Run-time Adaptable Architectures for Heterogeneous Behavior Embedded Systems

Antonio Carlos S. Beck[1], Mateus B. Rutzig[1], Georgi Gaydadjiev[2], Luigi Carro[1],

[1] Universidade Federal do Rio Grande do Sul – Porto Alegre/Brazil
[2] Delft University of Technology, Computer Engineering – Delft/The Netherlands
{caco, mbrutzig}@inf.ufrgs.br, g.n.gaydadjiev@ewi.tudelft.nl, carro@inf.ufrgs.br

Abstract. As embedded applications are getting more complex, they are also demanding highly diverse computational capabilities. The majority of all previously proposed reconfigurable architectures targets static data stream oriented applications, optimizing very specific computational kernels, corresponding to the typical embedded systems characteristics in the past. Modern embedded devices, however, impose totally new requirements. They are expected to support a wide variety of programs on a single platform. Besides getting more heterogeneous, these applications have very distinct behaviors. In this paper we explore this trend in more detail. First, we present a study about the behavioral difference of embedded applications based on the Mibench benchmark suite. Thereafter, we analyze the potential optimizations and constraints for two different run-time dynamic reconfigurable architectures with distinct programmability strategies: a fine-grain FPGA based accelerator and a coarse-grain array composed by ordinary functional units. Finally, we demonstrate that reconfigurable systems that are focused to single data stream behavior may not suffice anymore.

1 Introduction

While the number of embedded systems continues to grow, new and different devices, like cellular phones, mp3 players and digital cameras keep appearing on the market. Moreover, a new trend can be observed: the multi-functional embedded system, which performs a wide range of different applications with diverse behaviors, e.g. present day portable phones or PDAs. As a consequence, simple general purpose or DSP processors cannot handle the additional computational power required by these devices anymore. Although a large number of techniques that can solve the performance problem are available, they mainly exploit the instruction level parallelism (ILP) intrinsic to the application, e.g. the superscalar architectures. However, these architectures spend a considerable amount of power while finding the ILP [1]. For that reason, alternative approaches, such as reconfigurable fabrics, have been gaining importance in the embedded domain, speeding up critical parts of data stream oriented programs. By translating a sequence of operations into a hardware circuit performing the same computation, one could speed up the system and reduce energy consumption significantly [2]. This is done at the price of additional silicon area, exactly the resource mostly available in new technology generations.

Recent FPGA based reconfigurable devices targeting embedded systems 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 distinct kernels for optimization. This way, speeding up small parts of the software allows one to obtain huge gains. However, as commented before, the number of applications a single embedded device must handle is increasing. Nowadays, it is very common to find embedded systems with ten or more functions with radically different behaviors. Hence, for each of these applications, different optimizations are required. This, in consequence, increases the design cycle, since mapping code to reconfigurable logic usually involves some transformation, which is done manually or by the use of special languages or tool chains. To make this situation even more complicated, some of these 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. As it will be demonstrated, the JPEG decoder is one such example.

In accordance to the facts commented above, the main contributions of this paper are:

- To evaluate the potential performance of two representative run-time dynamically reconfigurable architectures with different programmability strategies for the same set of embedded systems benchmarks;
- To provide indication for the need of mixed behavior architectures supporting different reconfiguration mechanisms based on the above results.

This work is organized as follows: Section 2 discusses classic reconfigurable architectures and shows how they are applied in embedded systems. In Section 3 we discuss two representative architectures with different configuration strategies used in our study. Section 4 summarizes the analysis done on the benchmark set with these two systems. Finally, Section 5 draws conclusions and indicates some directions.

2 Related Work

Reconfigurable systems have already shown to be very effective in mapping certain pieces of the software to reconfigurable logic. Huge software speedups as well as significant system energy reductions have been previously reported [2]. Careful classification study in respect to coupling, granularity and instructions type is presented in [3]. In accordance with this study, in this section we discuss only the most relevant work. For instance, processors like Chimaera [4] have a tightly coupled reconfigurable array in the processor core, limited to combinational logic only. The array is, in fact, an additional functional unit (FU) in the processor pipeline, sharing the resources with all normal FUs. This simplifies the control logic and diminishes the communication overhead between the reconfigurable array and the rest of the system. The GARP machine [5] is a MIPS compatible processor with a loosely coupled reconfigurable array. The communication is done using dedicated move instructions.

More recently, new reconfigurable architectures, very similar to the dataflow approaches, were proposed. For example, TRIPS is based on a hybrid von-Neumann/dataflow architecture that combines an instance of coarse-grained, polymorphous grid processor core with an adaptive on-chip memory system [6]. TRIPS uses three different execution modes, focusing on instruction-, data- or thread-

level parallelism. Wavescalar [7], on the other hand, totally abandons the program counter and the linear von-Neumann execution model that is limiting the amount of exploited parallelism. As these two examples, Piperench is also a reconfigurable machine highly relying on compiler driven resource allocation [8].

Specifically concerning embedded systems, Stitt et al. [9] presented the first studies about the benefits and feasibility of dynamic partitioning using reconfigurable logic, showing good results for a number of popular embedded system benchmarks. This approach, called *warp processing*, is based on a complex SoC. It is composed by a microprocessor to execute the application software, another microprocessor where a simplified CAD algorithm runs, local memory and a dedicated FPGA array. Firstly, the microprocessor executes the code, and a profiler monitors the instructions in order to detect critical regions. Next, the CAD software decompiles the application to a control flow graph, synthesizes it and maps the circuit onto the FPGA structure. At last the original binary code is modified to use the generated hardware blocks. In [10] another reconfigurable architecture is presented, called Computer Configurable Accelerator (CCA), working in granularity of the instruction level. The CCA is composed by very simple functional units, tightly coupled to an ARM processor.

We can observe two main trends concerning reconfigurable logic organization for embedded systems. The first one is aimed at optimizing the most executed kernels, with a fine grain array usually being relatively loosely coupled to the processor, or avoiding a central processor at all [6][7][8][9]. The second class are tightly coupled coarse grain units, embedded as ordinary FUs in the processor pipeline [4][5][10], attacking lowest levels of parallelism, such as ILP. That is why in this work we are considering two representative examples from these two distinctive groups for our evaluation.

3 Evaluated Architectures

In this study we are employing two different run-time dynamically reconfigurable systems to perform our analysis. Although they represent different programming paradigms, both are implemented using the same general-purpose processor (GPP): a MIPS R3000. The first one is Molen, an FPGA reconfigurable architecture [11]. Because of the high penalties incurred by reconfiguration, it is usually used to speed up the applications at loop and subroutine levels. Since it is based on fine grain devices and variable size of instruction blocks, code can be transformed into reconfigurable logic with huge potential for optimizations. However, one must take into account that there is a design effort to create the optimized hardware blocks for each kernel. Implementing huge parts of the code in a reconfigurable logic can sometimes become very complex, putting pressure on the development time, which, in turn, cannot be always tolerated. The design process can be seen as designing a VHDL engine for a specific subroutine inside the targeted program.

The second reconfigurable system is tightly coupled to the processor and has a coarse-grain nature [12], acting at the instruction level. This way, the allocation of reconfigurable resources becomes easier, since the implementation of execution blocks is usually based on a simple allocation of instructions to an array of ordinary functional units. However, as it does not work at bit level and optimizes smaller parts of the code, the degree of potential performance gains of such system is not the same

as for fine grain approaches. The designer effort, however, can be significantly decreased (or avoided), and a vast number of different reconfigurations could be available, which makes this approach more suitable for systems with a large number of hot spots. In the two sub-sections to follow both systems are explained in more details.

3.1 Fine-Grain based reconfigurable system

The two main components in the Molen organization are depicted in Figure 1a. More precisely, they are the *Core Processor*, which is a GPP (in this case study MIPS R3000), 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.

3.2 Coarse-grain based reconfigurable system

This reconfigurable array operates as an additional functional unit in the execution stage, using an approach similar to Chimaera's in this aspect. This way, no external accesses (from the processor perspective) to the array are necessary. The array is two-dimensional, organized in rows and columns, and composed by ordinary functional units, e.g ALUs, shifters, multipliers, etc.

An overview of the general organization of the DIM system is shown in Figure 1b. 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 a particular type of functional units. 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 can vary depending on the technology and the way the functional unit was implemented. Moreover, regular multiplexers are responsible for the routing (Figure 1c and Figure 1d). Presently, the reconfigurable array does not support floating point operations.

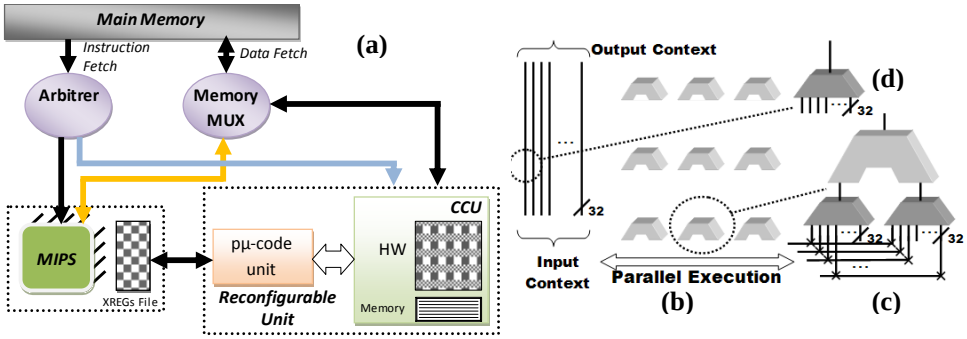

Fig. 1. The (a) FPGA-based and the (b) coarse-grain array reconfigurable architectures

The array works together with a special hardware designed to detect and transform instruction groups for reconfigurable hardware execution. This is done concurrently while the main processor fetches other instructions. When this unit detects a minimum number of instructions worth being executed in the array, a binary translation is applied to this code sequence. After that, this configuration is saved in a special cache, and indexed by the program counter (PC). The next time this sequence is found, the dependence analysis is no longer required: the processor just loads the configuration from the special cache and the operands from the register bank, then activating the reconfigurable hardware as functional unit. Thereafter, the array executes the configuration with that context and writes back the results, instead of executing the normal instruction flow of the processor. Finally, the PC is updated in order to continue with the execution of the normal (not translated) instructions.

4 Experimental Results

4.1 Benchmarks Evaluation

In our study we use the Mibench Benchmark Suite [13]. This suite has been chosen because it has a large range of different application behaviors. We are using all benchmarks with no representative floating point computations and that could be compiled successfully to the target architecture.

First, we characterize the algorithms regarding the number of instructions executed per branch (classifying them as control or dataflow oriented based on these numbers). As it can be observed in Figure 2b, 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 (BB) has, the better, since there is more room for exploiting parallelism (a basic block is code that has one entry point, one exit point and no jump instructions contained within it). 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. Our experiments presented in Figure 2b show similar trends as the ones reported in [13], where the same benchmark set was compiled on a different processor (ARM).

Figure 2a shows our analysis of distinct kernels based on the basic blocks present in the programs and their execution rates. The methodology involves investigating the number of basic blocks responsible for a certain percentage of the total number of basic block execution figures. 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 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.

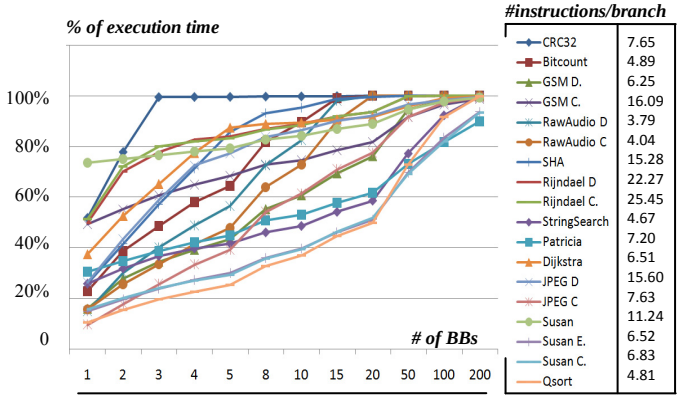

Fig. 2. (a) Instruction per Branch rate. (b) How many BBs are necessary to cover a certain amount of execution time

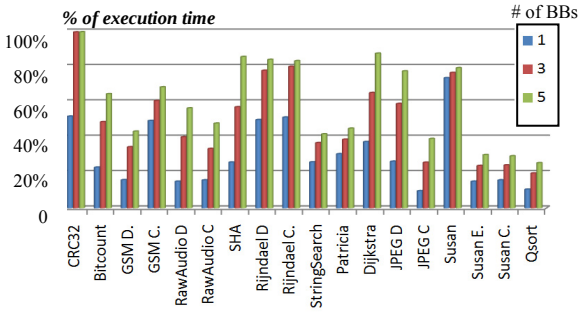

Fig. 3. Amount of execution time covered by 1, 3 or 5 BBS in each application

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 2x 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 3 illustrates this trend by using the cases with 1, 3 and 5 basic blocks. Note that, mainly when we consider the most executed basic block only (first bar of each benchmark), the shape of the graph is directly proportional to the instructions per branch ratios shown in Figure 2b (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.

4.2 Potential of Optimization

In this section, we first study the potentiality of fine grain reconfigurable arrays. Considering the optimization of loops and subroutines, we analyze the level of

performance gains if a determined number of such hot spots is mapped to the Molen based reconfigurable logic.

Fine Grain reconfigurable architecture. In this first experiment, we are assuming 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 itterations. For example, let us assume a loop that 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 4a 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.

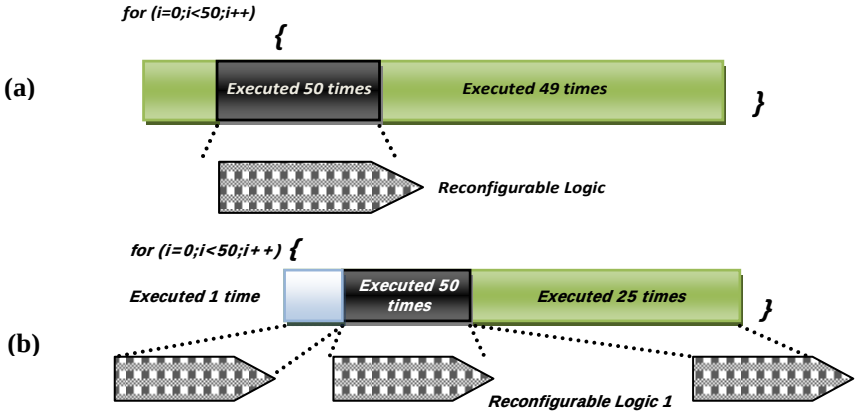

Fig. 4. (a) Just a small part of the loop can be optimized (b) Different pieces of reconfigurable logic are used to speed up the entire loop

Figures 5a and 5b 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. 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 FPGA 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 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. As a consequence, human interaction for changing and adapting parts of the code would be required.

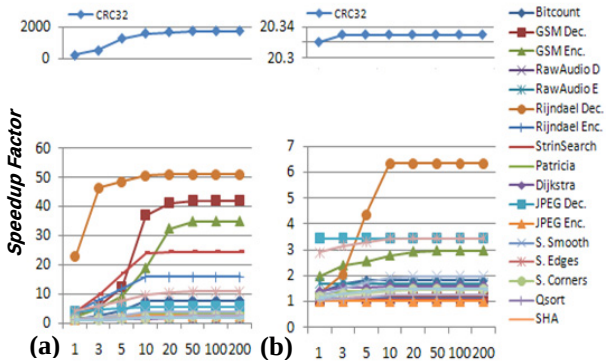

Fig. 5. Speedup factor considering different numbers of subroutines and loops being executed in 1 cycle

In the first experiment, besides considering infinite hardware resources and no communication overhead between the processor and reconfigurable logic, we were also assuming an infinite number of memory ports with zero delay, which is practically infeasible for any relatively complex configuration. Now, in Figures 6a and 6b, we consider a more realistic assumption: 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.

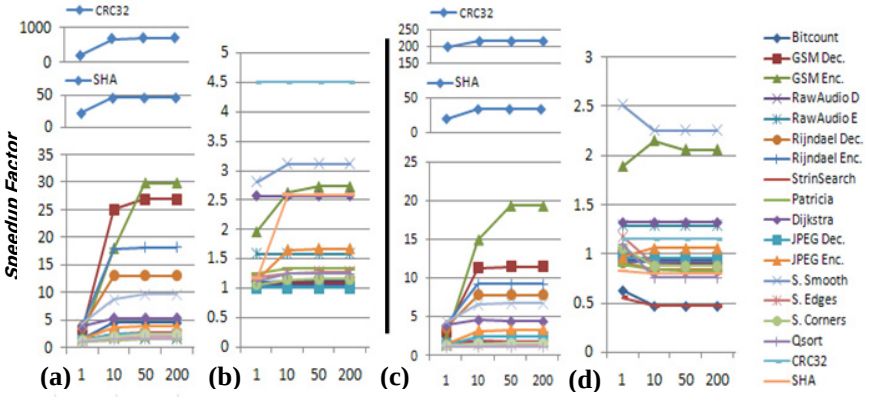

Fig. 6. Speedup factor considering different numbers of subroutines and loops being executed in (c) (d) 5 cycles and (e) (f) 20 cycles in reconfigurable logic

In Figure 6c and 6d we present 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 really large code blocks or those that have massive memory accesses. Also, 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.

In Figure 7 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 4b illustrates this assumption. A reconfigurable unit would be available for each part of the code.

In this experiment it is considered that the execution of each configuration would take 5 cycles. Comparing against Figures 6a and 6b (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.

Coarse Grain reconfigurable architecture. Now, we analyze the performance improvements when considering such architecture. Since it works at the instruction level, and in this version 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 2a): for a determined basic block, the more it is executed, more performance boost it represents. Even this coarse grain reconfigurable array does not demonstrate the same level of performance gains as fine grain reconfigurable systems show, more different configurations are available to be executed on this kind of system. This way, even applications which do not have very distinct kernels could be optimized.

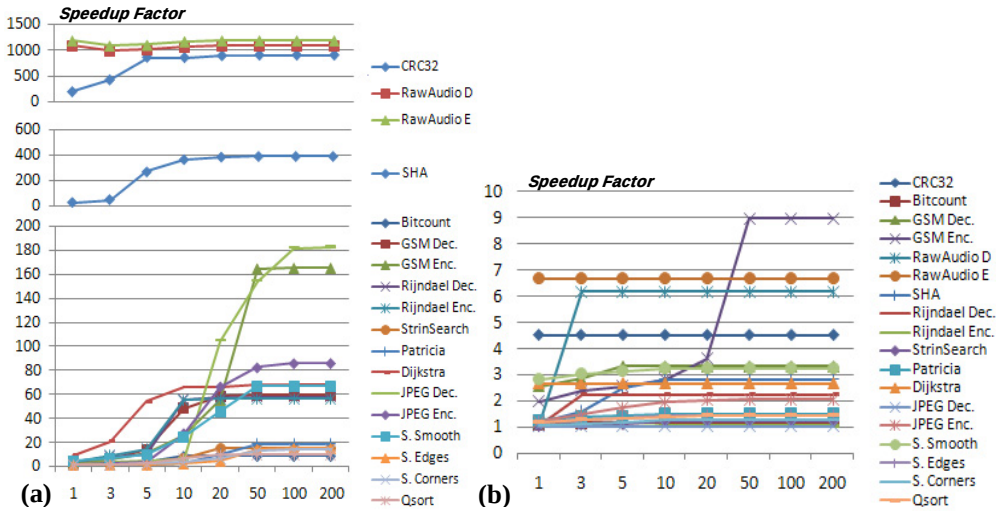

Fig. 7. Infinite configurations available for (a) subroutine or (b) loop optimizations: each one would take 5 cycles to be executed.

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 8a against the ones of Figure 5a, 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 that 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 are the basic blocks (in number of instructions, kind of operations, memory accesses etc). This assumption is demonstrated in Figure 8b: 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 8c, considering 20 cycles per basic block execution on the reconfigurable array, this situation is even more evident.

Mixed Systems. Hybrid systems are envisioned as a mix of the two different reconfiguration approaches investigated above. In Figure 9a it is assumed that the most executed subroutine would be implemented into the FPGA reconfigurable logic. Then, considering that the first 3 most executed basic blocks would be contained in this subroutine, the next 17 would be executed on the coarse grain unit. In Figure 9b the same approach is used, but now considering 2 subroutines implemented on FPGA and 45 basic blocks on the coarse grain (assuming that the first 5 basic blocks are part of the two subroutines already implemented on FPGA).

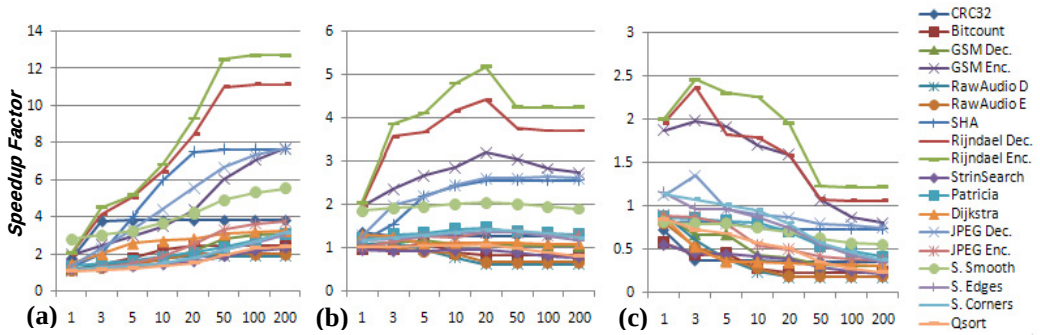

Fig. 8. Optimization at instruction-level with the basic block as limit. (a) 1 cycle, (b) 5 cycles and (c) 20 cycles per BB execution

As it can be observed, even though some algorithms do not present a significant performance gain while having this mixed architecture, the majority of the benchmark applications can take advantage of this system. This group would achieve almost the same performance boost when just using one or two pieces of FPGA-based reconfigurable logic together with the coarse grain unit, when comparing against a large number of huge blocks just implemented in FPGA.

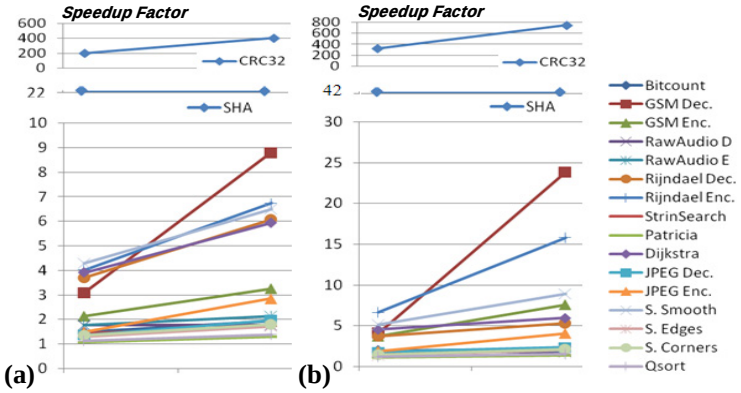

Fig. 9. Simulating a mixed system. (a) One subroutine optimized by FPGA and 17 basic blocks by coarse grain unit. (b) two subroutines - FPGA / 45 BBs – coarse grain

ILP, context loading and saving. Still assuming mixed systems, we repeat the previous examples but now considering the ILP available in each hot spot when computing their execution time. For instance, subroutines with more exposed ILP will execute faster than those with almost no parallelism. The algorithms chosen for this experiment are: *Rawaudio Decoder*, *Rijndael Encoder*, *CRC32* and *Quicksort*, since they are the most control and dataflow ones, and have a small or great number of distinct kernels, respectively.

The following assumptions have been done:

- Three simple arithmetic/logic execution per cycle, one multiplication, infinite memory ports, when executing on the coarse grain reconfigurable architecture;
- Nine simple arithmetic/logic execution per cycle, three multiplications, infinite memory ports when executing on the fine grain logic.

We also vary the reconfiguration (which comprises the configuration of the unit itself and the context loading, as register and memory reads) and write back times, considering them as being responsible for more 10%, 30% and 50% of the total execution time spent by each hot spot when using the coarse grain unit; and 3 times more when executing on fine grain logic. This way, if we are considering that FPGA is 3 times faster when executing its instructions, we are also assuming that it is 3 times slower for the reconfiguration process.

As it can be observed in Figures 10a and 10b, the importance of each architecture varies in each algorithm. In *CRC32*, there is no need of having a mixed architecture to speed up the system, since there is a small number of distinct kernels to be optimized. *Quicksort*, on the other hand, is quite different. Its gains are all presented by the coarse grain array, exactly because of its behavior is exactly the opposite of *CRC32*. Both *Rawaudio* and *Rijndael* benefits from the mixed system, but at different levels.

It is important to point out that the reconfiguration time starts to play an important role depending on the time it takes. For instance, in the *CRC32* algorithm, more time can be spent in the reconfiguration/write back than the in the execution itself. This way, for a given hot spot, one needs to pay attention on the size of the context and the resources available for its loading.

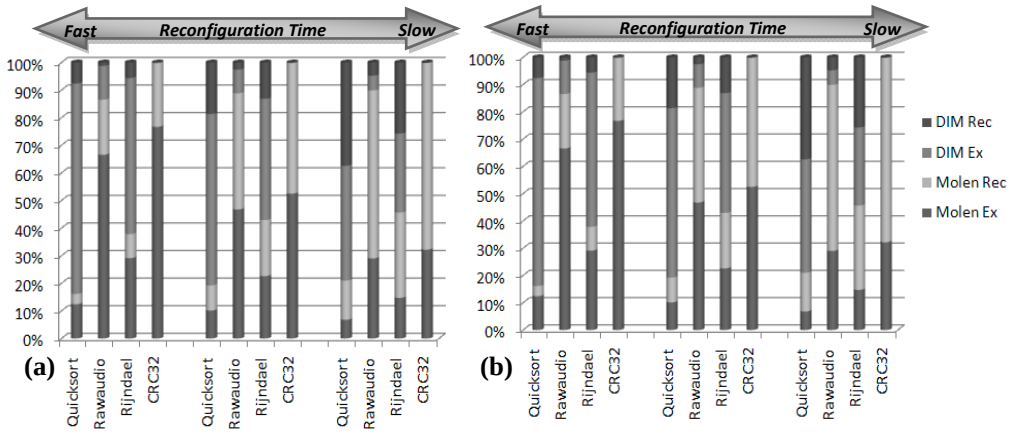

Fig. 10. Mixed system – Two subroutines executed on FPGA and 45 basic blocks on the coarse grain unit, considering different reconfiguration/write back times

5 Conclusions and directions

Reconfigurable systems have already proven to be an efficient strategy to serve as platform for modern embedded systems. However, the number of different applications being executed on these systems has been increasing. In addition, the characteristics of the embedded system workloads have been changing as well. In this work we demonstrated that reconfigurable systems focusing just on the most executed kernels still can be used for some algorithms. This strategy, however, may require long development times that may not be acceptable. On the other hand, an alternative option is the employment of simpler reconfigurable architectures. The latter do not bring as much improvement as the fine grained approaches show, but could be easier to implemented due to its simplicity.

Considering fixed embedded applications, or yet those with long lifetime periods such as an MP3 player, FPGA based reconfigurable systems with high granularity grains can be a good choice. According to the results section, one can observe algorithms that present huge performance improvements, such as *CRC32*, *SHA* or *Dijkstra*. They need a small number of hot spots to achieve such gains, which would lead to a short development time.

On the other hand, there are other embedded devices, such as PDAs, with almost no pre installed applications. As they give to the user the freedom for installing applications, PDAs become a very heterogeneous environment. They will probably demand more flexible reconfigurable devices. Considering the benchmark set, some examples can be cited, as *Rijndael* and *Susan Corners*, which can just show good improvements when a large number of hot spots are optimized. As it was commented before, one also needs to take into consideration the assumption of having a great number of these applications being executed on the same device at the same time.

Finally, one could consider the example of next generation of mobile phones. Besides having fixed applications such as audio, image and video decompression, various Java applets can be installed. This way, mixed systems behavior appears as being a good choice. Moreover, we could envision some general directions when

developing reconfigurable systems targeting to embedded applications, based on the discussion in section 3:

- Automated tools aimed at searching hot spots to be executed on reconfigurable logic may need human interaction;
- Algorithms with a great number of instructions/branch tend to present a small number of distinct kernels for optimization;
- For both level of granularities, some hot spots, even if they are highly used through program execution, may not be worth to be implemented in reconfigurable logic, due to low amount of ILP, great number of memory accesses for a given configuration, high overhead for context loading/saving;
- Partially reconfigurable systems may be a good choice in the sense that they can adapt themselves to parts of loops or methods that change while keeping other kernels always in the reconfigurable logic;
- The use of mixed systems, composed by both kinds of architectures, can be a good alternative. However, the reconfiguration/write back times must be taken into account when choosing the kernels to be optimized.

As future work, we will analyze both architectures in more details, as their area overhead and power consumption.

References

1. 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 in conjunction with Micro-33, 1999.
2. Stitt, G., Vahid F., “The Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic”. In IEEE Design and Test of Computers, 2002.
3. Compton K., Hauck, S., “Reconfigurable computing: A survey of systems and software”. In ACM Computing Surveys, vol. 34, no. 2, pp. 171-210, June 2002.
4. Hauck, S., Fry, T., Hosler, M., Kao, J., “The Chimaera reconfigurable functional unit”. In Proc. IEEE Symp. FPGAs for Custom Computing Machines, Napa Valley, CA, pp. 87–96, 1997.
5 Hauser, J. R., Wawrzynek, J., “Garp: a MIPS processor with a reconfigurable coprocessor”. In Proc. 1997 IEEE Symp. Field Programmable Custom Computing Machines, pp. 12–21, 1997.
6. Sankaralingam, k., Nagarajan, R., Liu, H. , Kim, C. , Huh, J. , Burger, D. , Keckler, S. W., Moore C. R., “Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture”. In Proc. of the 30th Int. Symp. on Computer Architecture, pp. 422-433, June 2003.
7. Swanson, S., Michelson, K., Schwerin, A., Oskin. M.: WaveScalar. MICRO-36, Dec. 2003.
8. Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., Taylor, R.R., "PipeRench: A Reconfigurable Architecture and Compiler". In IEEE Computer, pp. 70-77, April, 2000.
9. Lysecky, R., Stitt, G., Vahid, F., "Warp Processors". In ACM Transactions on Design Automation of Electronic Systems (TODAES), pp. 659-681, July 2006.
10. Clark, N., Kudlur, M., Park, H. Mahlke, S., Flautner, K. "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization". In International Symposium on Microarchitecture (MICRO-37), pp. 30-40, Dec. 2004.
11. Beck, A. C. S., Rutzig M. B., Gaydadjiev, G. N., Carro, L. “Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications”. In Design, Automation and Test in Europe (DATE), Munique, March 2008.
12. Vassiliadis, S., Wong, S., Cotofana. S. “The MOLEN ρμ-coded processor”. In Lecture Notes in Computer Science, 2147:275–285, 2001.
13. 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 4th Workshop on Workload Characterization, Austin, TX, Dec. 2001.