Hardware Virtualization On Coarse-Grained Reconfigurable Architectures

Thiago Berticelli Lo
Departamento de Ensino, Pesquisa e Extensão
Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense, IFSUL
Charqueadas/RS, Brazil
thiagolo@charqueadas.ifsul.edu.br

Luigi Carro, Antonio Carlos Schneider Beck
Instituto de Informática – PPGC
Universidade Federal do Rio Grande do Sul, UFRGS
Porto Alegre/RS, Brazil
{carro, caco}@inf.ufrgs.br

***Abstract*—Even though reconfigurable systems have already proven to be an alternative to speedup embedded applications with reduced energy costs, because of their adaptability and high flexibility, their use result in a significant overhead in chip area. Therefore, this work addresses this issue by the use of the hardware virtualization technique, using a coarse-grained reconfigurable array as study case. We explore two different virtualization alternatives, achieving a reduction in area of up 96%, with marginal performance loss comparing to the original architecture.**

Keywords—hardware virtualization; adaptable architectures, reconfigurable systems;

I. INTRODUCTION

The advance of integrated circuits technology over the years has allowed greater transistor integration. Smartphones are an example: one can find several features in a single electronic device. These devices are still classified as embedded systems, even though they have to execute several heterogeneous applications that present a mix of control and dataflow behaviors. Therefore, embedded systems are increasingly complex and need high performance computing. However, they are also tied to a number of design restrictions such as area occupation, energy consumption, memory footprint and time-to-market constraints.

An approach that emerged as an alternative to the aforementioned restrictions is reconfigurable computing. They provide flexibility of implementation, thanks to its ability to adapt to the behavior of the target applications after manufacturing. They have already proven to be able to satisfy the constraints imposed by embedded systems [1][2][3][4][5]. They may be classified into two categories: coarse- and fine-grained, according to the level of reconfiguration (word and bit, respectively).

However, in order to achieve significant performance gains, large quantities of redundant functional units are usually necessary (in the case of coarse-grained systems), which also impacts the interconnections system (i.e.: increase the amount of multiplexors). Consequently, there is a considerable increase in the area occupied [6].

In this scenario, hardware virtualization can be an interesting technique for embedded systems, particularly for systems that use reconfigurable architectures where applications should be mapped and executed on hardware with limited area [7]. This is the main proposal of this work: the use of virtualization to reduce the area of a reconfigurable system, analyzing the impact on the performance and area. As a case study, we use the coarse-grain architecture tightly coupled to the MIPS R3000 processor [8].

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3presents the architecture of the reconfigurable data path. Section 4 describes the hardware virtualization implementation, while the results are presented in Section 5. Finally, Section 6 draws the final considerations of this paper.

II. RELATED WORK

In [9], using the same case study of this work, the authors presented a tool called ARISE, which receives a set of applications and gives a result the number of functional units and the right structure of the reconfigurable array considering the available ILP of the benchmark set. This optimized architecture has achieved a reduction of 41%, on average, compared to the original hardware. The disadvantage of this approach is that when the applications set changes, the reconfigurable array must be modified, which is not always possible, since the system may have already been deployed.

In another study [10], the authors address the problem of area reduction by optimizing the interconnections. In the case study of this particular work, the interconnections are responsible for approximately 50% of the total area of the chip. The work suggests the use of Omega Networks [11] instead of multiplexers. They present a reduction of 33% in the area of the interconnection, which corresponds to 17% of the total area.

Another approach to achieve area reduction is through hardware virtualization (which can be applied not only to reconfigurable systems). In[7], the author classified hardware virtualization into three categories: Temporal partitioning, virtualized execution, and virtual machine. The one used in this work is classified as virtualized execution.

In the case of reconfigurable systems, virtualized execution can be used to implement scalable and forward compatible

systems. A scheduler must be implemented in the runtime system to map the application either spatially or temporally, depending on the number of operators that are available in a particular implementation of the reconfigurable architecture. A major advantage of this approach is the possibility of the design space exploration, allowing the designer to explore the relationship between are cost and performance, maximizing the gains for each particular project.

Examples for virtualized execution architectures are Score [12], Zippy[13], WASMI [14] and PipeRench [15]. Among the architectures mentioned, PipeRench is the most similar to what is proposed in this paper. The basic principle of PipeRenchis the so-called “pipelined reconfiguration” [15]. It means that a given kernel is broken into pieces, and these pieces can be reconfigured and executed on demand.This way, the parts of a given kernel are multiplexed in time and space into the reconfigurable logic.

The device is organized into a physical pipeline of stripes, which representthe minimal reconfigurable hardware blocks. Each stripe has aninterconnection network and a set of Processing Elements (PEs). A stripe’s output is strictlypipelined and connects to the next stripe via an interconnection network (see Fig. 1). Hardware virtualization is achieved by allowing an application to use an unlimited number of virtual stripes [16].

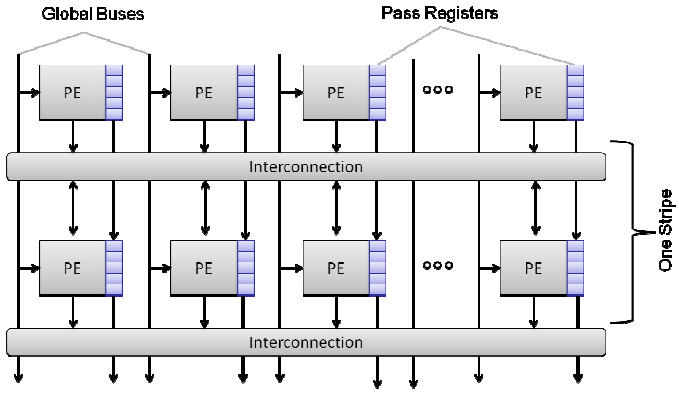

Fig. 1. Piperench architecture overview [16].

In opposite to the Piperench, the case study of this work maintains full binary compatibility, since it does not change the binary code prior to its execution, thanks to a binary translation mechanism that is used together with the reconfigurable logic, as it will be explained next. Therefore, to the best of the authors’knowledge, there is no reconfigurable system that is completely dynamic and transparent that uses virtualization.

III. Case Study

Figure 2 shows the structure of the reconfigurable architecture employed as case study for this work. The reconfigurable unit is organized as a two-dimensional array of functional units, interconnected by multiplexers. As can be observed in Fig. 2b,the functional units are divided into groups(e.g. ALU, Load/Store, Multiplier). Depending on the delay of the functional units that belong to each group, more than one operation can be executed within one equivalent processor cycle, which corresponds to one level in the reconfigurable array. In this case study, according to the MIPS R3000 critical path, one reconfigurable architecture level corresponds to three rows of basic ALUs in sequence, one Load/Store or one Multiplication [17]. Anexample of a hot spot (Fig. 2a) allocation can be seen in Fig. 2c. One level can be observed in more details in Fig. 1c. The structure of the array is very similar to other coarse grain reconfigurable architectures.

This architecture has a binary translation mechanism (BT) [18], which is implemented in hardware and operates in parallel to the processor. At run time, the BT unit detects sequences of instructions that can be executed in the reconfigurable architecture. This sequence is translated into a configuration through the BT mechanism, and saved in the context memory. These sequences are indexed by the Program Counter (PC) register. This process is very similar to the use of automated static tools, employed to find the best kernels of the application and transform them to reconfigurable instructions. However, it is much simpler compared to such tools, so it can be implemented in hardware and executed at run time. A sequence of instructions (kernel) that is transformed to be executed in the reconfigurable array by the Binary Translation hardware is called a configuration.

During execution, the PC of the current incoming instruction is compared to the ones saved in a table, in order to check if there is a configuration with the same value of the current PC. If a configuration is found, it is fetched from the context memory. Other reconfigurable systems work in a very similar way. However, instead of having the configurations indexed by the PC in a table, special instructions in the code indicate where they are in the context memory.

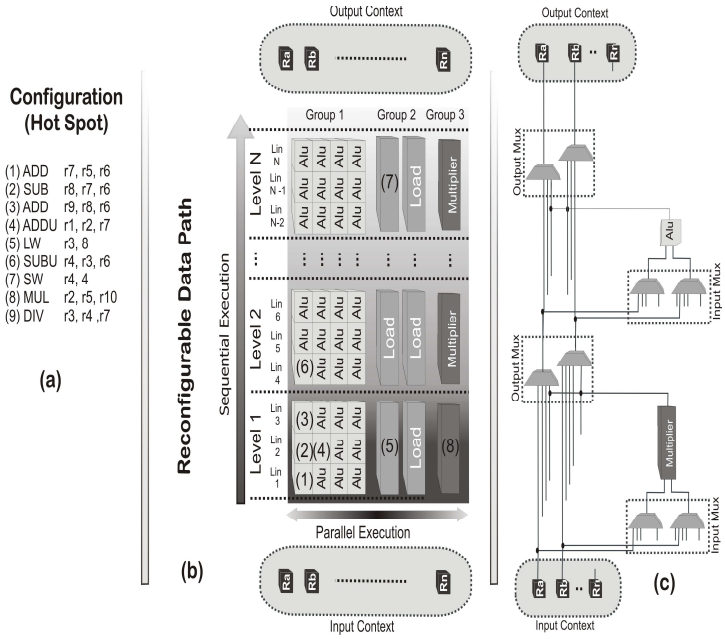

Fig. 2. General overview of the reconfigurable array [17].

The rest of the process works as follows: initially, values of the input context are fetched from the register file while the configuration bits are fetched from the context memory. Then, that configuration is executed, taking a given number of

equivalent processor cycles. Finally, results are written back to the register bank.

The number of context memory entries determines the number of configurations that can be stored in the context memory. The size of each entry depends on the number of functional units that compose the reconfigurable array. These parameters are determined at design time.Each entry in the context memory contains the following data:

- The input context, composed of the operands field, where it can be found references from the register file, immediate values and memory pointers.
- The configuration bits, that indicate which operation each functional unit will perform and the and the routing between them (by configuring the multiplexers).

Moreover, the following additional fields are found:

- The final PC address, in order to update the PC value after the sequence of instructions is executed in the array.
- The number of cycles taken by the operation in the array.

IV. HARDWARE VIRTUALIZATION

In reconfigurable architectures, operations are organized spatially, while in regular processors, they are mainly structured in time (e.g. each instruction is executed in one cycle of the CPU). While processors can run applications as large as the memory capacity, reconfigurable architectures are problems when the application mapping exceeds the size of the reconfigurable array. With the set of hardware virtualization techniques can overcome this limitation by exploiting the reconfigurability of devices [7].

The proposed hardware virtualization is applied by modifying the aforementioned reconfigurable architecture, creating virtual pipeline levels. The main idea is to take advantage of the regularity of the architecture and to virtualize the total number of levels with a smaller number of physical levels (called stages).It is done by inserting registers in the output context lines of each level (refer to Figure 2c). These registers store the outputs of the context lines which will be used for the next level, making it similar to a pipeline. From this modification, two ways of implementing reconfigurable array are modeled. While in the first model only one physical level is used to virtualize all others, the second one uses two physical levels.

A. One Stage Virtualization

This implementation uses only one physical stage, which is equivalent to a reconfigurable array level. The outputs of this level are connected to their own entry points, resulting in a cyclic pipelined stage, as shown in Fig. 3.

Each cycle stage is reconfigured to virtualize the next virtual level that will be executed (making it possible to run as many levels as necessary). Thus, the hardware virtualization provides great savings in area.

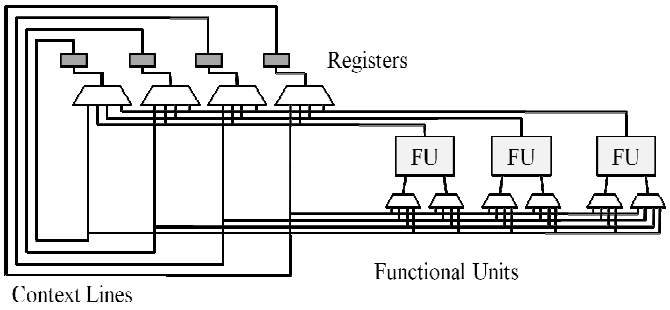

Fig. 3. Hardware virtualization with a physical stage.

The operation proceeds as follows: First the reconfiguration of functional units and interconnections are performed.Thereafter, the execution of the current level is initiated.While executing the current level, the next level is fetched from the context memory in parallel.After execution, the registers of the context lines store the results, which will be the input to the next virtual level that will be processed. All these phases occur in the same cycle.These steps are repeated according to the number of levels.

The control mechanism of the circuit of Fig. 3 is defined by the fetch of the next level configuration, configuration and execution of the current level. The time cycle (1) and the total execution time (2) can be computed by the following equations:

$$TS1 = Tc + Te + Ts \quad (1)$$

$$T1 = N \cdot TS1 \quad (2)$$

where:

TS1 = Execution time of a stage (time cycle)

Tc = Configuration time

Te = Execution time of a level (functional units)

Ts = Setup time of register

Tte1 = Total execution time

N = Number of levels to be executed

As can be noted in (1), the configuration time is part of thestage execution time, which was not included in the original architecture, since in this case the configuration starts only when the configuration was fetched from the context memory. Therefore, even though this approach is the one which presents the smallest chip area possible, it results in performance overhead, which led us to develop the two-stages hardware for the virtualization.

B. Two Stages Virtualization

A 2-stage reconfigurable (Fig. 4) array was implemented to solve the problem of the increasing the total execution time, as discussed above.

The outputs of the stage 1 are connected to the stage 2 inputs; while the stage 2 outputs are connected to the inputs of stage 1, as observed in Figure 4. As in the previous model, the

outputs of context lines also have the stage registers to store the intermediate results (between stages 1 and 2), which results in a 2-stage cyclic pipeline.

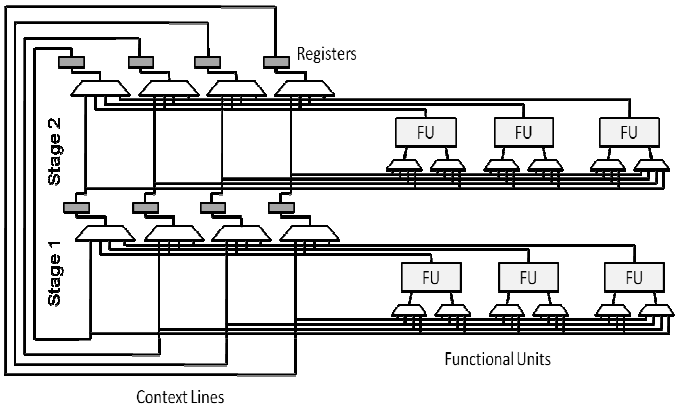

Fig. 4. Hardware virtualization with two physical stage.

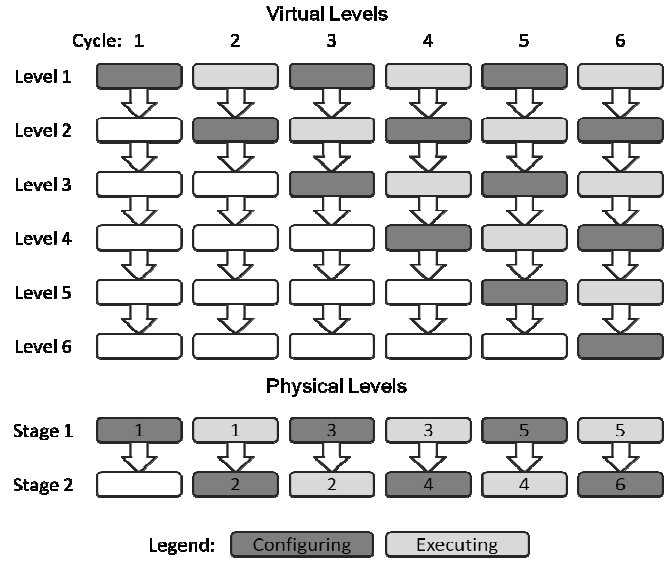

Fig. 5. Mapping of virtual levels in physical levels (stages).

In the 2-stage model, the execution occurs in parallel with reconfiguration: while a stage is running, the other is being reconfigured, according to the next virtual level. All levels are virtualized to be executed in the two available stages.

Figure 5 illustrates the hardware virtualization by mapping a configuration that uses 6 levels in two physical stages.In the first cycle, the virtual level 1 is configured in the stage 1 (physical level). In the following cycle, the virtual level 2 is mapped into the second physical level, while the configuration in the first stage is executed. In the third cycle the virtual level 3 is configured in stage 1,the same physical level that mapped the virtual level 1 before. This process is repeated until there is no more virtual levels to execute.

The control mechanism operates as follows: when an n-level is executing, the previous level output is stored, the level n+1 is reconfigured and the configuration bitsof level n+2 is fetched from context memory. These steps occur in parallel, in the same cycle.The time cycle (3) and the total execution time (4) can be computed by the following equations:

$$TS2 = Te + Ts \tag{3}$$

$$T2 = N \cdot TS2 \tag{4}$$

where:

TS2 = Execution time of a stage (time cycle)

Te = Execution time of a level (functional units)

Ts = Setup time of register

Tte2 = Total execution time

N = Number of levels to be executed

In this approach, as in the original, the reconfiguration time is shorter than the execution time of a level, which justifies the execution time of the stage in the equation.

What differentiates the 2-stage model from the 1-stage model is the reconfiguration process. In the 2-stage model the reconfiguration occurs during execution, while at the 1-stage model the reconfiguration and execution occur sequentially, which directly impacts the processing time. In contrast, the second model requires twice the area, since it is composed of two stages.

Table I shows this procedure in more details. After the configuration was found (e.g. instruction $I6), at each clock

TABLE I. EXECUTION STEPS.

Cycle	Processor Pipeline Stages					Original Mechanism	Proposed Mechanism
	IF	ID	EX	MEM	WB		
1	$I5	$I4	$I3	$I2	$I1		
2	$I6	$I5	$I4	$I3	$I2	PC found	PC found
3			$I5	$I4	$I3	Fetch Conf.	Fetch Initial Context
4				$I5	$I4	Conf. Array	Wait WB
5					$I5	Waiting Write Back	Wait WB || Fecth Level 1
6						Fetch Reg.	Fetch Reg. || Conf. Level 1 || Fetch Level 2
7						Executing	Executing Level 1 || Conf. Level 2 || Fetch Level 3
8						Executing	Executing Level 2|| Conf. Level 3 || Fetch Level 4
9						Executing	Executing Level 3 || Conf. Level 4
10						Executing	Executing Level 4
11	$I31					Write Back	Write Back
12	$I32	$I31					

cycle one level is fetched from the context memory. The fetch of the next level will be performed while the previously fetched level is executing. Therefore, no additional delay is inserted and the performance is maintained. The same technique can be easily adapted to other reconfigurable systems.

V. Experimental Results

In our study we have used a SystemC model of both reconfigurable system and the MIPS R3000 processor executing the Mibench Benchmark Suite [19] to extract performance results and the number of reconfiguration memory requests. A VHDL version was used to gather energy data and area. The experiments were performed using the CMOS 90 nm technology.

As presented in Table II, four different array setups were considered to evaluate the performance. Each setup is composed of a different number of functional units and levels. Setup 4 is the largest one.

TABLE II. Different Setups For The Array

	Setup 1	Setup 2	Setup 3	Setup 4
Total #Levels	8	16	32	64
Total #Rows	24	48	96	192
#ALU / level	24	24	36	36
#Multipliers / level	1	2	2	2
#Ld|St /level	2	6	6	6

A. *Performance*

In the virtualized model, as in the original one, each level has a computation time equal to one cycle of the processor. Due to the insertion of the registers between the physical stages in the virtualized model, the critical path of the functional units has changed (an increase of 0.5%). However, this delay does not cause any performance loss because even with this small increment the critical path of the reconfigurable array is shorter than the MIPS processor.

Nevertheless,if this technique is used in other architectures, the register setup time may cause small increments in the critical path, so the operating frequency may be affected. One way to reduce this impact is increasing the number of stages (physical levels) and putting the registers every two or four levels, decreasing their influence in the total processing time. For comparison purposes, let us consider an array with the same number of functional units per level of the Setup 4, but with an unlimited number of levels (referred as Setup Inf.).

Figure 6 presents the speedup of each benchmark application, using the MIPS R3000 as baseline and considering the Setup 4 and varying the number of entries (configurations) that the context memory can keep. A context memory with more than 256entries has not shown significant performance gains, considering this benchmark set.

Figure 7 shows the speedup obtained for each of the application with an infinite number of levels (Setup Inf.). Comparing these two figures, one can be see that the applications that benefited the most from this setup are *RjindaelD* and *RjindaelE*, with a gain of approximately 130% and 100% in speedup compared with the Setup 4 (which has the best speedup results among all setups).These applications required 85 levels to achieve this speedup.

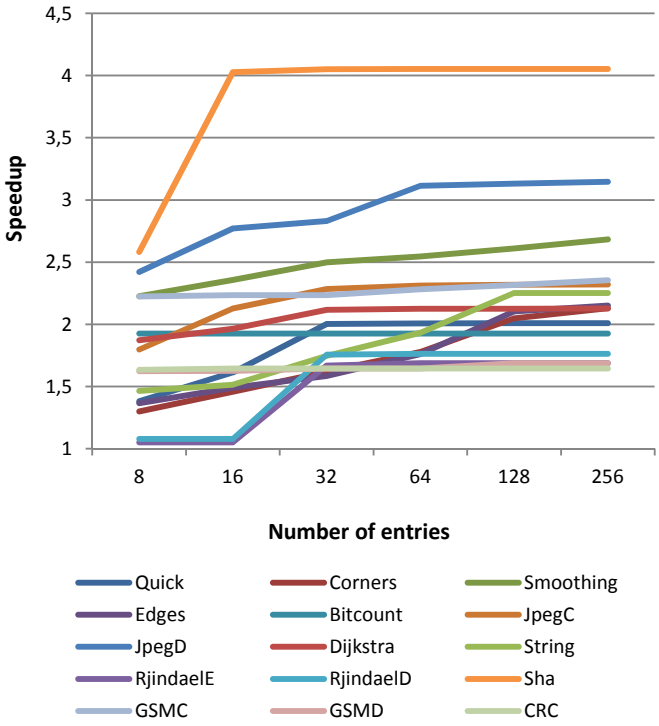

Fig. 6. Speedup achieved by Setup 4 for Mibench benchmark.

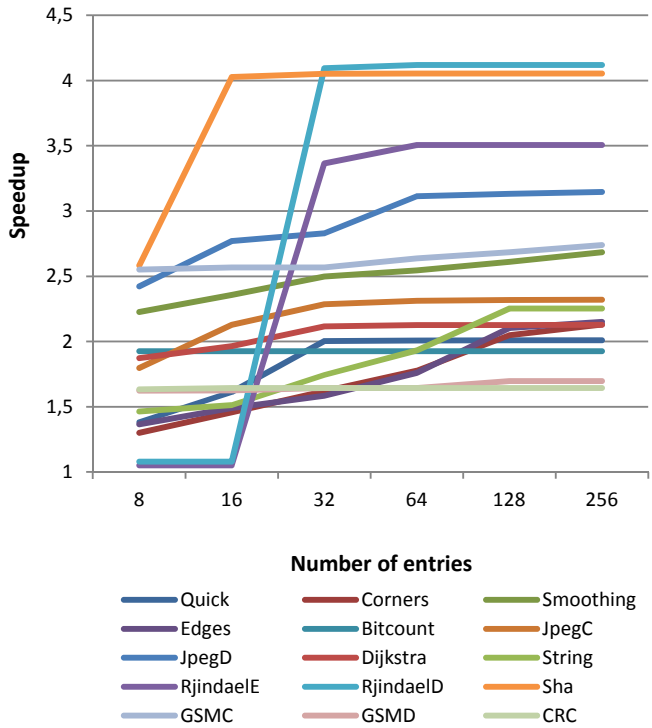

Fig. 7. Speedup achieved by Setup Inf. for Mibench Benchmark.

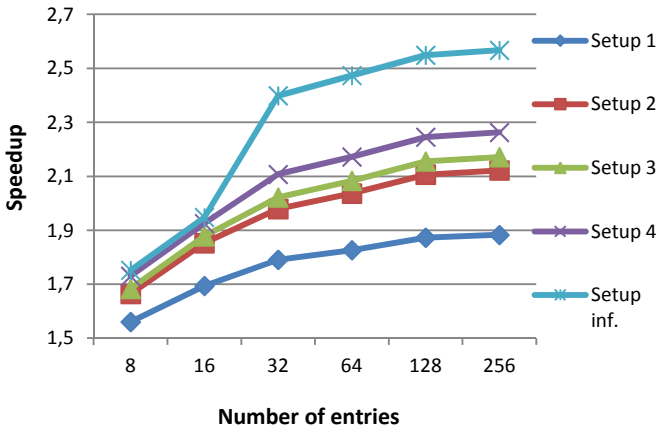

Fig. 8. Average speedup.

Figure 8 presents the average speedup for each setup in Table II and the Setup Inf. In this figure we can see that with a unlimited number of levels, larger segments of the applications may be executed in the array and, consequently, a higher performance can be achieved. The average speedup of the infinite setup was of 18% and 13% higher than Setup 3 and 4, respectively.

B. Area

The virtualization technique significantly reduces the area, since now only 2 physical levels are implemented. The Table III shows the results in the area (gates) and the relation of reconfigurable array compared to the MIPS R3000 processor.

The Setup 3 and 4 needed an area of 161 and 322 times greater than the size of the processor, respectively. Through the virtualization technique, this size decreased in a factor of 10 times. These results show that the reconfigurable architectures can save area using hardware virtualizationtechnique. In situations wherethe application mapping exceeds the size of the reconfigurable array, this technique allows to better exploit the reconfigurability of hardware and achieve higher performance in some applications.

TABLE III. AREA RESULTS

		Setup 1	Setup 2	Setup 3	Setup 4
Area (Gates)	**Original**	785.743	1.621.024	4.335.680	8.671.360
	Pipeline	196.692	202.884	271.364	271.364
Area in relationto MIPS	**Original**	29,2	60,3	161,3	322,5
	Pipeline	7.3	7.5	10.1	10.1
Reduction (%)		75,0%	87,5%	93,7%	96,9%

VI. CONCLUSIONS AND FUTURE WORKS

In this paper, we show the technique of hardware virtualization applied to a coarse-grained reconfigurable array. The results show that hardware virtualization is a valuable concept. In the architecture used as case-study, the performancewas maintained with an area reduction of up to 96.9%. As future work, we will use omega networks for optimizing the interconnections. Therefore, the union of both techniques will allow further reductions in the area used by the reconfigurable array.

REFERENCES

[1] P. Ienne and R. Leupers, “Customizable Embedded Processors: Design Technologies and Applications”. San Mateo : Morgan Kaufmann, 2006.

[2] T. J. Todman, G. A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P. Y. K. Cheung, “Reconfigurable computing: architectures and design methods”, Computers and Digital Techniques, IEE Proceedings, vol.152, no.2, pp. 193- 207, Mar 2005.

[3] R. Hartenstein, “A decade of reconfigurable computing: a visionary retrospective”, In Proceedings of the conference on Design, automation and test in Europe. Munich, Germany: IEEE Press, 2001.

[4] A. C. S. Beck, C. A. L. Lisbôa, L. Carro. Adaptable Embedded Systems. Springer New York, 2013.

[5] A. C. S. Beck, L. Carro. Dynamic Reconfigurable Architectures and Transparent Optimization Techniques: Automatic Acceleration of Software Execution. Springer, 2010.

[6] S. Borkar, “Electronics beyond nano-scale CMOS”. In Design Automation Conference, 2006 43rd ACM/IEEE, pages 807–808, 2006.

[7] C. Plessl and M. Platzner, “Virtualization of hardware – introduction and survey”. In Proc. 4rd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA), pages 63– 69. CSREA Press, 2004.

[8] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev and L. Carro. “Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications”. In: Design, Automation and Test in Europe, 2008, Munique.

[9] M. B. Rutzig, A. C. S. Beck Filho and L. Carro, “Balancing Reconfigurable Data Path Resources According to Applications Requirements”. In: 15th Reconfigurable Architecture Workshop, 2008, Miami. Proceedings of 15th Reconfigurable Architecture Workshop, 2008.

[10] R.S. Ferreira, M. Laure,M. B. Rutzig, A.C.S. Beck and L. Carro. “Reducing Interconnection Cost in Coarse-Grained Dynamic Computing through Multistage Network”. In: International Conference on Field Programmable Logic and Applications, 2008, Heidelberg.

[11] D. H. Lawrie, “Access and alignment of data in an array processor”, IEEE Trans. Comput., vol. 24, no. 12, 1975.

[12] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek and A. Dehon, “Stream computations organized for reconfigurable execution (SCORE)”. In Proc. 10th Int. Conf. on Field Programmable Logic and Applications (FPL), pages 605–614, 2000.

[13] C. Plessl and M. Platzner, “Zippy: A coarse-grained reconfigurable array with support for hardware virtualization”, in Proc. 16th Int. Conf. on Application-specific Systems, Architecture and Processors (ASAP 05), July 2005, pp. 213–218.

[14] T. Fuji, K. Furuta,M. Motomura,M. Nomura,M. Mizuno, K. Anjo,K. Wakabayashi, Y. Hirota,Y. Nakazawa,H. Itoh andM. A. Yamashina, “A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture”. In 46th IEEE Int. Solid-State Circuits Conf. (ISSCC), Dig. Tech. Papers, pages 364–365,1999.

[15] S. Goldstien, H. Schmit, M. Moe, M. Budiuy, S. Cadambi, R. Taylora and R. Laufer. “PipeRench: A Coprocessor for Streaming Multimedia Acceleration”. in Proc. International Symposium on Computer Architecture (ISCA), Atlanta, GA, 1999.

[16] H. Schmit, D. Whelihan, M. Moe, B. Levine, and R. R. Taylor. “PipeRech: A virtualized programmable datapath in 0.18 micron technology”. In Proc. 24th IEEE Custom Integrated Circuits Conf. (CICC), 2002.

[17] A. C. S. Beck and L. Carro, “Automatic Dataflow Execution with Reconfiguration and Dynamic Instruction Merging”. In: Very Large Scale Iintegration, VLSI-SOC, 2006, Perth. Proceedings. New York: IEEE Computer Society, 2007. p. 30–35.

[18] T. B. Ló, A. C. S. Beck, M. B. Rutzig and L. Carro,”A low-energy approach for context memory in reconfigurable systems”. In Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010.

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst and T. M. Austin. “MiBench: A Free, Commercially Representative Embedded Benchmark Suite”, in 4th Workshop on Workload Characterization, 2001.