Improving Embedded System Design by means of HW-SW Compilation on Reconfigurable Coprocessors [*]

José M. Moya †
josem@die.upm.es

Fernando Rincón ‡
Fernando.Rincon@uclm.es

Francisco Moya ‡
Francisco.Moya@uclm.es

Juan Carlos López ‡
JuanCarlos.Lopez@uclm.es

† Dept. Electronic Engineering
Technical University of Madrid
Madrid. Spain

‡ Dept. Computer Science
U. of Castilla-La Mancha
Ciudad Real. Spain

ABSTRACT

This article describes a new approach to HW-SW codesign for complex embedded systems, using high-level programming languages. Unlike in previous approaches, the designer does not need to acquire new skills, because most of the design process is automated. The hardware extensions are implemented as simple coprocessors consisting of a reconfigurable datapath and a control memory. Our approach is demonstrated with a simple image processing application, obtaining a 100% performance improvement.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other architecture styles—*Adaptable architectures*; C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Retargetable compilers*

General Terms

Design, Performance

Keywords

Hardware-software codesign, Reconfigurable datapaths

1. INTRODUCTION

Nowadays, the embedded systems community tends to use standard general-purpose microprocessors with little or no specific hardware. This trend is consistent with the increased adoption of multi-source component-based methodologies to reduce development time.

Many factors contribute to this situation: 1) a vast majority of the embedded systems designers are not familiar with state-of-the-art hardware design tools; 2) there are lots of tightly integrated, well tested tool chains for embedded software development but almost none of the EDA tools have been integrated into these tool chains yet; 3) considerably less training is needed to effectively use software development environments compared to their hardware counterparts; and finally, 4) software development enjoys shorter iterations during the development process. Nevertheless the current approach, compared to a judicious combination of hardware and software, leads to oversized microprocessors, lower performance and potentially higher production costs.

The introduction of application-specific hardware during the design of an embedded system brings a lot of problems. First, methodologies and tools used for hardware and software development are very different from each other. They use different specification languages and there are very few engineers with enough expertise in both, hardware and software development. Second, partitioning the functionality between hardware and software components must be done very early in the design process, which usually leads to sub-optimal results. Third, given there is not an unified methodology, hardware development and software development are carried out independently of each other. This precludes potential optimizations and makes it harder to integrate both parts. Finally, current technologies for designing HW-SW systems do not scale well. When we face large problems manual partition becomes impractical, and automatic partitioning (based on the iterative evaluation of a significant amount of alternatives) becomes unacceptably slow because the design space grows exponentially with the size of the problem.

In this paper we present a new reconfigurable architecture, and a new methodology to hardware-software codesign specially designed to overcome these limitations of current approaches to the design of complex embedded systems. We describe a general-purpose method to build customized hardware-software systems, with minor user intervention.

2. OUR APPROACH

Reuse becomes most important as complexity grows. A complex system, whose behavior needs more than a million lines of code to

[*]This work has been funded by the Spanish Ministry of Science and Technology (TIC2000-0583) and the Regional Government of Castilla-La Mancha (PBI-02-024)

be described, can not be designed from scratch, or the design cost and the time to market would be unacceptable.

Thus, flexibility and reuse become the key design goals for modern design environments for complex HW-SW systems. In this section we will describe our FLExible COSynthesis methodology (FLECOS), which has been designed with these goals in mind.

Along this paper we use the term *architecture* to refer to the set of available resources, the available functional units and the way they are connected.

The synthesis tools make no distinction between the functional units inside a microprocessor and other external functional units located in an external reconfigurable device. They just have different run-time characteristics.

The functional units are grouped into *subsystems*, which correspond to physical independent entities, such as a microprocessor, or a reconfigurable coprocessor. Complex communication mechanisms between two or more subsystems are also represented as functional units.

Next, we describe in more detail the hardware platform, the development tools, and the methodology we propose to bring HW-SW codesign techniques to programmers of embedded systems.

2.1 The hardware

We propose a hardware architecture based on a standard microprocessor connected to one or more REconfigurable Datapaths (RED devices), that work as coprocessors.

RED has been conceived with the following ideas in mind:

- Providing HW acceleration to critical operations but avoiding the problem of HW/SW partitioning. This implies working at the level of operations or system calls, which is a lower degree of granularity compared to classical HW/SW codesign approaches.
- Provide a flexible architecture that allows the implementation of operations at a variable level of complexity.
- Achieving a high percentage of utilization of reconfigurable hardware.
- Including a data cache to reduce the overhead produced when accessing to data memory.
- Reducing the overhead caused by communication between the main processor and the coprocessor, reducing the number of orders, but without increasing instruction width.

Figure 1 shows the top level architecture of the proposed datapath. Basically, the reconfigurable architecture is composed of a pipeline that will perform the desired operations. The pipeline will be fed with data from a local register bank. The data will then flow through the different stages of the pipe[1]. The intermediate results will be stored at the registers that isolate the stages, and when the operation is completed, the final result will be stored back in the register bank.

This simple architecture has two main advantages. Since the size of the operands will be fixed, there is no need for more sequential logic than the necessary to store intermediate data. Also the interconnection schema will be simpler compared with an FPGA.

The partitioning of an operation into datapath stages provides certain flexibility in the number of cycles needed to complete one operation, since all the stages can store their result directly into the register bank.

[1]Note in figure 1 how operands can be fed into the pipeline at different stages.

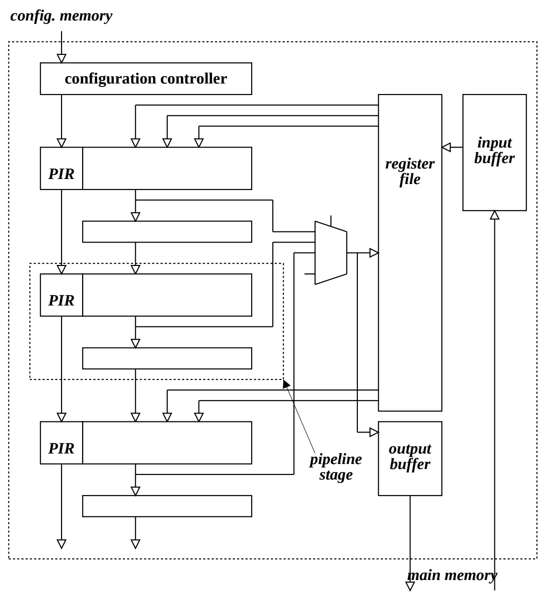

Figure 1: RED Architecture

Another important characteristic of this architecture is the use of dynamic reconfiguration. For every pipeline stage, the combinational logic has multiple configuration contexts [1], that can be loaded at the beginning to provide the different operations required to execute a task set. These operators are selected by the designer at compile-time, and the synthesis tools generate the sequence of active configuration contexts for every pipeline stage. The active configuration context is selected by the *pipeline instruction registers* (PIR), shown in figure 1.

As a result, pipelining will make it possible to start a new operation at each cycle, and dynamic reconfiguration will allow this operation to be different from the previous one, providing certain kind of parallelism between operations inside the pipeline, and therefore increasing the performance and the reuse of the dedicated hardware resources.

The use of dynamic reconfiguration techniques leads to new opportunities for the design of highly adaptable and fault-tolerant systems, with user-selectable tradeoff between quality of service and functionality.

2.2 The development tools

Any behavior specification may be implemented using different architectures. Some applications may require a large, fast, pipelined multiplier, but others can meet constraints with the minimum area consumption. In both cases it is useful to manipulate the compact, reusable behavior specification, while still being able to change the resources that will implement that behavior. However, most hardware-software codesign systems fail to adequately separate behavior from architecture. For example, SYSTEMC-based solutions advocate for a refinement process that incrementally adds structural information to an initial pure-behavioral specification, cluttering it needlessly. In many cases, the architecture selection is reduced to choose one microprocessor and one FPGA [6]. In most cases, there is very little control on how the hardware part will be implemented.

Another weak point of current codesign systems is the specification of constraints. Usually, the designer may only choose the maximum execution time and the maximum area of the hardware part, and in most cases, these constraints are checked only at the end of the design process.

We try to overcome these limitations by decoupling behavior, architecture, and design criteria. Consequently, a complete system specification in FLECOS is composed of three parts:

1. **Design criteria**. Algorithm for design space exploration. Dynamic strategies [7] may be used to improve the quality and efficiency of the exploration.

2. **Architecture description**. Number and type of the available functional units, and the system programming interface of this architecture (specifying the required instructions for every operation).

3. **Behavior specification**. A description of the behavior of the system using a general-purpose programming language, such as C, C++, Ada, Java or Pascal.

Unlike in previous refinement methodologies, the three parts of the specification are kept separate, simplifying reuse. As a result, the design of an embedded system may be seen as a search in the 3-D space. The mission of the synthesis tools is to map the behavior into the architecture, applying the specified design criteria.

Design criteria change very little or not at all for different designs. However, new transformations may be added to optimize specific applications.

Our current implementation of the FLECOS methodology is based on the industry standard GCC compiler suite. We have chosen GCC because it is freely available, high-quality, easily retargetable, portable, and widely used in the embedded systems community.

However, there are fundamental differences between the software development process and the required process for hardware-software codesign: 1) A software compiler has fixed design criteria, only slightly parameterizable with some compiler options. 2) Although GCC has been ported to many platforms, all the architecture-dependent code is static. There is no way to change the architecture without recompiling the whole compiler.

To overcome these limitations, we have moved all the design dependent code (which is most of the sources) out of the compiler, into shared objects, making the executable independent of the behavior, the architecture, and the design-criteria.

The architecture specification is implemented as a shared object containing all the architecture-dependent code of the compiler, and includes all the information of the hardware operators added by the designer.

The design criteria are represented by the optimization script, which defines the required sequence of transformations of the original specification to map it onto the specified architecture. This optimizer specification defines the goals of the synthesis process.

2.3 The methodology

The proposed methodology is similar to the typical profile-based refinement in software development, with the following phases: 1) design, code, and test a software-only solution; 2) get information of the run-time behavior of the program, based on profiling tools and simulators; and 3) optimize only what needs to be optimized. We will describe these three phases in more detail below.

Although these steps are common in the design of embedded software, to the best of our knowledge, it has never been applied to combined hardware-software designs. As there are fundamental differences with software-only development, most of the tools involved in the design process need to be redesigned to cope with extra hardware units, but the user interface is kept mostly unmodified from current software development tools.

Specification. The behavior specification does not change with respect to software, but we have to provide a suitable architecture description and the design criteria. See [8] for more information about the specification in the FLECOS environment.

The initial architecture description corresponds to the target microprocessor with no extra hardware. Later, based on the results, the designer can manually add the new RED operations to the architecture to maximize reuse opportunities. This architecture refinement process is guided by multiple profiling results and should be fairly simple for the designer. Of course, this task is susceptible to be automated, although not currently done.

The default optimizer should also be adequate as an initial specification of the design criteria.

Collecting run-time information. Depending of the specific requirements, we can use several tools to obtain useful data to optimize the design:

- Function profiling, using standard software profilers, allows a coarse-grain performance analysis, useful to discard most of the code in large designs.
- Basic-block profiling, using coverage tools such as `gcov`, allows the designer to decide the datapaths that should be implemented in RED. We provide a visual tool to evaluate different tradeoffs.
- The instruction-level simulator has been modified to generate a trace of memory accesses. This allows the designer to change the memory layout to minimize cache misses, or to maximize locality.
- As most of the power consumption in software systems is spent in level-one data cache, we can use a cache simulator (i. e. Dinero IV) to analyze the trace of memory accesses, and use its output to estimate the power consumption in every part of the specification.

The visual analysis tools give the designer hints on what should be moved to a RED coprocessor.

HW-SW optimization. Finally, to optimize the design we should only consider the bottlenecks shown by the analysis tools. The designer decides which basic blocks should be moved to a RED coprocessor, and automatically, the internal datapath is synthesized into a RED configuration.

When a basic block is moved to hardware, the resulting RED operators are added to the architecture description of the HW-SW compiler, and the datapath is replaced by a suitable mixture of core and RED operations.

3. AN EXAMPLE: A SOBEL FILTER

As an example of the proposed method, we will design a hardware-software version of an horizontal Sobel filter, that is frequently used for edge detection.

We will implement this filter in a prototyping board containing an ARM7TDMI processor and an Altera FPGA. The main goal to guide our decision will be performance. The results shown are based on simulation.

3.1 Specification

As previously stated, the first step should be the design and implementation of the filter as a software program. Figure 2 shows the main loop of our C implementation.

We use a modified version of the backend of GCC[2] as the HW-SW compiler, and thus, we can use all the specification languages

```
for (r=1; r<ROWS-1; r++)
  for (c=1; c<COLS-1; c++) {

    /* Apply Sobel operator. */
    pixel =    image_in[r-1][c+1]
             - image_in[r-1][c-1]
             + 2*image_in[r][c+1]
             - 2*image_in[r][c-1]
             + image_in[r+1][c+1]
             - image_in[r+1][c-1];

    /* Normalize and take absolute value */
    pixel = abs(pixel/4);

    /* Store in output array */
    image_out[r][c] = (unsigned char) pixel;
  }
```

Figure 2: Sobel filter specification

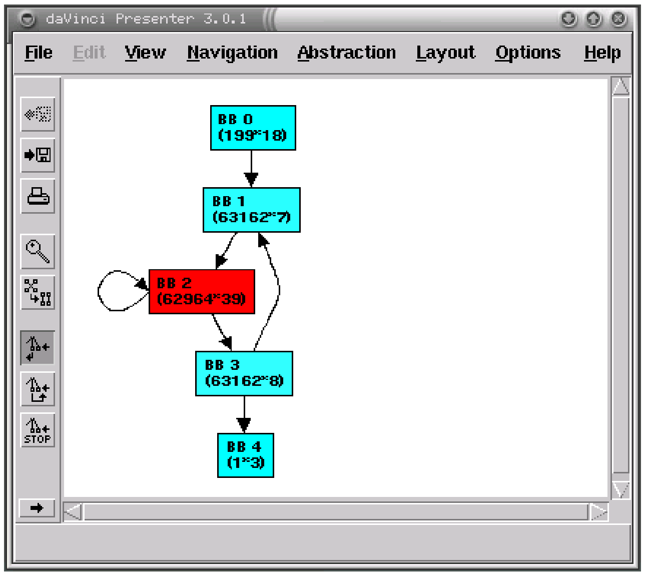

Figure 3: Interactive tool to analyze run-time properties (CFG view of the Sobel function).

supported by GCC (C, C++, Objective-C, Java, Ada, Fortran, Pascal, etc).

As we use standard programming languages, we can use standard tools to test and debug our specification in the development computer.

3.2 Collecting run-time information

Once we have a working program, we can obtain a working product by just cross-compiling the application for the target microprocessor, but it will not be optimized for our special needs. We can improve significantly our design (performance, cost, power consumption, etc.) adding a specialized RED coprocessor.

First, to make a fast hardware-software tradeoff, we need to discover the weak points of our design based on the design criteria. As the target microprocessor does not need to be the same as the development machine, we need to run the application in a simulator. We use the instruction-level simulator built into gdb, which is generated automatically from a description of the architecture, and it is easy to extend to include the new coprocessors.

The example presented here is a very simple program with only 4 basic blocks, and it is fairly obvious that the inner basic block is where the processor spends most of the execution time. However, with very complex designs it is not so easy and an automated tool is required. This tool shows the control flow graph and suggests the basic blocks that should be moved to hardware, but it is the designer's decision if they are actually moved or not. Figure 3 shows the control flow graph of our Sobel function, where the basic block BB2 is marked red to indicate that it is the one that determines the execution time of the whole function.

Thus, BB2 is the basic block that we should implement in a RED coprocessor. The assembler code of this block is shown in figure 4. It is interesting to note that most instructions are dedicated to the calculation of the addresses of the operands, and not the Sobel calculation itself. Thus, an application-specific address generation unit would certainly improve the performance by a huge factor. This is the main source of speedup in our example.

```
.L10:
    @ basic block 2
    ldr      r1, .L17+16
    add      r0, r6, r1
    add      r2, r0, r4
    mvn      r3, #320
    ldrb     r0, [r2, r3]
    ldrb     ip, [r2, r9]
    ldrb     r1, [r2, #1]
    rsb      r3, r0, ip
    ldrb     ip, [r2, #-1]
    add      r3, r3, r1, asl #1
    ldrb     r0, [r2, r8]
    ldrb     r1, [r2, sl]
    sub      r3, r3, ip, asl #1
    add      r2, r3, r0
    rsb      ip, r1, r2
    mov      r2, ip, asr #31
    add      r1, ip, r2, lsr #30
    mov      r2, r1, asr #2
    cmp      r2, #0
    rsblt    r2, r2, #0
    str      r2, [r7, #0]
    ldrb     ip, [r7, #0]
    strb     ip, [r5, r4]
    add      r4, r4, #1
    cmp      r4, lr
    ble      .L10
```

Figure 4: Assembler output for ARM7TDMI of the inner block.

3.3 HW-SW optimization

For this example we have used the original optimization sequence defined by GCC, which is specially tuned for performance. We have only changed the architecture description to make use of external RED operators. It is important to remark that all the functional units (internal and external) are taken into account by the different transformations, not requiring an initial hardware-software partition.

Our goal is maximum performance, and we have seen that it is mainly determined by the basic block shown in figure 4. Thus, we should concentrate on moving this whole basic block to a hardware coprocessor, reducing the total execution time. Figure 5 shows the GCC scheduling for this block, which is 35 instruction cycles long.

As it is best seen in figure 2, every iteration requires 6 read accesses to memory and 1 write access. If we do not alter the buses and memory hierarchy, taking into account that the ARM requires 2

```
clock  core
=====  ========================
0      r1='image_in'
1      r1='image_in'
2      r0=r6+r1
3      r2=r0+r4
4      r3=0xfffffebf
5      r0=zxn([r2+r3])
6      r0=zxn([r2+r3])
7      ip=zxn([r2+r9])
8      ip=zxn([r2+r9])
9      r1=zxn([r2+1])
10     r1=zxn([r2+1])
11     r3=ip-r0
12     ip=zxn([r2-1])
13     ip=zxn([r2-1])
14     r3=r1*2+r3
15     r0=zxn([r2+r8])
16     r0=zxn([r2+r8])
17     r1=zxn([r2+sl])
18     r1=zxn([r2+sl])
19     r3=r3-ip*2
20     r2=r3+r0
21     ip=r2-r1
22     r2=ip>>31
23     r1=r2 0>>30+ip
24     r2=r1>>2
25     {r2=abs(r2);clobber cc;}
26     [r7]=r2
    .
28     ip=[r7]
    .
30     [r5+r4]=ip
31     [r5+r4]=ip
32     r4=r4+1
33     cc=cmp(r4,lr)
34     pc={(cc<=0)?L317:pc}
```

Figure 5: GCC scheduling for the inner block.

instruction cycles for every memory access, every iteration would require at least 14 instruction cycles to complete (plus two more cycles to get the base address of the input image).

Moving everything to a RED coprocessor, we can reduce the total latency of this block to only 17 cycles (see figure 6). Once the designer selects the functionality of the operators that should be moved to a RED unit, the synthesis of the datapath is done automatically, using standard high-level synthesis tools. The only required user intervention is to select the basic blocks that should be moved to the coprocessor.

The copro-core unit, shown in figure 6, refers to the coprocessor datapath, while the copro-mem unit deals with external memory. The register bank has two separated data buses to allow simultaneous accesses from both units. The only restriction is that the same register can't be written from both units at the same time.

The resulting schedule can be interpreted as follows. From cycle 0 to 6 the copro-core unit computes the six memory addresses for the six operands of the Sobel operator, and the address to store the result. At the same time, as soon as the first memory address is ready, copro-mem starts accessing external memory. Since memory accesses take 2 cycles it's clear that at least 12 cycles are needed for obtaining all operands. However, since both units (copro-core and copro-mem) perform in parallel, it is possible to compute Sobel operator as long as new operands are extracted. When the last computation is finished the result is stored in the output image.

Meanwhile, the ARM unit increments the register where the current x coordinate is stored, checks if it has reached the limit, and jumps accordingly at the end of the basic block. Note also how the value in r4 is passed to RED, so the new column value is ready for the next iteration.

For the implementation of the described functionality only two new operations must be defined for RED. The first one (op1) performs address computations. Since addresses are generated adding a constant offset to the current (x,y) position being filtered, op1 will be equivalent to one adder plus some additional logic to select the right offset (just one state in the datapath).

The second operation needs three stages. The first one computes the first subtraction, the second subtracts two new operands and append its result to the accumulated from the previous stage. The third and last, appends again a new subtraction, shifts right the result and takes the absolute value.

The result is roughly a 100% improvement in performance with minimum design effort.

4. RELATED WORK

During the last decade, the research community has shown a strong interest in the unification of hardware and software specifications to simplify the design of complex hardware-software systems. High-level programming languages are common in large-scale system design and debugging, and they are now being considered for the whole design process.

There are also some commercial products that support C or C++ as specification languages for hardware design. But their main objective is to integrate the support of C/C++ models into existing design flows based on Verilog or VHDL. The specification language is extended to support the underlying model of computation of the HDL, and some constructs are restricted to avoid difficult analysis and transformations. Some notable examples of this approach are The Open SYSTEMC Initiative [10], and OCAPI from IMEC [9]. Our approach is the opposite, we integrate some hardware synthesis tools into the existing software design flows.

The Synopsys Nimble Compiler [6] has similar goals but takes a different approach. First, it implements a classical hardware-software codesign process, where the first step is the hardware-software partitioning, and then hardware and software are optimized separately. In contrast, we optimize hardware and software simultaneously, increasing opportunities to share resources. Second, the Nimble Compiler hides the architecture and the design criteria into the compiler, which we think is less flexible.

The Tensilica Xtensa Processor Generator [11] is also aimed at synthesizing hardware and software from a C specification, but the hardware is just an instance of a parameterized general-purpose processor, and the tools make no effort to include external specialized hardware.

Regarding the RED architecture, there are other related proposals still under research. One is GARP, from University of California at Berkeley [5], and is based on a MIPS core plus a reconfigurable array organized in a set of contexts. Another one is Chimaera [4], from Northwestern University. Chimaera is a reconfigurable functional unit that is able to provide a set of simultaneous hardware operators, that can be replaced dynamically, during the execution of the application. The third one is Piperench [3], from Carnegie Mellon University. PipeRench is a dynamically reconfigurable datapath that allows the implementation of pipelined operations that are not constrained by the physical number of stages in the datapath.

RED shares some features from all of them. Since it will work at the level of operations, the internal architecture does not have to be as general as typical FPGAs. It is a datapath much like PipeRench, although the granularity of the stages, the use of local memory and

```
clock  core          copro-core                            copro-mem
=====  ============  ==============================        =========
0                    (op1)   R0=&image_in[r][c+1]
1                    (op1)   R1=&image_in[r-1][c-1]        R7=[R0]
2                    (op1)   R2=&image_in[r][c+1]          R7=[R0]
3                    (op1)   R3=&image_in[r][c-1]          R8=[R1]
4                    (op1)   R4=&image_in[r+1][c+1]        R8=[R1]
5                    (op1)   R5=&image_in[r+1][c-1]        R9=[R2]
6                    (op1)   R6=&image_out[r][c]           R9=[R2]
7      r4=r4+0x1     (op2-1) S1 = R7 - R8                  R10=[R3]
8                    wait                                  R10=[R3]
9                    (op2-2) S2 = S1 + 2*(R9-R10)          R11=[R4]
10                   wait                                  R11=[R4]
11                   wait                                  R12=[R5]
12                   wait                                  R12=[R5]
13     R14=r4        (op2-3) R0 = abs((S2+R11-R12)/4)      R14=r4
15     cc=cmp(r4,lr)                                       [R6]=R0
16     cond. jump                                          [R6]=R0
```

Figure 6: RED scheduling for the inner block.

the reconfiguration schema is rather different. Reconfiguration is based, as in Garp, in the use of several switchable contexts. With respect to Chimaera, RED also provides simultaneous execution of instructions, but it is achieved through the use of a pipeline whose stages can change their functionality at each clock cycle.

5. CONCLUSIONS AND FUTURE WORK

The proposed architecture and methodology make it possible for unexperienced designers to develop hardware-software systems in an easy, and flexible manner, leading to greatly improved designs.

We have shown with a simple example how performance can be significantly improved with minor human intervention.

This seams to be the natural extension of the methodologies and tools currently used in the design of microprocessor-based embedded systems, and so it should be easy to adopt in current design flows.

The RED architecture has proved to be a very effective solution for hardware-software codesign, because of its special characteristics:

- It is more flexible and general-purpose than a traditional coprocessor. It allows the implementation of all type of operators, that can even handle more than two operands simultaneously.
- Its internal architecture is pipeline-oriented. For that reason the complexity of the programmable cells that compose it, as well as the routing resources, are expected to be much lower compared to traditional FPGAs. This would make RED more suitable to be integrated into a SoC (together with a microprocessor core).
- The fact of having several contexts almost eliminates reconfiguration latency, although at the cost of more configuration memory.
- It is very easy to integrate into a traditional compiling process, giving very good results with a minimum effort.

Our method and tools make no distinction between hardware and software resources. This leads to true codesign, in contrast to other methodologies which develop hardware and software separately.

Thanks to its flexible architecture, it also constitutes a good framework to experiment with new strategies for design space exploration, and new design methodologies. Currently, we are working on the integration of a module for automatic design space exploration. Also, we are implementing a lightweight operating system kernel that will support commonly used abstractions in subsystems with very different dynamic behavior (microprocessors, FPGAs and ASICs).

6. REFERENCES

[1] A. DeHon. *Reconfigurable Architectures for General Purpose Computing*. PhD thesis, MIT, 545 Technology Sq., Cambridge MA 02139, September 1996.

[2] GNU Compiler Collection. [on-line]. Available from WWW: `http://gcc.gnu.org/`.

[3] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. Piperench: a coprocessor for streaming multimedia acceleration. *ISCA*, 1999.

[4] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 1997.

[5] R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 1997.

[6] Yanbing Li, Tim Callahan, Ervan Darnell, Randolf Harr, Uday Kurkure, and Jon Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *Design Automation Conference*, 2000.

[7] F. Moya, J. M. Moya, and J. C. López. Global design space exploration strategy for system synthesis. In *Proc. of the 6th European Concurrent Engineering Conference*, April 1999.

[8] J. Moya, F. Moya, S. Dominguez, and J. Lopez. Multi-language specification of heterogeneous systems. *Forum on Design Languages (FDL'2000)*, September 2000.

[9] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsen. A programming environment for the design of complex high speed ASICs. In *Design Automation Conference*, 1998.

[10] Open SystemC Initiative. [on-line]. Available from WWW: `http://www.systemc.org/`.

[11] Tensilica Xtensa Processor Generator. [on-line]. Available from WWW: `http://www.tensilica.com/`.