guardar estas configurações é necessária. Desta maneira, a principal novidade de tal arquitetura é seu caráter dinâmico: além da unidade reconfigurável ser dinamicamente configurada, as seqüências de instruções a serem executadas nela também são detectadas e transformadas em uma configuração do array em tempo de execução.

Contribuições

Além da unidade reconfigurável, o *hardware* especial foi desenvolvido com o objetivo de fazer a transformação. Quando esta unidade percebe que há um certo número de instruções que podem ser executadas na unidade, uma tradução binária é aplicada a esta seqüência. Esta tradução transforma a seqüência original de instruções em uma configuração da unidade reconfigurável, que por sua vez desempenha exatamente a mesma função desta seqüência. Após isto ocorrer, esta configuração é gravada em uma memória especial, chamada *cache* de reconfiguração. Na próxima vez que esta seqüência for achada, a unidade irá executar esta configuração na unidade reconfigurável ao invés da seqüência normal de instruções do processador. Desta vez, a análise de dependência não é mais necessária: o processador simplesmente necessita recuperar as informações de configuração da memória especial. Desta forma, dependendo do tamanho desta *cache* especial, o aumento de desempenho pode ser estendido para qualquer parte do programa, não apenas nos laços mais executados do mesmo.

Na primeira parte do trabalho desenvolvido, a técnica foi acoplada ao Femtojava (BECK; CARRO, 2003), um processador para sistemas embarcados que executa nativamente bytecodes da linguagem Java (BECK; CARRO, 2005) (BECK; CARRO, 2005B). Foi mostrado um grande aumento de desempenho e redução no consumo de energia, mesmo quando comparada a versões VLIW da mesma arquitetura (BECK; CARRO, 2004). Os dados de desempenho e potência foram obtidos através de simulação, utilizando uma ferramenta configurável ciclo-a-ciclo (BECK; CARRO, 2003B). Somando-se a isso, foi mostrado também que a técnica de Dynamic Instruction Merging pode ser beneficiada do método particular de computação de máquinas de pilha, podendo assim detectar e transformar as instruções com um baixo nível de complexidade (BECK; CARRO, 2005B). Esta também foi comparada com métodos tradicionais de detecção de máquinas RISC (GOMES et al., 2005) (GOMES et al., 2005B).

Após, o trabalho foi focado na implementação da mesma técnica em arquiteturas RISC. Como poderia ser esperado, há diferenças tanto na estrutura da unidade reconfigurável como no algoritmo de detecção, já que anteriormente uma máquina de pilha foi utilizada. Primeiramente, a técnica foi implementada usando a ferramenta Simplescalar (BURGER; AUSTIN, 1997) junto com o conjunto de benchmarks MIBench (GUTHAUS et al., 2001). Desta forma, foi possível comparar a técnica proposta com processadores superescalares (BECK et al., 2007) (BECK et al., 2006) (BECK et al., 2006b). Foi demonstrado que houve aumento de desempenho em relação a estes, resolvendo alguns problemas conhecidos de limitação de paralelismo. O terceiro estudo de caso foi a implementação da técnica em um processador MIPS R3000, amplamente utilizado em sistemas embarcados (BECK et al., 2008). Novamente, ótimos resultados foram alcançados. Outros trabalhos periféricos também foram desenvolvidos: uma ferramenta que automaticamente, dependendo do paralelismo disponível na aplicação, constrói a unidade reconfigurável composta por um número mínimo de unidades funcionais que explora tal paralelismo, resultando em uma grande diminuição na área e na potência do circuito reconfigurável (RUTZIG et al., 2008).