

Chapter 2

Reconfigurable Systems

Abstract As previously discussed, it is possible to take advantage of reconfigurable computing to overcome the main problems that nowadays architectures are facing. Therefore, this chapter aims to explain the basics of reconfigurable systems. It starts with a basic explanation on how these architectures work, their main principles and steps. After that, the principle of merged instruction is introduced, showing how a reconfigurable unit can increase the IPC and affect the number of instructions issued and executed per cycle. The second part of this chapter starts with an overview on the classification of reconfigurable systems, including granularity, instruction types and coupling. Finally, the chapter presents a detailed analysis of the potential gains that reconfigurable computing can present, discussing the main differences, advantages and drawbacks of fine and coarse grain reconfigurable units.

2.1 Introduction

Reconfigurable systems are those architectures that have the capability to adapt themselves to a given application, providing some kind of hardware specialization to it. Through this adaptation, they are expected to achieve great improvements, in terms of performance acceleration and energy savings, when compared to general purpose, 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) [30] or Application-Specific Integrated Circuits (ASICs) [36].

As an example, let us consider an old ASIC, the STA013. It is an MP3 decoder produced by ST Microelectronics few years ago. It can decode music, at real time, running at 14.7 MHz. Can one imagine the last Intel General Purpose Processor (GPP) decoding an MP3 at real time with that operating frequency? The chip provided by ST is cheaper, faster and consumes less power than any processor that could perform the same task at real time. However, it cannot do anything more than MP3 decoding. For complex systems found nowadays, with a wide range of different applications being executed on it, the Application-Specific approach would

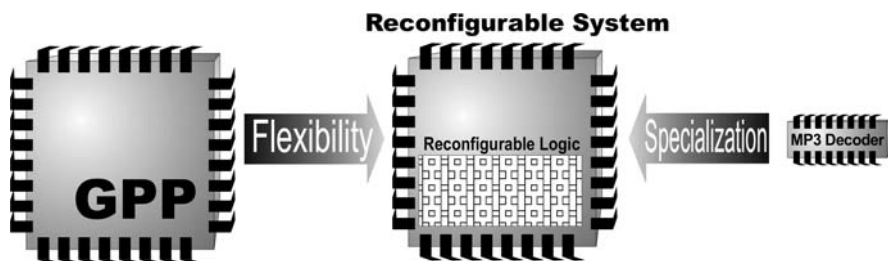


Fig. 2.1 Reconfigurable systems: hardware specialization with flexibility

lead to a huge die size, becoming very expensive, since a large number of hardware components would be necessary. On the other hand, a GPP would be able to execute everything, but it is very likely that it would not satisfy either performance or energy constraints of this system.

Reconfigurable architectures were created exactly to fill the gap between specialized hardware and general purpose processing with generic devices. This way, a reconfigurable architecture can be viewed as an intermediate approach between an Application-Specific hardware and a GPP, as Fig. 2.1 illustrates. A reconfigurable system could be configured according to the task at hand, meeting the aforementioned system constraints with a reasonable area occupation, and still being useful for other general-purpose applications. Hence, as Application-Specific components have specialized hardware that accelerate the execution of the applications they were designed for, a system with reconfigurable capabilities would have almost the same benefit without having to commit the hardware into silicon for just one application: computational structures could be adapted after design, in the same way programmable processors can adapt to application changes.

It is important to discuss why reconfigurable architectures can be useful in another point of view. First, let us remember that current architectures used nowadays are based on the Von Neumann model. The problem there is that the Von Neumann model is control-driven, meaning that its execution is based on the program counter. This way, these architectures are still withheld by the so-called Von Neumann bottleneck. Besides representing the data traffic problem, it also has kept people tied to word-at-a-time thinking, instead of encouraging one to think in terms of the larger conceptual units of the task at hand. In contrast, dataflow machines are data-driven: the execution of a given part of the software starts soon after the data required for such operation is ready, so they can explore the maximum parallelism available in the application. However, the employment of dataflow machines implies in the use of special compilers or tools and, most importantly, it changes the programming paradigm. The greatest advantage of reconfigurable architectures is that they can merge both concepts, making possible the use of the very same principle of dataflow architectures, but still using already available tools and compilers, maintaining the programming paradigm.

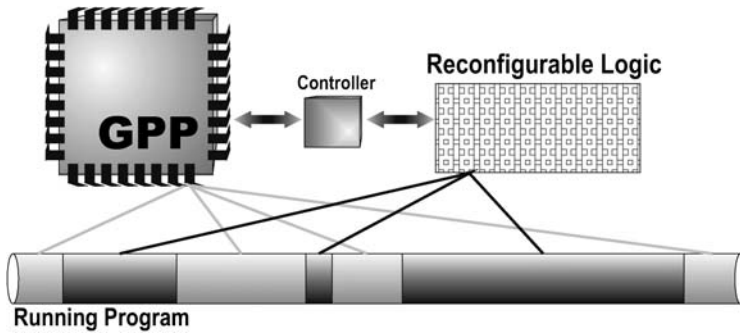


Fig. 2.2 The basic principle of a system making use of reconfigurable logic

2.2 Basic Principles

As already discussed, a reconfigurable architecture is the system that has the ability to adapt itself to perform several and different hardware computations, according to the needs of a given program. This program will not be necessarily always the same. In Fig. 2.2, the basic principle of a computational system working together with a reconfigurable hardware is illustrated. Usually, it is comprised of a reconfigurable logic implemented in hardware, a special component to control and reconfigure it (sometimes it is also responsible for the communication mechanism), a context memory to keep the configurations, and a GPP. Pieces of code are executed on reconfigurable logic (gray), while others are executed by the GPP (dark). The main challenge is to find the best tradeoff considering which pieces of code should be executed on reconfigurable logic. The more software is being executed on reconfigurable logic the better, since it is being executed in a more efficient manner. However, there is a cost associated to it: the need for extra area and memory, which are obviously limited resources.

Systems provided of reconfigurable logic are often called Reconfigurable Instruction Set Processors (RISP) [22], and they will be the focus of this and the next chapters. The reconfigurable logic includes a set of programmable processing units, which can be reconfigured in the field to implement logic operations or functions, and programmable interconnections between them.

2.2.1 Reconfiguration Steps

To execute a program taking advantage of the reconfigurable logic, usually the following steps are necessary (illustrated in Fig. 2.3):

1. *Code Analysis*: the first thing to do is to identify parts of the code that can be transformed for execution on the reconfigurable logic. The goal of this step is to find the best tradeoff considering performance and available resources regarding

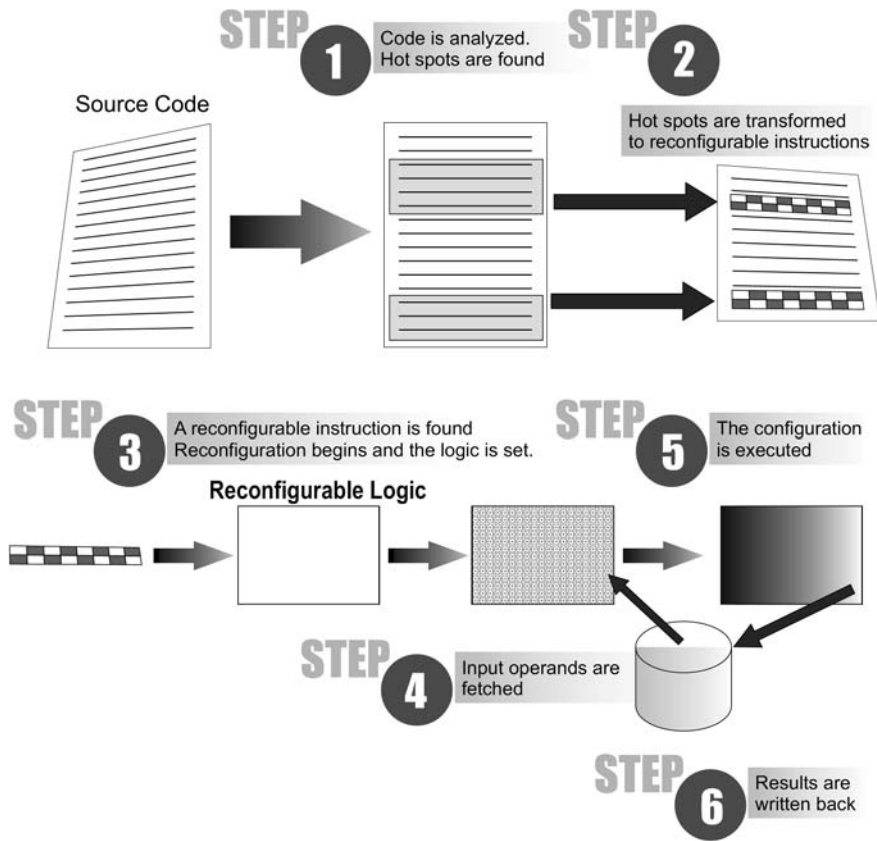


Fig. 2.3 Basic steps in a reconfigurable system

the reconfigurable unit (RU). Usually, the code is not analyzed statically: an execution trace that was previously generated is employed, so dynamic information can be extracted, since it is hard to figure (sometimes impossible) the most executed kernels by just analyzing the source or assembly code. This step can be performed either by automated tools or manually by the designer.

2. *Code transformation:* Once the best candidate parts of code to be accelerated (named as *hot spots* or *kernels*) are found, they need to be replaced by reconfigurable instructions. The reconfigurable instructions will be handled by the control unit of the reconfigurable system. The source code of the processor can also be modified to explicitly communicate with the reconfigurable logic, using native processor instructions.
3. *Reconfiguration:* After code transformation, it is time to send it to the reconfigurable system. When a reconfigurable instruction is found, the programmable components of the reconfigurable logic are organized as a function according to that instruction. This is achieved by downloading from a special memory a set of configuration bits, called *configuration context*. 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.

4. *Input Context Loading*: To perform a given reconfigurable operation, a set of inputs is necessary. They can come from the register file, a shared memory or even be transmitted using message passing.
5. *Execution*: After the reconfigurable unit is set and the proper input operands are ready, execution begins. The operation will be executed in a more efficient manner in comparison with the execution on a GPP.
6. *Write back*: The results of the reconfigurable operation are saved back to the register file, to the memory or transmitted from the reconfigurable unit to the reconfigurable control unit or GPP.

Steps 3 to 6 are repeated while reconfigurable instructions are found in the code, until the end of its execution.

2.3 Underlying Execution Mechanism

To understand how the gains are obtained by the employment of reconfigurable logic, let us start with a very simple example, considering that one wants to build a circuit to multiply a given number by the constant seven. For that, the designer has only two available components: adders and registers. The first choice is to use just one adder and one register (Fig. 2.4a). The result would be generated by repeating seven times the sum operation, so six cycles would be necessary, considering that the register had been reset at the beginning of the operation.

Another choice is to completely replace sequential for combinational logic, eliminating the register and putting six adders directly connected to each other

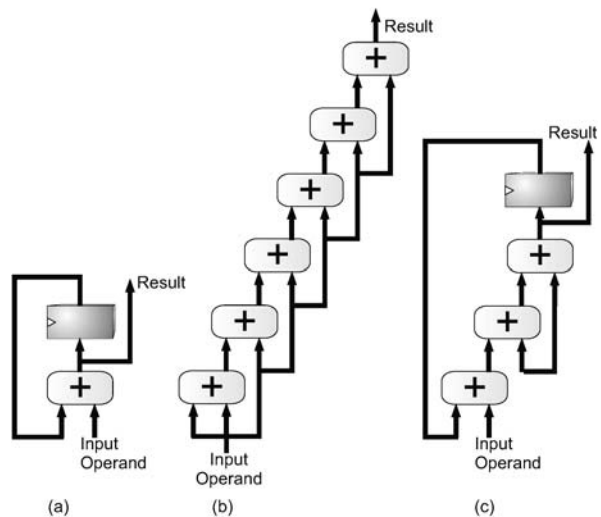


Fig. 2.4 Different ways of performing the same computation

(Fig. 2.4b). The critical path of the circuit will increase, thereby increasing the clock period of the system. However, when considering the total execution time, the second option will be faster, since setup and hold times of the register have been removed. In a certain way, this represents the difference between control and data driven executions commented before. In the first case, the next computation will be performed at the next cycle. In the second case, the next computation will start soon after the previous one was ready.

One could write that the Execution Time (ET) for an algorithm mapped to hardware is

$$A_{sequential} = n * A_{cell} + A_{control} + A_{registers} \quad (2.1)$$

$$ET = number_{cycles} * cycle_{time} \quad (2.2)$$

And for the hardware algorithm of figure (Fig. 2.4a) one has

$$ET_a = 6 * [T_{pFF} + T_{adder} + T_{set}] \quad (2.3)$$

and for (Fig. 2.4b) one has

$$ET_b = 1 * [6 * T_{adder}] \quad (2.4)$$

and one immediately verifies the second case is faster because the delays of the flip-flops are not in the critical path. However, since one is dealing with combinational logic, one could further optimize by substituting the adder chain by an adder tree, as in Fig. 2.4c, and hence the new execution time would be given by

$$ET_c = 2 * [3 * T_{adder}] \quad (2.5)$$

This would be a compromise of both aforementioned examples. However, the main idea remains the same: to replace, in some level, sequential for combinational logic to group a sequence of operations (or instructions) together. It is interesting to note that in real life circuits, sometimes putting more combinational data to work in a sequential fashion would not increase the critical path, since this path could be localized somewhere else. In some processors, for example, the functional units are not responsible for the critical path of the circuit, so grouping them together may be a good idea.

This way, grouping instructions together to be executed in a more efficient mechanism is the main principle of any kind of application specific hardware, such as ASIP or ASIC. More area is occupied and, consequently, more power is spent. However, one should note that fewer flip-flops are used, and these are a major source of power dissipation. Moreover, as less time is necessary to compute the operations (hence there are performance gains), it is very likely that there will be also energy savings.

Now, let us take the same adder chain presented before, and replace the adders for complete ALUs. Besides, different values can be used as input for these new

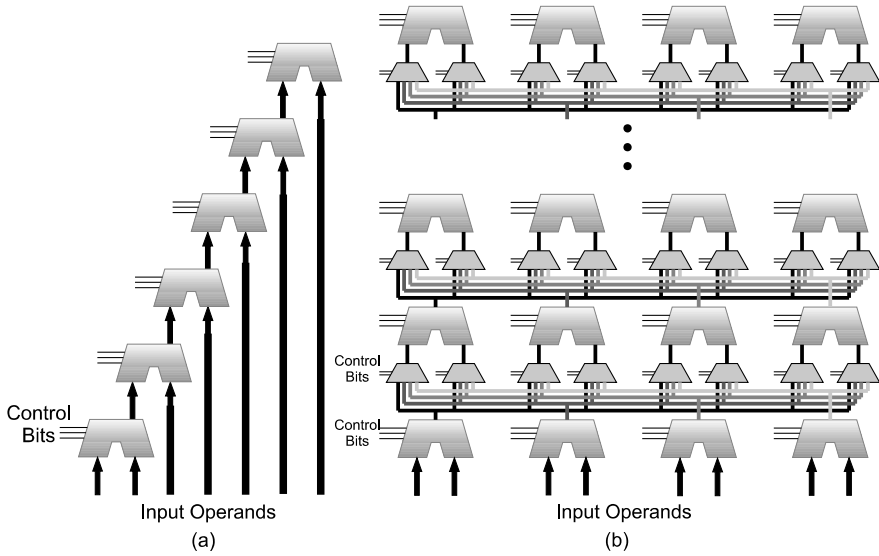


Fig. 2.5 Principles of reconfiguration

ALUs, as it can be observed in Fig. 2.5a. More area is being spent, and it is very likely that the circuit will not be fast as it was before (for example, at least one multiplexer was added at the end of the ALU to select which operation to send as output). Moreover, more control circuitry is necessary (to configure the ALUs). On the other hand, now there is certain flexibility: any arithmetic and logic operation can be performed. Extending this concept even more, it is possible to add ALUs working in parallel, and multiplexers to route the values between them (Fig. 2.5b). Again, the critical path increases, even more control hardware is necessary, but there is still more flexibility, besides the possibility of executing operations in parallel. The main principle remains the same: to group instructions to be executed in a more efficient manner, but now with some flexibility. This is, in fact, an example of a coarse grain reconfigurable array, and it will be seen in more details later in this chapter.

Figure 2.6 graphically shows the difference between using reconfigurable logic and a traditional parallel architecture to execute instructions. The upper part of the figure demonstrates the execution of several instructions on a traditional parallel architecture, such as the superscalar ones. These instructions are represented as boxes. Those that have the same texture represent instructions that are data dependent and hence cannot be executed in parallel, while non-dependent instructions can be executed concurrently. There is a limit, though: no matter how many functional units are available, sequences of dependent instructions must be executed in order. On the other hand, by using the data-driven approach and combinational logic, one is able to reduce the time spent by executing exactly the sequences of dependent instructions in a more efficient manner (avoiding the flip-flop delays in the reconfigurable logic),

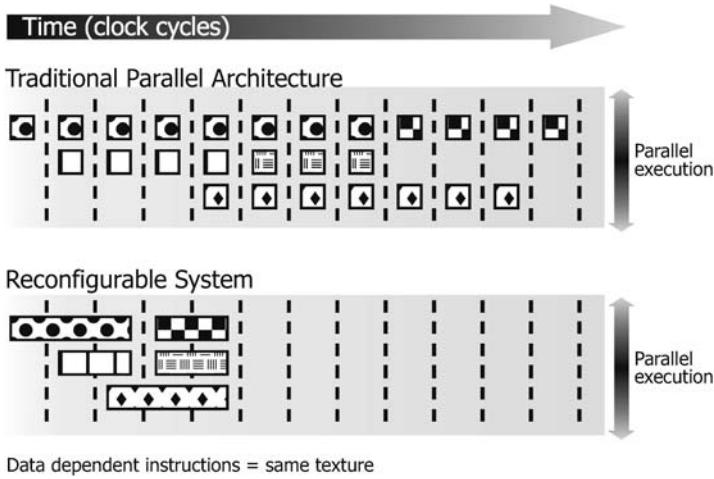


Fig. 2.6 Performance gains obtained when using reconfigurable logic

at the cost of extra area. Consequently, as a legacy of dataflow machines, reconfigurable systems, besides being able to explore the parallelism between instructions, can also speed up instructions which are data dependent between themselves, in opposite to traditional architectures.

2.4 Advantages of Using Reconfigurable Logic

The widely used Patterson [27] metrics of relative performance through measures such as IPC (Instructions Per Cycle) are well suited for comparing different processor technologies and ISA (Instruction Set Architecture), as it abstracts concepts such as clock frequency. As described in [34], however, to better understand the performance evolution in the microprocessor industry, it is interesting to note the *Absolute Processor Performance (Ppa)* metric denoted as:

$$Ppa = fc * 1 / CPII * IPHI * OPI(\text{operations/sec}) \quad (2.6)$$

In (2.6), *CPII*, *IPHI* 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 *IPHI* variable is related to the issue parallelism, or the average number of dynamically fetched instructions issued to execution per issue interval. Therefore, temporal (*CPII*) and

issue (*IPII*) parallelisms can be illustrated by the following equations:

$$IPII = (\text{Number of Instructions}) / (\text{Number of issues}) \quad (2.7)$$

$$CPII = (\text{Number of Cycles}) / (\text{Number of Issues}) \quad (2.8)$$

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 notice 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, while the second is related to the number of instructions that are dynamically issued to be executed in parallel, such as the ones sent for execution in a superscalar processor after scheduling. Figure 2.7 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 (2.6). One of these approaches, for example, deals with the *CPII* metric by increasing instructions throughput with pipelining [27]. Moreover, the *CPII* metric has also been well covered with efficient branch prediction mechanisms and memory hierarchies, though this metric is still limited by pipeline stalls such as the ones caused by cache misses. The *OPI* rate has 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.

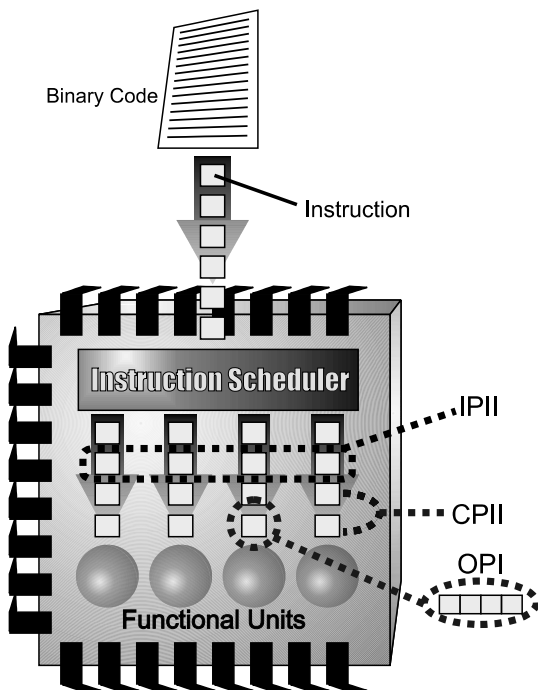


Fig. 2.7 Gains obtained when using reconfigurable logic

2.4.1 Application

A reconfigurable system targets to increase exactly the *IPII* rate. As can be observed in (2.7), in order to increase the *IPII* number, it is necessary to increase the execution efficiency by decreasing the number of issues. Considering that a sequence of instructions is identified and grouped to be executed on the reconfigurable system, more instructions will be issued by issue interval (so increasing the *IPII* rate). Equation (2.9) shows how the number of issues is affected by the technique:

$$\text{Number of Issues} = \text{Total number of executed Instructions} + \text{NMI} * (1 - \text{AMIL}) \quad (2.9)$$

where the *Average Merged Instructions Length (AMIL)* is the average group size in number of instructions; while the *Number of Merged Instructions (NMI)* counts how many merged instructions¹ were issued for execution on the combinational logic. This can be represented by the following equation:

$$\text{NMI} = \text{MIR} * \text{Total number of executed Instructions} \quad (2.10)$$

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 for a processor, which is actually the case of specialized hardware (ASIPs or ASICs) 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 high 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 on 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 a group of instructions.

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

2.4.2 An Instruction Merging Example

The following example illustrates the concept previously proposed.

Figure 2.8a 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 a given GPP

¹In this chapter the set of instructions that are executed on reconfigurable hardware is called merged instructions; in previous works, several and different names have been used.

architecture has an $IPII$ rate of one, typical of RISC scalar architectures, and that *inst d* causes a pipeline stall of 5 cycles (for instance, this instruction must wait for the result of another one, in a typical case of true data dependence), while all other instructions are executed in one cycle, this trace of 14 instructions would take 18 cycles to execute. This results in a $CPII$ of 1.28 and an IPC of 0.78.

If, however, instructions of number one to five are merged (which is represented by *Inst M*, as shown in Fig. 2.8b, and executed in two cycles, the whole sequence would then be executed in 14 cycles. Note that the left column in Fig. 2.8b represents

Number	Instruction	Cycle	
1	inst a	1	$CPII = 1.28$ $IPII = 1$
2	inst b	2	
3	inst b	3	
4	inst a	4	$IPC = 0.78$
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	

(a)

Issue	Instruction	Cycle	
1	inst M	1	$CPII = 1.5$ $IPII = 1.4$
2	inst b	3	
3	inst a	4	
4	inst a	5	$IPC = 1$
5	inst b	6	
6	inst d	11	
7	inst e	12	
8	inst b	13	
9	inst c	14	
10	inst a	15	

(b)

Issue	Instruction	Cycle	
1	inst M	1	$CPII = 1.375$ $IPII = 1.75$
2	inst b	3	
3	inst a	4	
4	inst a	5	$IPC = 1.27$
5	inst M2	8	
6	inst b	9	
7	inst c	10	
8	inst a	11	

(c)

Fig. 2.8 (a) Execution trace of a given application;
(b) Trace with one merged instruction; (c) Trace with two merged instructions

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 Fig. 2.8b) 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. In this example, the fact of executing these instructions in a dataflow manner would mask the delay effects of data dependency. This is illustrated in Fig. 2.8c. This way, when using a reconfigurable system, the interval of execution between a set of instruction and another can be longer than the usual. However, as more instructions are executed per time slice, IPC increases.

Later in this chapter, an ideal solution is analyzed, which is capable of executing merged instructions in just one cycle, meaning that the $CPII$ inside the reconfigurable fabric is 1. This will show the potential gains of using reconfigurable logic when affecting the $AMIL$ and $IPII$ rates.

2.5 Reconfigurable Logic Classification

In the reconfigurable field, there is a great variety of classifications, as it can be observed in some surveys published about the subject [22, 25, 35, 37]. In this book, the most common ones are discussed.

2.5.1 Code Analysis and Transformation

This subject concerns how the best hot spots are found in order to replace them with reconfigurable instructions (transforming the code) and the level of automation of this process.

Code analysis can be done in the binary/source code, or yet in the trace generated from the execution of the program on the target GPP. The designer can find the hot spots analyzing the source code (looking for loops with great number of interactions, for instance), or the trace. The greatest advantage of using the trace is that it contains dynamic information. For instance, the designer cannot know if loops with non-fixed bounds are the most used ones by only analyzing the source code. The designer can also benefit from automated tools to do this job. These tools usually work on the trace and can indicate to the designer which are the most executed kernels.

After the hot spots were found, it is time to replace them with reconfigurable instructions. These instructions are related to the communication, reconfiguration

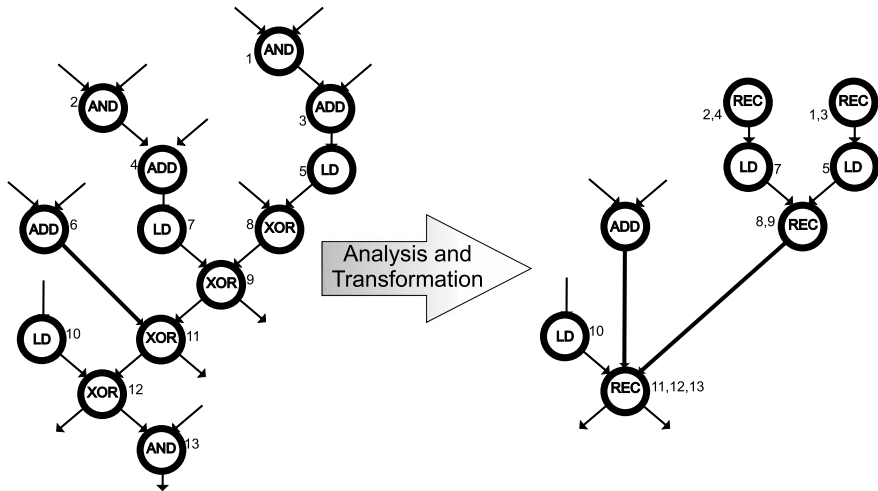


Fig. 2.9 Analysis and transformation of a code sequence based on DFG analysis

and execution processes. Again, the level of automation is variable. It could be the designer's responsibility the whole work of replacing the hotspots with reconfigurable instructions directly in the assembly code. Yet, code annotation can be used. For instance, macros can be employed in the source code to indicate that there will be a reconfigurable instruction. The assembler then will be used to automatically generate the modified code. Finally, there is the complete automated process: given a set of constraints related to a given reconfigurable architecture, a tool will obtain information about the most used hot spots and transform them to reconfigurable instructions, handling issues such as communication between the GPP and reconfigurable logic, reconfiguration overheads, execution and write back of results. It is important to note that such tools are highly dependent to the reconfigurable system they were built to be used with.

Automated tools usually involve some complex graph analysis in order to find the best alternatives for code transformation. To better illustrate this, let us consider an example based on [24], demonstrated in Fig. 2.9. As it can be observed, the sequence of instructions is organized in a DFG (Data Flow Graph). Some sequences are merged together and transformed to a reconfigurable instruction.

These automated tools sometimes can also include another level of code transformations. These happen before code analysis, and are employed to better expose code parallelism, using compiler techniques such as superblock [29] or hyperblock [31].

2.5.2 RU Coupling

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

The position of the reconfigurable logic, relative to the microprocessor, directly affects performance. The benefit obtained from executing a piece of code on it depends on communication and execution costs. The time necessary to execute an operation on the reconfigurable logic 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 standalone processor, then an improvement can be obtained.

The reconfigurable logic 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 is usually 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, having full access to the processor's registers. Some part of the processor (usually the decoder) is responsible to activate the reconfigurable logic, when necessary.

Figure 2.10 illustrates these three different types of coupling. The two first interconnection schemes are usually called loosely coupled. The functional unit approach, in turn, is named tightly coupled. As stated before, the efficiency of each technique depends on two things: the time required to transfer data 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 reconfigurable logic. Usually, loosely coupled units can execute larger chunks of code, and are faster than the tightly coupled ones, mainly because they have more area available. For loosely coupled units, there is a need for faster execution times: it is necessary to overcome some of the overhead brought by the high delays caused by the data transfers. The data exchange is usually performed using shared memory, while the communication can be done using shared memory or message passing.

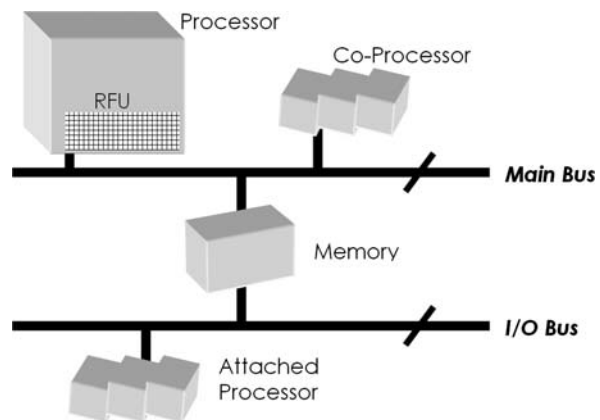


Fig. 2.10 Different types of RU coupling

A tightly coupled RU, although increasing the area taken by the processor itself, makes the control logic simpler. Besides, it minimizes 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, which is usually employed for the communication between the main processor and the reconfigurable unit. On the other hand, the tightly coupled functional unit must run fast enough in order to avoid increasing the processor's cycle time, and hence the amount of logic that can be packed is limited.

When there is a reconfigurable unit 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 and more transistors available within the same die, the RFU based system is becoming a very common implementation.

2.5.3 Granularity

The granularity of a reconfigurable unit defines its level of data manipulation: the smallest possible parts for reconfiguration (or building blocks) for fine-grained logic are usually gates (efficient for bit level operations), while in coarse-grained RUs these blocks are larger (like ALUs), 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. They are usually implemented with or based on FPGA. An example of an FPGA architecture is shown in Fig. 2.11. It consists of a 2-D array of Configurable Logic Blocks (CLBs) used to implement both combinational and sequential logics. Each CLB typically consists of a 4-input lookup table (LUT) and a flip-flop. The lookup table is responsible for executing a given logic operation, so it can implement any 1-bit logic function. Programmable interconnects surround CLBs, ensuring the communication between them. These interconnections can be either direct connections via programmable switches (e.g., pass transistors) or a mesh structure using Switch Boxes (S-Box), as illustrated in the example. Finally, programmable I/O cells surround the array, for communication with the external environment.

A coarse-grain reconfigurable system, in turn, consists of reconfigurable PEs that implement word-level operations and special-purpose interconnections retaining enough flexibility to map different applications onto the system. While bit-oriented algorithms can take better benefit from fine-grained systems, the coarse-grain approach may be the best alternative for computation intensive applications. Coarse grain architectures are implemented using off the shelf functional units, or yet special functional units targeted to a given domain of application. The interconnection resources are usually multiplexers, crossbars or buses. Figure 2.12 illustrates a simple reconfigurable array of functional units, connected to each other using crossbars. The word length of this array is 16 bits long.

Fig. 2.11 A typical FPGA architecture

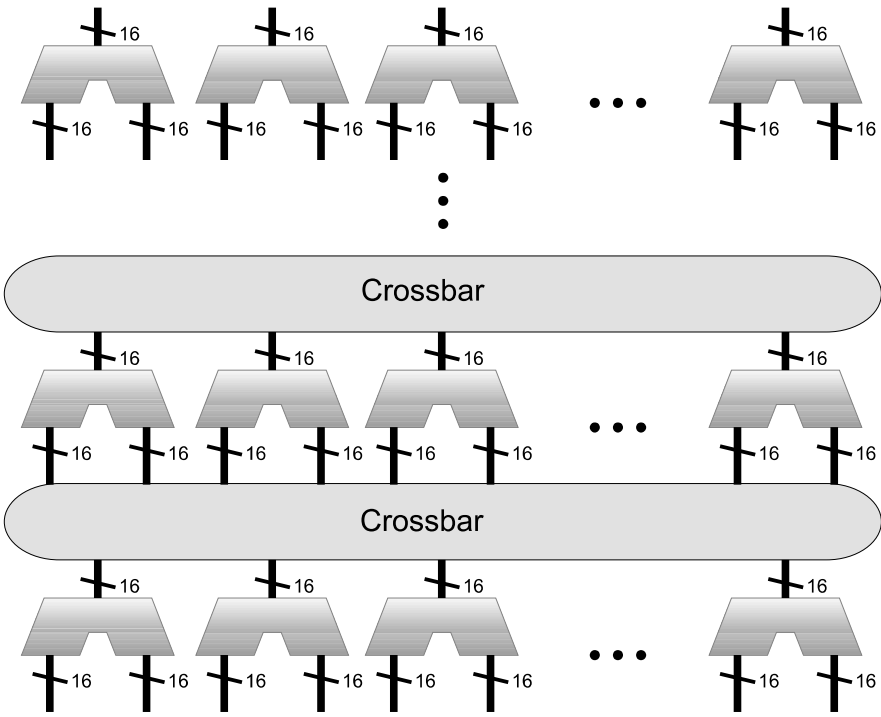
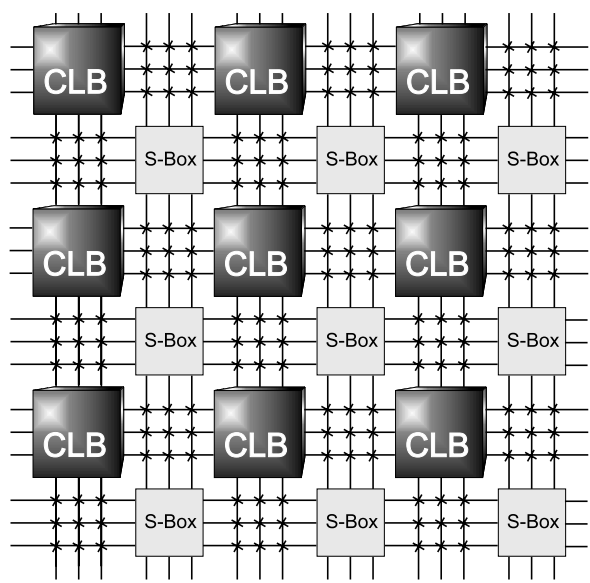


Fig. 2.12 An example of a coarse grain reconfigurable array of functional units

Granularity also affects the size of the configuration context 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 reconfigurable 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.

2.5.4 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 usually 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. This field can give two different kinds of information:

- *Address*: The special field indicates the memory address of the configuration data.
- *Instruction number*: An instruction identifier of small length is embedded in the instruction word. This identifier indexes a configuration table where some information, such as the configuration data address, is stored. The number of reconfigurable instructions is limited to the size of the table.

The first approach needs more instruction word bits, but has the benefit that the number of different instructions is not limited to the size of a table, as in the second case. On the other hand, when using the configuration table approach, the table can be changed on the fly, so the processor can more easily 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.

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 reconfigurable unit 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.

The major drawback of a dedicated register file is that more control for synchronizations is necessary.

In the same way, 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 particular set of applications can benefit from this type, such as FIR filtering, discrete cosine transformation (DCT) or other signal processing related algorithms. 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. Instructions can also be classified in many other ways, such as execution time, pipelining, internal state etc. For more details on these classifications, refer to [25].

2.5.5 Reconfigurability

The logic can be programmed at different moments. If it can only be programmed at startup, before execution begins, this unit is not reconfigurable (it is configurable). If the logic can be programmed after initialization, then it is called reconfigurable. The application can be divided in different blocks of functionality, so the RU can be reconfigured according to the needs of each individual block. In this manner, the program adaptation is done in a per block basis. The reconfigurable logic is simpler to be implemented if the fabric is blocked during reconfiguration. However, if the RU can be used while being reconfigured, it is possible to increase performance. This can be done, for example, by dividing the RU in segments that can be configured independently from each other. The process of reconfiguring just parts of the logic is called partial reconfiguration [25].

Reconfiguration times depend on the size of the configuration data. The configuration data is usually composed of the configuration bits for the unit reconfiguration, as well as information about the input context. These times depend on the configuration method used. For instance, in the PRISC processor [21], the RU 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. Hence, prefetching the configuration data can reduce the time the processor is stalled waiting for reconfiguration. The employment of this approach can be done automatically by software tools [32].

2.6 Directions

In this sub-section, some tradeoffs that should be taken into account while developing a reconfigurable architecture are analyzed. First, a known benchmark set is

evaluated in order to figure what is the best strategy to take in terms of granularity. Then, the impact of this analysis in both fine and coarse grain reconfigurable systems performing high levels simulations is studied. Finally, other issues are considered, such as reconfiguration and execution times, and the growing number of applications being executed at the same time on a system.

2.6.1 Heterogeneous Behavior of the Applications

In [23] a subset of the Mibench Benchmark Suite [26] is used, which represents a complete set of diverse algorithm behaviors. As a matter of fact, this suite has been chosen because, according to [26], it has a larger range of different behaviors when compared against other benchmark sets, e.g. SPEC2000 [28]. 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 Fig. 2.13, the *RawAudio Decoder* algorithm is the most control flow oriented one (a high rate of executed branches) while the *Rinjdael Encoder* is quite the opposite, being data flow oriented. 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.

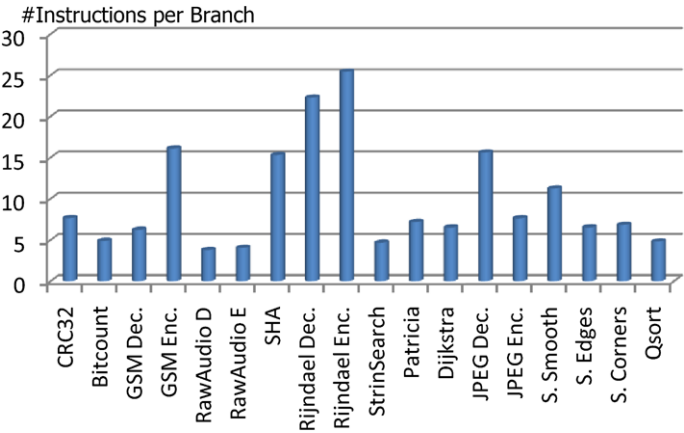


Fig. 2.13 Instruction per Branch Rate

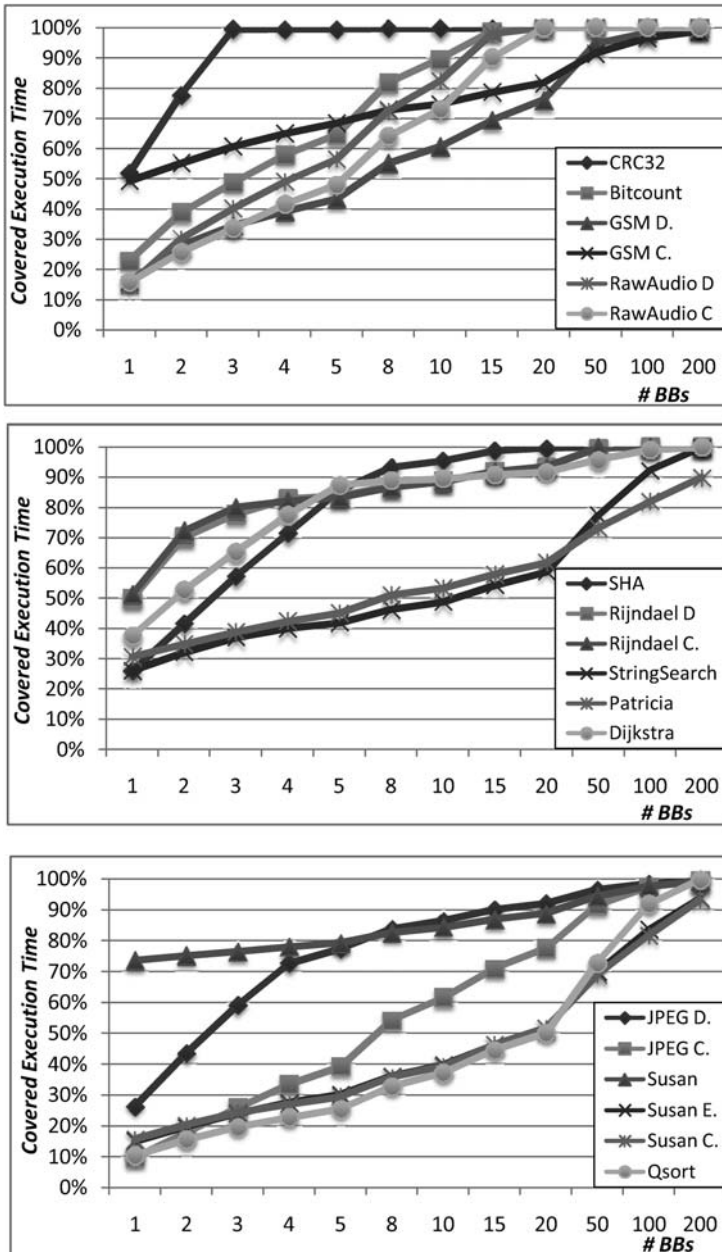


Fig. 2.14 How many BBs are necessary to cover a certain amount of execution time?

Figure 2.14 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.

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 factor of 2x, according to Amdahl's law, all 20 different basic blocks should be mapped into reconfigurable logic, and each should be accelerated by a factor of 4. 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 2.15 illustrates this scenario by using the cases with 1, 3 and 5 basic blocks. Note that, mainly when one considers only the most executed basic blocks (first bar of each benchmark), the shape of the graph is very similar to the instructions per branch ratios shown in Fig. 2.13 (with some exceptions, such as the *CRC32* or *JPEG decoder* algorithms). A deeper study about this issue could be envisioned to indicate some directions regarding the reconfigurable arrays optimization just based on very simple profile statistics.

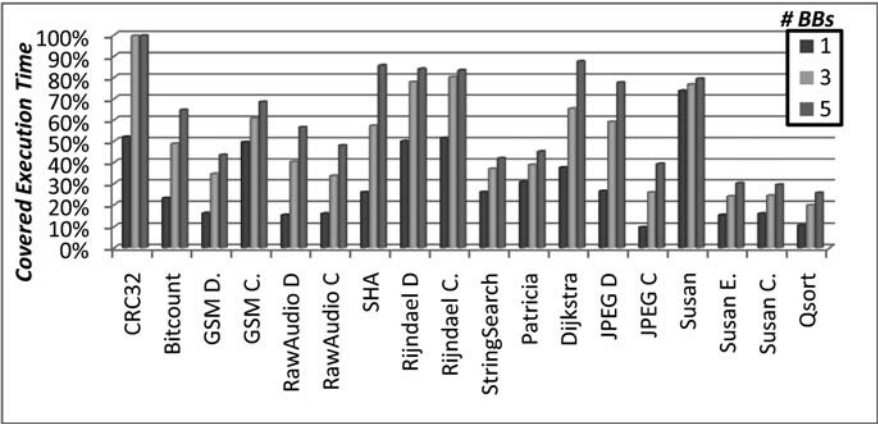


Fig. 2.15 Amount of execution time covered by 1, 3 or 5 basic blocks in each application

2.6.2 Potential for Using Fine Grained Reconfigurable Arrays

In this section, the potentiality of fine grain reconfigurable arrays is evaluated. Considering hot spots as being loops or subroutines, the level of performance gains one can obtain whenever a determined number of them 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 that is common to all iterations. For example, let us assume that a loop is executed 50 times. 100% of the code is executed 49 times, but only 20% is executed 50 times. This way, just this part will be optimized, since it comprises the common part of code executed in all loop iterations. Figure 2.16 illustrates this case. Moreover, subroutines called inside loops are not suited for optimization.

Figures 2.17a and b show, in the y-axis, the performance improvements (speedup factor) when implementing a different amount 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 (considering how many times it is repeated and its number of instructions). It is assumed that the execution time for each one of these hot spots would be of one cycle, when reconfigurable hardware is used. Although extremely optimistic, this can give us an upper bound on the execution time. 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 of the reasons for the lack of optimization in some algorithms is the methodology used for code allocation on the reconfigurable logic, explained above. This way, even if there is 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, whenever heterogeneity on the application set comes into play. Consequently, human interaction for changing and adapting parts of the code is required, with obvious impact on design time costs and time-to-market.

In the first experiment described above, besides considering infinite hardware resources and no communication overhead between the processor and the reconfigurable logic, it has also been assumed an infinite number of memory ports with zero

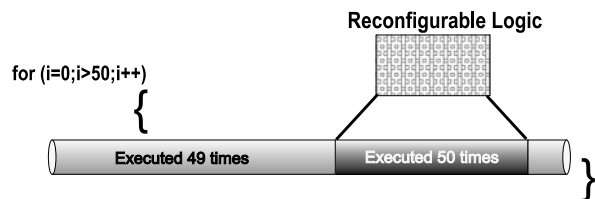


Fig. 2.16 Just a small part of the loop can be optimized

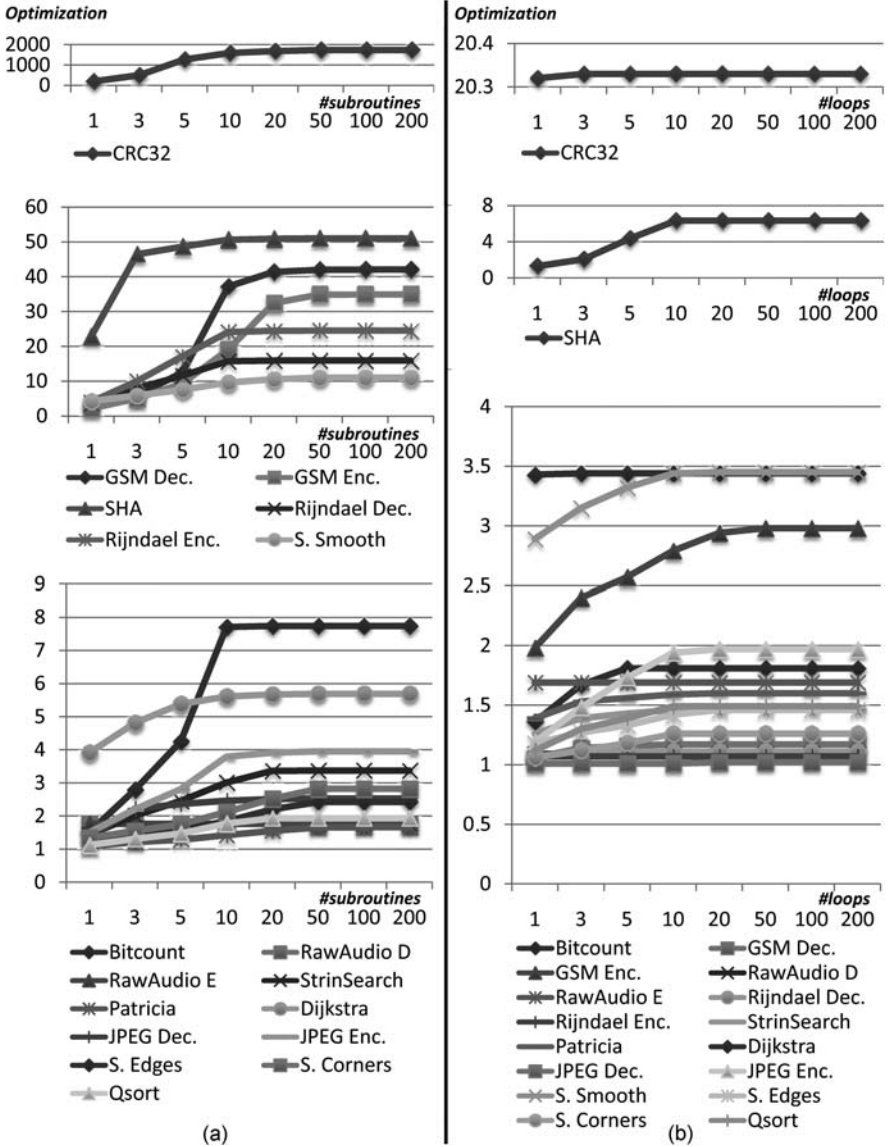


Fig. 2.17 Performance gains considering different numbers of (a) subroutines and (b) loops being executed in 1 cycle in reconfigurable logic

delay, which is practically infeasible for any relatively complex configuration. Now, in Fig. 2.18, a more realistic assumption is considered: each hot spot would take 5 cycles to be executed on the reconfigurable logic. These extra cycles were added to give a hint on the impact of limited memory ports. When comparing this experiment

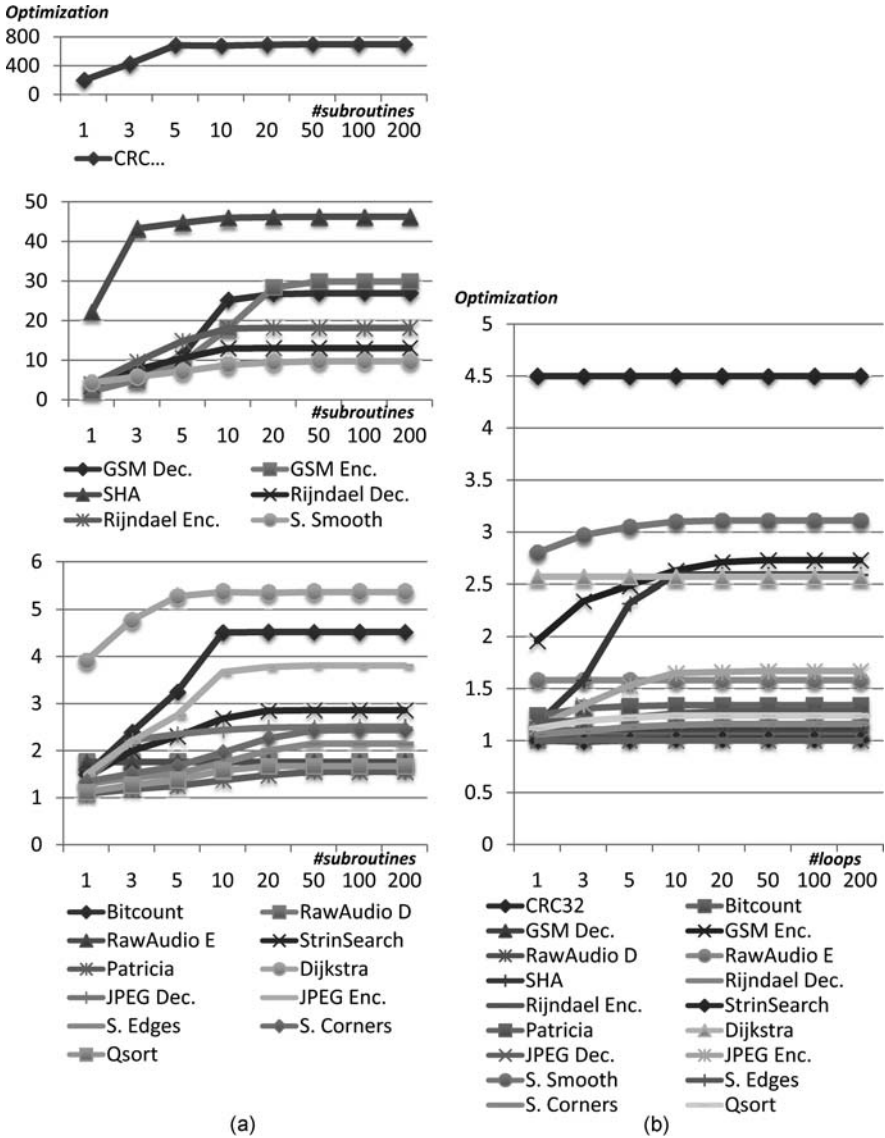


Fig. 2.18 Performance gains, but now considering 5 cycles per hot spot execution. (a) Subroutines and (b) loops

with the previous one, it can be observed that, although the algorithms that present performance speedups are the same, the speedup levels varies a lot.

Figure 2.19 presents the same analysis, but considering more pessimistic assumptions. Now, each hot spot would take 20 cycles to be executed. Although usually a reconfigurable unit would not take that long to compute one configuration, there are

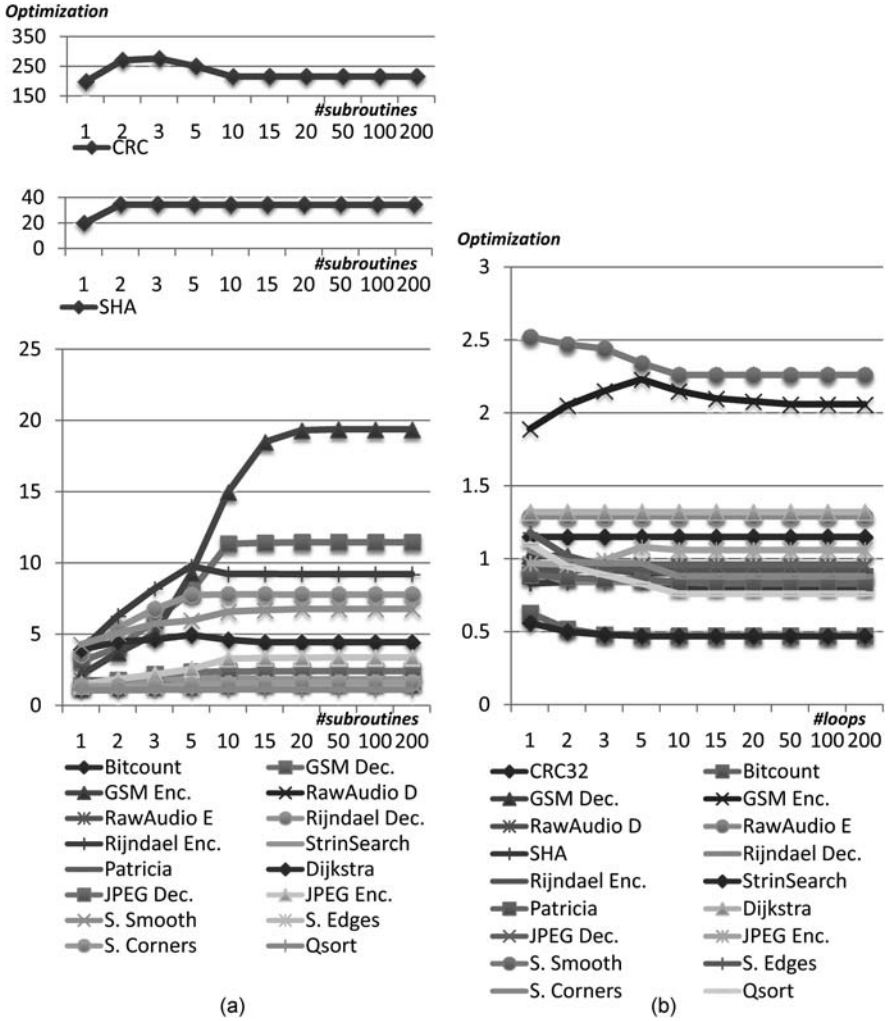


Fig. 2.19 Now considering 20 cycles per hot spot execution. (a) Subroutines and (b) loops

some exceptions, such as configurations with large code blocks, huge context sizes or those that have massive memory accesses. In the same figure, one can observe that some algorithms present even performance loss. 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.

Now, a different methodology is considered: it is assumed that enough reconfigurable hardware is available to support infinite configurations. This way, in opposite to the previous methodology, entire loops or subroutines could be optimized, regardless if all instructions inside them are executed in all iterations. Figure 2.20

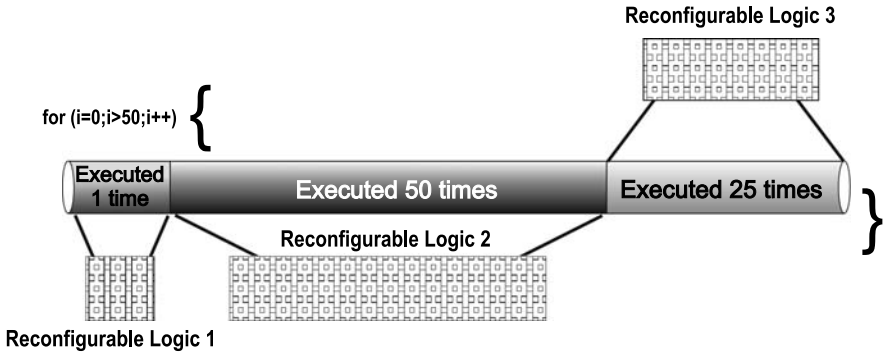


Fig. 2.20 Different pieces of reconfigurable logic are used to speed up the entire loop

illustrates this assumption. A different configuration would be available for each part of the code to be executed on reconfigurable hardware.

In the experiment presented in Figs. 2.21a and b, it is considered that the execution of each configuration would take 5 cycles. Comparing against Fig. 2.18 (same experiment, but using the previous methodology), huge improvements are shown, mainly when considering subroutine optimizations. This, in fact, reinforces the importance of using 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.

2.6.3 Coarse Grain Reconfigurable Architectures

In this section, the potential of performance improvements considering that a coarse grain array is employed is analyzed. In these experiments, let us consider that the optimization will take place at the instruction level, and no speculative execution is supported. Therefore, the optimization is limited to basic block boundaries. Consequently, the level of optimization is directly proportional to the BBs execution rate (Fig. 2.14). For a determined basic block, the more it is executed, more performance boosts it represents for the overall acceleration.

Assuming that one configuration takes just one cycle to be executed, let us compare the instruction level optimization (representing the coarse grain architecture) against the subroutine level (representing the fine grain), which had previously shown more performance improvements than the loop level. When comparing the results in Fig. 2.22a to the ones in Fig. 2.17, one can observe that, for some algorithms, no matter how many basic blocks are optimized, the level of optimization will not reach the ones presented when executing only one subroutine in a fine grain

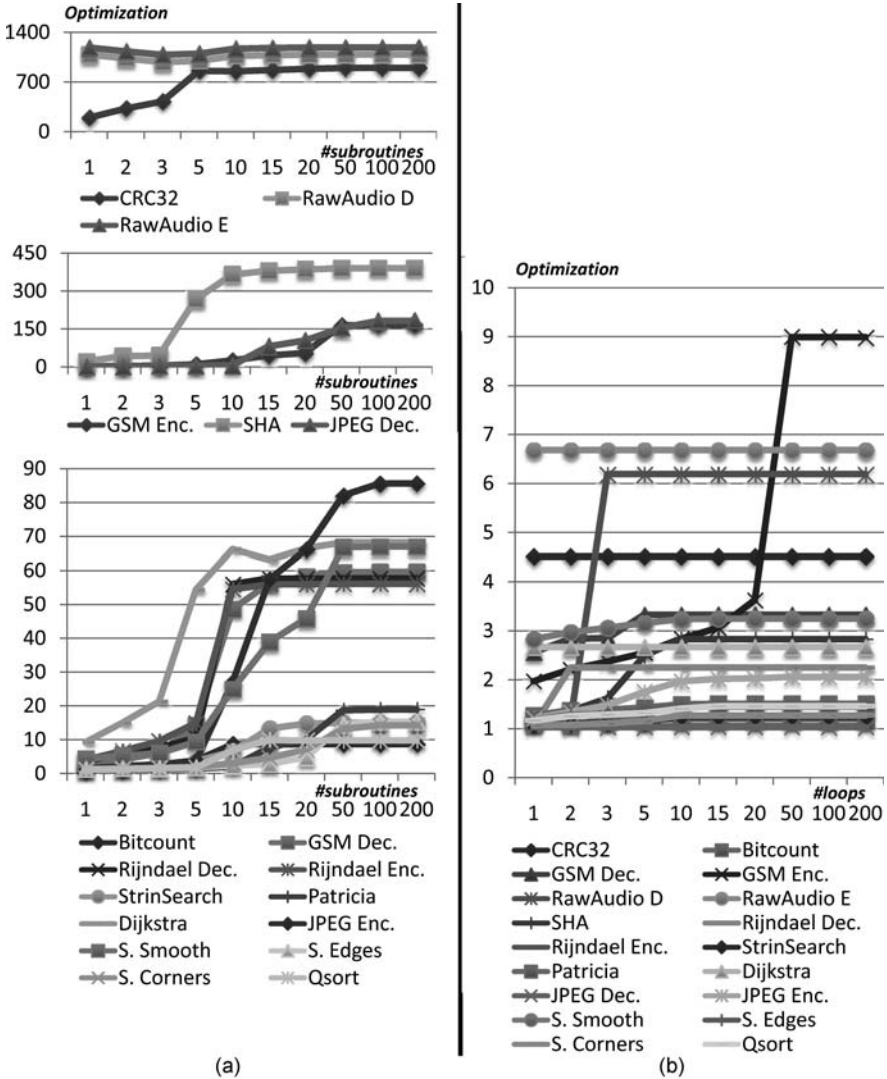


Fig. 2.21 Infinite configurations available for (a) subroutine optimization: each one would take 5 cycles to be executed. (b) The same, considering loops

system. However, for others, mainly the most complex ones, the level of optimization is almost the same for basic blocks or subroutines (they can be observed at the bottom of the figure). In the later case, using the instruction level optimization would be the best choice: it is easier and cheaper to implement 10 different configurations in coarse grain logic than 10 in a fine grain system, since much less connections and reconfigurable gates will be involved, with much less control circuits and associated delays.

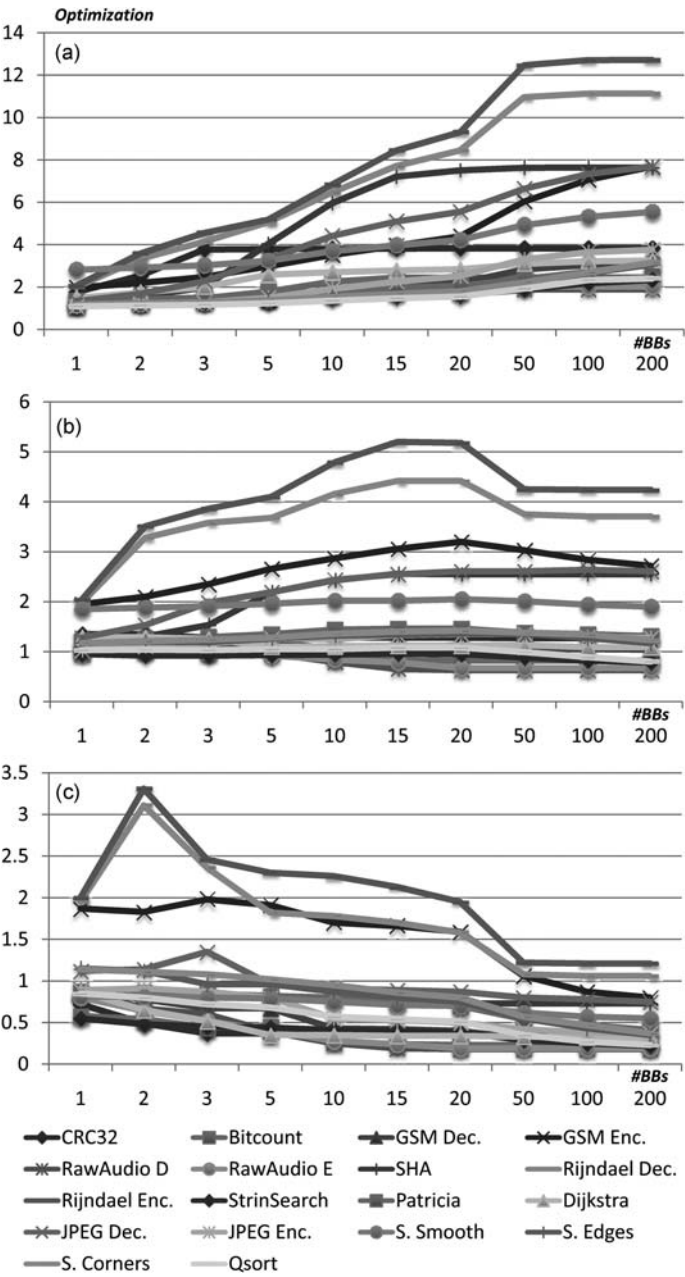


Fig. 2.22 Optimization at instruction-level with the basic block as limit. (a) 1 cycle, (b) 5 cycles, (c) 20 cycles per BB execution

When assuming that each configuration would take 5 cycles to be executed, 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 Fig. 2.22b: in the *Rinjdael* algorithms, the use of a coarse grain system is worth until a certain number of basic blocks being implemented on reconfigurable logic is reached. After that, there is performance loss. Considering 20 cycles per basic block execution (Fig. 2.22c), this situation becomes more evident. This shows that, as for fine grain reconfigurable architectures, there is a necessity of small reconfiguration and context loading times. These are easier to be achieved in coarse grain architectures though, since the size of each configuration is usually much smaller than fine grain ones.

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 would be available to be executed on this kind of system, considering that the memory size for keeping contexts would be smaller. In this case, the chances of optimization for applications that do not have very distinct kernels would increase when comparing against fine grain systems.

2.6.4 Comparing Both Granularities

Considering fixed applications, or data stream based ones, or yet those with long lifetime periods such as an MP3 player, fine grain reconfigurable systems may be a good choice. Some algorithms present huge potential for performance improvements, such as *CRC32*, *SHA* or *Dijkstra*. Only a small number of hot spots has to be optimized in these examples for them to present good acceleration results. This strategy, however, usually requires long development times, since a translation from the software code to a hardware description language amenable to synthesis must take place. Moreover, although there are tools that try to ease this task [33], their efficiency is quite limited, and several design iterations with human intervention are necessary.

Furthermore, it is important to point out a new industry trend: the number of different applications being executed on the systems is increasing and 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. Hence, for each application, different optimizations would be required. This, in consequence, would 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, even if they do not bring as much improvement as the fine grained approaches show.

The authors in [37] advocate in favor of coarse grain architectures. According to the authors, there are some reasons about why one should employ a coarse grain reconfigurable system, as follows:

- *Small configuration contexts:* Coarse grain reconfigurable units need 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 the basic processing elements of coarse grain structures, since 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 reconfigurations, even at run-time.
- *Reduced context memory size:* Being also a consequence of the first statement, the context memory size is also reduced. 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 optimal design of interconnections for the target domain.
- *Silicon area efficiency and reduced routing overhead:* Coarse grained units are optimally-designed hardwired units that are not built by combing a number of CLBs and interconnection wires, resulting in reduced routing overhead and better area utilization.

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

- *Low performance and high power consumption:* This happens mainly because word level modules need to be built by connecting a number of CLBs using a large number of programmable switches.
- *Large context and configuration time:* The configuration of CLBs and interconnections between them are performed at bit-level. This results in a large configuration context that has to be downloaded from the context memory, increasing configuration time, which 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, demanding a large context memory. Because of that, in many times the reconfiguration contexts are stored in external memories increasing even more the time and power necessary 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.

However, still according to the authors in [37], 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. However, as it will be shown in the sequel, there are actually examples of reconfigurable accelerators that show excellent performance under a dynamic environment to optimize different kernels.

References

21. Athanas, P.M., Silverman, H.F.: Processor reconfiguration through instruction-set metamorphosis. *Computer* **26**(3), 11–18 (1993). doi:[10.1109/2.204677](https://doi.org/10.1109/2.204677)
22. Barat, F., Lauwereins, R.: Reconfigurable instruction set processors: A survey. In: RSP'00: Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000), p. 168. IEEE Computer Society, Los Alamitos (2000)
23. Beck, A.C., Rutzig, M.B., Gaydadjiev, G., Carro, L.: Run-time adaptable architectures for heterogeneous behavior embedded systems. In: ARC'08: Proceedings of the 4th International Workshop on Reconfigurable Computing, pp. 111–124. Springer, Berlin/Heidelberg (2008)
24. Clark, N., Kudlur, M., Park, H., Mahlke, S., Flautner, K.: Application-specific processing on a general-purpose core via transparent instruction set customization. In: MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 30–40. IEEE Computer Society, Los Alamitos (2004). doi:[10.1109/MICRO.2004.5](https://doi.org/10.1109/MICRO.2004.5)
25. Compton, K., Hauck, S.: Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.* **34**(2), 171–210 (2002). doi:[10.1145/508352.508353](https://doi.org/10.1145/508352.508353)
26. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: a free, commercially representative embedded benchmark suite. In: WWC'01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop, pp. 3–14. IEEE Computer Society, Los Alamitos (2001). doi:[10.1109/WWC.2001.15](https://doi.org/10.1109/WWC.2001.15)
27. Hennessy, J.L., Patterson, D.A.: *Computer Architecture: A Quantitative Approach*, 4th edn. Morgan Kaufmann, San Mateo (2006)
28. Henning, J.L.: Spec cpu2000: Measuring cpu performance in the new millennium. *Computer* **33**(7), 28–35 (2000). doi:[10.1109/2.869367](https://doi.org/10.1109/2.869367)
29. Hwu, W.M.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Queller, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., Lavery, D.M.: The superblock: an effective technique for vliw and superscalar compilation. In: *Instruction-level Parallel Processors*, pp. 234–253 (1995)
30. Jain, M.K., Balakrishnan, M., Kumar, A.: Asip design methodologies: Survey and issues. In: VLSID'01: Proceedings of the 14th International Conference on VLSI Design (VLSID'01), p. 76. IEEE Computer Society, Los Alamitos (2001)
31. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: MICRO 25: Proceedings of the 25th Annual International Symposium on Microarchitecture, pp. 45–54. IEEE Computer Society, Los Alamitos (1992). doi:[10.1145/144953.144998](https://doi.org/10.1145/144953.144998)
32. Panainte, E.M., Bertels, K., Vassiliadis, S.: The Molen compiler for reconfigurable processors. *ACM Trans. Embed. Comput. Syst.* **6**(1), 6 (2007). doi:[10.1145/1210268.1210274](https://doi.org/10.1145/1210268.1210274)
33. Patel, S.J., Lumetta, S.S.: Replay: A hardware framework for dynamic optimization. *IEEE Trans. Comput.* **50**(6), 590–608 (2001). doi:[10.1109/12.931895](https://doi.org/10.1109/12.931895)
34. Sima, D.: Decisive aspects in the evolution of microprocessors. *Proc. IEEE* **92**(12), 1896–1926 (2004)
35. Singh, H., Lee, M.H., Lu, G., Bagherzadeh, N., Kurdahi, F.J., Filho, E.M.C.: Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.* **49**(5), 465–481 (2000). doi:[10.1109/12.859540](https://doi.org/10.1109/12.859540)

36. Smith, M.J.S.: Application-Specific Integrated Circuits. Addison-Wesley, Reading (2008)
37. Theodoridis, G., Soudris, D., Vassiliadis, S.: A survey of coarse-grain reconfigurable architectures and cad tools. In: Fine- and Coarse-Grain Reconfigurable Computing, pp. 89–149. Springer, Dordrecht (2007). <http://www.springerlink.com/content/j118u3m6m225q264/>

Dynamic Reconfigurable Architectures and Transparent
Optimization Techniques

Automatic Acceleration of Software Execution

Beck Fl., A.C.S.; Carro, L.

2010, XVII, 177 p., Hardcover

ISBN: 978-90-481-3912-5