Implementation of a UMTS Turbo-decoder on a dynamically reconfigurable platform

Alberto La Rosa Claudio Passerone Francesco Gregoretti Luciano Lavagno
Politecnico di Torino - Dipartimento di Elettronica - Torino, Italy

Abstract

Modern embedded systems must execute a variety of high performance real-time tasks, such as audio and image compression and decompression, channel coding and encoding, etc. Reconfigurable platforms can effectively be used in these cases, because they allow to re-use the architecture for as many applications as possible.

This paper describes the implementation of a UMTS turbo-decoder on one such platform, the XiRisc reconfigurable processor. Our goal is to test the development framework and design flow that we already developed on a real industrial example. Our results shows that, with some manual effort from the designer, very good performance improvements can be achieved, using a flow close to embedded software development.

1. Introduction

Reconfigurable computing is emerging as a promising mean to tackle the ever-rising cost of design and masks for Application-Specific Integrated Circuits. Adding a reconfigurable portion to a Commercial Off The Shelf IC enables it to potentially support a much broader range of applications than a more traditional ASIC or microcontroller or DSP would. Moreover, *run-time reconfigurability* even allows one to adapt the hardware to changing needs and evolving standards, thus sharing the advantages of embedded software, with higher performance and lower power than a traditional processor, thus partially sharing the advantages of custom hardware.

A key problem with dynamically reconfigurable hardware is the inherent difficulty of programming it, since neither the synthesis, placement and routing-based hardware design flow, nor the compile, execute, debug software design flow directly support it. In a way, dynamic reconfiguration is a hybrid between software, where the CPU is "reconfigured" at every instruction execution and memory is abundant, and hardware, where reconfiguration occurs seldom and very partially, and memory is a scarce resource. Several approaches, some of which are summarized in the next section, have been proposed to tackle this problem, both at the architecture and at the CAD level.

In this case study, we consider an architecture ([6]) in which an FPGA has been added as one of the functional units of a RISC processor. This approach has one huge advantage over its competitors described in the next section: it can exploit, as discussed in the rest of this paper, a standard software development flow. The FPGA can be managed directly by the compiler, so today designers just need to tag portions of the source code that must be implemented as single FPGA-based instructions. In the future, we are looking at ways of automating this extraction, e.g. along the lines of [2, 4, 13]. The disadvantage of the approach is that the reconfigurable hardware accesses its main data memory via the normal processor pipeline (load and store instructions), thus it partially suffers from the traditional processor bottleneck. Without a direct connection to main memory, we avoid to face memory inconsistency problems ([9]) thus simplifying the software programming model.

In this paper we show that, by efficiently organizing the layout of data in memory, and by exploiting the dynamic reconfiguration capabilities of the processor, one can obtain an impressive speed-up (better than $11\times$) for a very compute-intensive task, like turbo-codes. The goal thus is to bridge the huge distance between software and hardware, currently at several orders of magnitude, in terms of cost, speed and power. The result is a piece of hardware that is reconfigured every 100 or so clock cycles (at least in the application described in this paper; other tradeoffs are possible for applications with a more static control pattern), and within that interval is devoted to a single computation task. Ideally, we would like to bring a traditional, hardware-bound task, that of decoding complex robust codes for wireless communication, within the "software" computation domain.

In the case of our target processor and software/hardware development flow, the design space exploration takes the form of identifying maximally time-consuming portions of the code, bounded by a number of accesses to memory and/or registers that is within the limit of one FPGA-based instruction. Once they are identified, possibly with the help of (currently manual) code restructuring, they can be tagged, and their impact on the overall execution time analyzed.

2. Related work

Coupling a general purpose processor with an FPGA generally requires to embed them on the same chip, oth-

1530-1591/04 $20.00 (c) 2004 IEEE

erwise communication bandwidth limitations may severely impact the performance of the entire system. There are several ways of designing the interaction between them, but they can be grouped into two main categories: (1) the reconfigurable array is another functional unit of the processor pipeline [14, 19, 11]; (2) the reconfigurable array is a co-processor communicating with the main processor [5].

In all the cited examples a modified C compiler is used to program the processor and to extract candidate kernels to be downloaded on the FPGA. In particular, the GARP compiler [5] and the related Nimble compiler [18] identify loops and find a pipelined implementation, using profiling-based techniques borrowed from VLIW compilers and other software optimizations.

Tensilica offers a configurable processor, called Xtensa, where new instructions can be easily added at design time within the pipeline. Selection of the new instructions is performed manually by using a simulator and a profiler. When the Xtensa processor is synthesized, a dedicated development tool-set is also generated that supports the newly added instruction as function intrinsics.

The Lisa tool-set [12] was designed specifically to facilitate the development of Application-Specific Instruction Processors, because it generates a complete tool chain from a single specification of the ASIP's instruction set.

Turbo Codes specifications for the third generation of cellular phone systems [1] were finalized in 1994. Since then, many research activities focused on their efficient implementation both in hardware and in software. More precisely, hardware approaches are common when high data rate are considered ($>$ 10 Mbps/iteration), while software ones are usually preferred for low data rate applications.

Reconfigurable approaches were considered too: literature shows examples ([17]) of efficient FPGA implementations even when targeting moderately high data rates (5 Mbps/iteration). As the processing power of general purpose processors and DSPs is increasing, attention is moving back to software implementations, since they are more flexible and less expensive. For example [10] and [7] describe efficient implementations of turbo decoding on state-of-the-art DSP architectures likes C6x and StarCore. Both fully exploit instruction level parallelism and dedicated functional units, like saturation adders, max operators or entire computation kernels.

Memory layout optimization plays an important role. Several studies report the minimum required precision for input and output values and internal computation (e.g. [3]), or memory allocation strategies that minimize data transfer or overall temporary memory requirements.

3. The development framework

We are targeting a reconfigurable processor that is being developed by the University of Bologna and is described in [6]. The processor is based on a 32-bit RISC (DLX) reference architecture, where both the instructions set and the architecture have been extended to support: (i) dedicated hardware logic to perform multiply-accumulate calculation and end-of-loop condition verification, (ii) a double datapath, with concurrent VLIW execution of two instructions, limited to arithmetic and logic instructions, and without direct access to the external memory subsystem and (iii) an FPGA to implement in hardware special kernels and instructions. The FPGA can be dynamically reconfigured at run-time; each cell may store up to 4 configurations that can be instantly switched when needed.

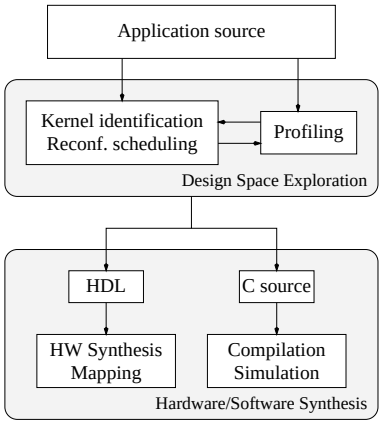

Figure 1. The proposed design flow

The FPGA, called PiCoGA (pGA), sits in the data-path of the processor, and can simultaneously access four input and two output registers at each clock cycle. Because of its location, it is viewed as another function unit of the processor, that can be used to implement new instructions. Being dynamically reconfigurable, the added instructions can be swapped in and out, thus virtually increasing the total area available. The pGA is organized in rows of CLBs and has a control unit that can independently activate each row to manage the flow of data and implement loops (including data-dependent ones). The array is capable of performing instructions with both fully pipelined execution and non-unit issue delay (due to shared resources).

We implemented a design flow to support the reconfigurable processor [15]. It starts from a *fully C* initial specification, where sections that must be moved to the pGA are manually annotated, and automatically generates the assembler code, the simulation model, and a hardware model useful for instruction latency and datapath cost estimation. A key characteristic is that it supports compilation and simulation of software including user-definable instructions, without the need to recompile the tool chain every time a new instruction is added.

The design flow, shown in Figure 1, is divided into two steps: design space exploration and hardware/software synthesis. In the first step, groups of statements, called *kernels*, that are candidate to become user-defined instructions are identified with the help of profiling. In the second step, the binary code and the bit-stream for the pGA are generated. Once kernels are identified, their control-data flow graphs (CDFG) are analysed to find a proper set of pGA instructions to cover them. Instructions are selected based on their

effect on the final performance, but also depending on constraints on the implementation, such as the maximum number of inputs and outputs that can be simultaneously read or written from the register file, or the amount of intermediate data values that can be stored in the pGA. Optimizations of the data layout in memory and of the bit width are also considered at this stage.

Different splittings of the CDFGs can be analysed and simulated to find the best trade-off between performance and area occupied on the pGA. When an optimal implementation is selected, the original source code (which is annotated in the CDFG) can be used to generate an HDL description of the hardware.

4. The UMTS turbo-decoder

We considered the specification of a turbo-decoder that follows the 3GPP recommendations for UMTS cellular phone systems [1]. We assume to receive a bit stream from a Recursive Systematic Convolutional encoder, with an 8 state trellis and rate equal to 1/3 (i.e. three bits are transmitted for each bit in the message, two of them being parity bits). The message size is variable, between 40 and 5114 bits. The turbo-decoder algorithm has many advantages over other encoding/decoding mechanisms when the transmission channels is noisy, and has very low signal to noise ratios. However, it has a high computational complexity, which has limited its application until recently, when enough computational power has become available. For this reason it is also necessary to perform several approximations over its textbook representation, in order to obtain an efficient and cost-effective implementation.

The decoder implements an iterative algorithm, repeated until convergence. Each iteration consists of several components, among which the most computationally intensive ones are (1) trellis metrics computation (γ); (2) forward probabilities computation (α); (3) backward probabilities computation (β); (4) maximum likelihood ratio computation (LLR).

4.1. Kernel identification

The four kernels mentioned above were identified from profiling information and from past work in this area ([8, 16]). When selecting dynamic instructions, we also had to take into account the limit to 4 register reads and 2 register writes imposed by the architecture on each pGA instruction. This translates to looking for subsets of the CDFG that have 4 input and 2 output arcs at most.

To increase the likelihood of finding such subsets, input and output data was packed into registers in couples, similarly to what is commonly done in Single Instruction Multiple Data (SIMD) machines. This was possible because an analysis of the required precision on the data used in the decoding process proved that 8 bits (rather than the 16 commonly used in software implementations of turbo-decoding) were sufficient to get correct results. Figure 2 represents four identical computations operating on

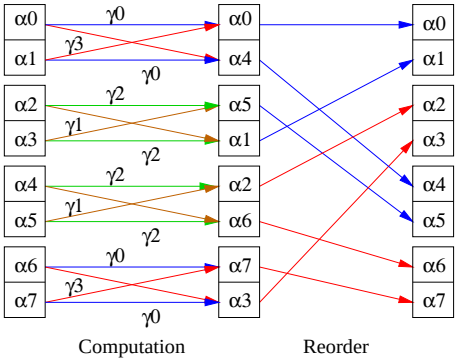

Figure 2. Optimization of the memory layout

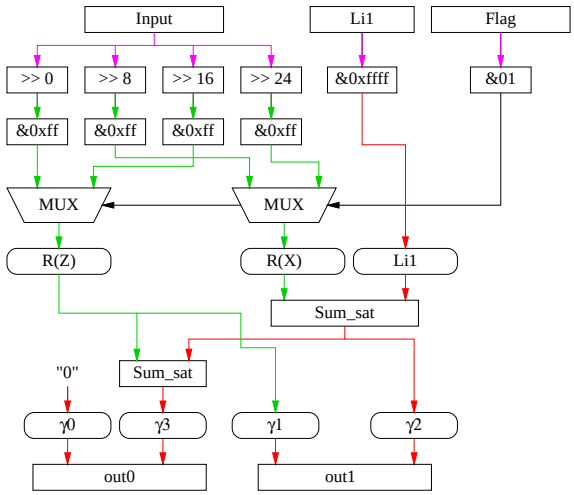

Figure 3. CDFG of Compute Gamma operation

8 operands and using 4 parameters, showing that an additional reordering of the output data is required. This added operation was also a candidate to be implemented in hardware on the reconfigurable device.

Beside allowing more data to be exchanged with the dynamic instructions, packing also had the added advantage of reducing the number of memory accesses to fetch and store operands during the decoding process. This is a benefit also when no dynamic instructions are used at all, and therefore *all results that we present are compared to this memory-optimized fully software implementation*.

The first kernel that we analyzed computes the metrics γ_i that are used in the successive recursive algorithm. The operation, that we called *Compute Gamma*, takes as input the data coming from the channel, and produces the values $\gamma_1, \ldots, \gamma_3$. Since input data can be represented on 8 bits, and two γ values can be packed into an output register, all values can be computed with a single dynamic instruction, whose CDFG is represented in Figure 3. Note that all additions include saturation logic, in order to keep the correct sign for all operand values.

The next two kernels that were analyzed are very similar: in fact, forward and backward probabilities use the very same computations, but traverse the trellis in opposite directions. This allowed us to derive a single dynamic instruction that can be shared by both kernels, plus two differ-

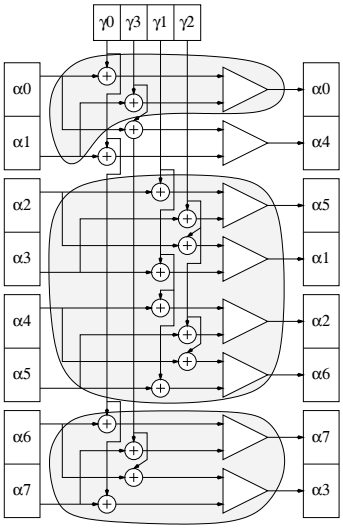

Figure 4. Butterfly operation on all operands

ent dynamic instructions to properly reorder the output data based on the traversal direction. The reordering is needed, as in the γ_i computation, due to the way data is packed into 32-bit wide registers, and would not be necessary in a standard implementation.

The main computation is a butterfly operation that uses the γ metrics previously calculated, and that is recursively applied to a set of operands, called $\alpha_0, \ldots, \alpha_7$ for the forward probabilities and $\beta_0, \ldots, \beta_7$ for the backward probabilities. After each step, the computed values need to be normalized before a new iteration can be applied.

Thanks to the packing of operands and metrics, we were able to perform two butterfly operations in a single dynamic instruction, despite the limitations in the input and output registers. Three of all the possible selections are shown in Figure 4, where the entire computation for all the operands is represented. The one on the top, which takes as input two operands and two metrics, computes half of the butterfly and produces one output operand. The one on the bottom computes both halves at the same time, using the same inputs and an additional output. Finally, the one in the middle, takes four operands and two metrics, packed into three input registers, and generates four output operands, packed into two registers, thus satisfying the limits imposed by the architecture.

Since four butterflies are needed for each iteration (either of the forward or of the backward computation), two successive overall executions are necessary. If a pipelined implementation of the butterfly is used, then a very good performance improvement can be achieved,

Reordering was also implemented using dynamic instructions. In this case, two instructions are derived from the kernel CDFG, including normalization. The two differ in the ordering of the output operands only. Each reordering operation is applied to 4 α or β parameters at a time, because of the output limit of 2 registers (which correspond to 4 packed parameters). Two successive calls are needed to complete an iteration, so pipelined implementations should be chosen to get best results. The first instruction execution uses operands with indices 0, 4, 5 and 1 of Figure 2, while the second one uses operands with indices 2, 6, 7 and 3. The structure of the two is identical, so we can use the same instruction.

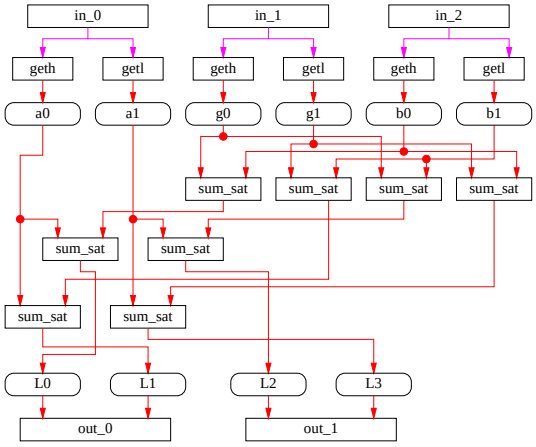

Figure 5. CDFG of Sum3 operation

The last kernel involves the computation of the maximum likelihood ratio, i.e. the probability that a transmitted bit is 0 or 1. This is a recursive algorithm that uses a threshold to determine when to stop. The textbook implementation includes several multiplications, while in practice logarithmic data encodings allow one to use additions instead. However, there is still the problem of calculating the log of a sum of exponentials, which is indicated using the `max*` elementary operator. Several approximations are possible, and we chose the *Linear-log-map* because it offers a good precision and a fast convergence [16].

The kernel itself was covered using two dynamic instructions: the first one computes the sum of three operands, while the second one computes the `max*`. As in the case of the butterfly, by appropriately packing the data, we were able to perform two parallel operations at the same time, as shown in Figure 5 for the CDFG for the instruction `sum3`.

4.2. Mapping to the pGA

Mapping dynamic instructions to the pGA means taking the CDFGs that were selected in the previous step and generating a configuration bit-stream for the pGA. Nodes present in the CDFG may need to be decomposed in elementary operations, in case they cannot be directly implemented by the Combinational Logic Blocks (CLBs) that are available in the array fabric.

Concurrency can be exploited to decrease the latency of the instructions. When data dependencies and long combinational paths force a multi-cycle operation, a pipelined implementation helps increasing the performance if the same instruction is called multiple times sequentially on different data. On the other hand, the improvements are limited by the available input and output bandwidths.

In this section, we just consider the `max*` and the butterfly (which includes a `max*`) operators. Similar consider-

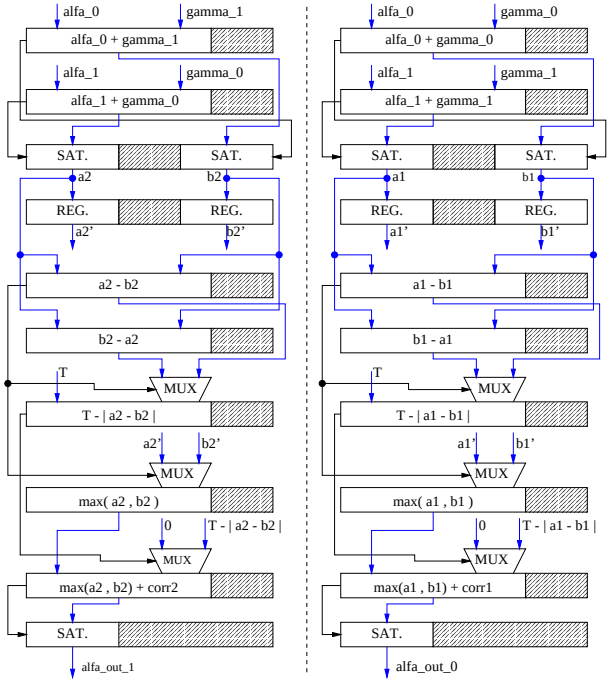

Figure 6. Mapping of the butterfly operation

ations can be done for other dynamic instructions, as well.

The `max*` implementation that we chose is a 2-segment piece-wise linear approximation of the function $\ln(e^x + e^y)$. It can be represented with the following C code:

if (abs(a - b) $\leq$ T) corr = ((T - abs(a - b)) >> S);
else corr = 0;
return (max(a, b) + corr);

To implement the abs() function, we concurrently compute both $(a - b)$ and $(b - a)$ and we use the sign bit to select one of the two, using multiplexers that are built into the array. Computing the max() also involves the sign of one of the two above subtractions, which is again used as control bit of a multiplexer. Registers are also added in the implementation to allow the instruction to be used in pipeline. Note also that the last addition requires to saturate the result, to keep the sign value correct.

The `max*` that we implemented using this algorithm occupies 5 rows of the array, out of a maximum of 24. As mentioned above, two `max*` can be executed by one instruction, by exploiting packed operands. The latency of the function is 5 clock cycles.

The butterfly instruction involves several additions, followed by a `max*` operation. Figure 6 shows the configuration of the top part of the array: here two halves of a butterfly are computed. The 10 rows that are shown are almost fully utilized, while the next 10 rows can be used to compute the other two halves. This results in computing two complete butterflies in a single pipelined dynamic instruction, with 20 out of 24 configured rows. The first 4 rows compute additions with saturation, while the next 6 perform the `max*` operation on the intermediate results, that are stored into registers. The total latency is 7 clock cycles,

Step	Speed-up	Exec. cycles	Saved cycles
Original	1×	177834	-
Gamma	1.02×	173706	4128
LLR	1.83×	96913	80921
Butterfly	1.53×	115816	62018
Reorder	1.10×	161826	15972
Final	11.73×	15157	162677

Table 1. Experimental results on 40-bit messages

but two such instructions can be issued in pipeline, resulting only in 8 clock cycles to compute all the butterflies for either the forward or the backward predictions.

The other dynamic instructions are easier to implement, as they only involve additions with saturation and/or data movements, which can be implemented using the routing facilities of the array itself.

5. Experimental results

We performed several simulations of the turbo-decoder, with different objectives. The first simulations were aimed at analyzing the algorithm to find the best implementation and to drive the selection of dynamic instructions. In particular, we wanted to:

- determine the best performance for different implementations of the `max*` operator;
- determine the minimum bit-widths for input, output and intermediate data, that still yield satisfactory results;
- determine the best threshold to stop the iterative process and declare convergence.

As mentioned above, we selected the piece-wise linear approximation of the `max*` operator. Despite being the most complex, it gives lower errors and ultimately allows the algorithm to converge faster, thus actually increasing the average performance. The chosen bit-widths, 8 bits for inputs and outputs, and 12 bits for intermediate computations, are sufficient to keep a reasonable precision [3].

After selecting and mapping dynamic instructions, we performed several simulations to determine the performance of the turbo-decoder, comparing it to a similarly optimized pure software implementation on the same architecture, but without the pGA. Table 1 shows the experimental results that we obtained on a 40-bit message, when different combinations of dynamic instructions are used. Column **Speed-up** indicates the improvement in performance over the entire algorithm, including also portions that were not implemented with dynamic instructions. **Exec. cycles** shows the total number of clock cycles needed to complete the decoding, and **Saved cycles** is the difference with respect to the original pure software implementation.

We first simulated the algorithm without any dynamic instructions, and then we re-simulated it using only one type of dynamic instruction at a time. All the implementations that we compare used the MAC and looping instructions and the 2-wide VLIW execution that are available on

the XiRisc processor. Therefore, our results only show the effects of dynamic instructions and of the pGA.

Computation of the butterfly operator and of the maximum likelihood ratio proved to be the most expensive functions, and thus benefitted most from hardware acceleration. When all dynamic instructions are used at the same time, the speed-up is a remarkable $11\times$. Although the two instructions of metrics computation and reordering show modest speed-ups when compared with the software-only implementation, their effect on the final result is very important. In fact, while their contribution to the total latency of the algorithm is low at the beginning, they become the bottlenecks when the other more complex operations are moved into hardware. For instance, if reordering were not implemented as a dynamic instruction, the total number of clock cycles would be around 31.000, corresponding to a speed-up of only $5.7\times$.

Simulations have been performed for different message lengths (640 and 5114 bits), and other combinations of dynamic instructions, yielding similar results. If we assume that the processor runs at 100 MHz (the low speed is due mostly to the manually designed pGA), then the final throughput is around 270 kbps for each iteration. This compares very favorably with other implementations that do not use external dedicated hardware accelerators. In [16], a throughput of 262 kbps/iteration was obtained on an Intel Pentium III running at 933 MHz, while [10] describes an implementation on a 200 MHz TMS C6x that reaches 286 kbps/iteration. Finally, [8] reaches 227 kbps/iteration on a 3DSP SP-5 SIMD machine. The UMTS standard requires a throughput of 2 Mbps, so 8 XiRisc processors would be needed running in parallel (on successive blocks of data) in order to satisfy it.

6. Conclusions

In this paper we showed how a reconfigurable processor can be used to dramatically speed up the execution of a highly data and control intensive task, the decoding of turbo-codes. We used a design flow that enables a software designer, who is mostly unaware of hardware design subtleties, to quickly assess the costs and gains due to executing selected pieces of C code as single instructions on an FPGA-based reconfigurable DLX functional unit.

The result is a factor of $11\times$ speed-up on the whole decoding algorithm. It was achieved, while simultaneously optimizing the memory layout, with only about 1 month of work by a software designer, who did not have any previous exposure to hardware design, synthesis, or reconfigurable computing.

References

[1] 3GPP. Technical specification group radio access network; multiplexing and channel coding. Technical Specification TS 25.212 v5.1.0, 2002.

[2] M. Arnold. *Instruction Set Extension for Embedded Processors*. PhD thesis, Delft University of Technology, Mar. 2001. ISBN 90-9014523-0.

[3] T. K. Blankenship and B. Classon. Fixed-point performance of low complexity Turbo Decoding alghoritms. Technical report, Motorola Labs, 2001.

[4] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafdazeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of the Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, 2002.

[5] T. Callahan and J. Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proc. of the Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000.

[6] F. Campi, R. Canegallo, and R. Guerrieri. IP-reusable 32-bit VLIW Risc core. In *Proc. of the European Solid-State Circuits Conference*, 2001.

[7] A. Chass, A. Gubeskys, and G. Kutz. Efficient software implmentation of the max-log-map turbo decoder on the starcore sc140 dsp. In *Proc. of the Intl. Conf. on Signal Processing Applications and Technology*, 2000.

[8] J. Harrison. Implementation of a 3gpp turbo decoder on a programmable dsp core. In *Proc. of the Communications Design Conference*, 2001.

[9] J. Jacob and P. Chow. Memory Interfacing and Instruction Specification for Reconfigurable Processors. In *Proc. Intl. Symp. on FPGAs*, 1999.

[10] W. J.Ebel. Turbo-code implementation on C6x. Technical report, Alexandria Research Institute-Virginia Polytechnic Institute and State University, 1999.

[11] B. Kastrup, A. Bink, and J. Hoogerbrugge. ConCISe: A compiler-driven CPLD-based instruction set accelerator. In *Proc. of the Symposium on Field-Programmable Custom Computing Machines*, 1999.

[12] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA-machine description language for cycle-accurate models of programmable DSP architectures. In *Proc. of the Design Automation Conference*, 1999.

[13] A. Peymandoust, L. Pozzi, P. Ienne, and G. D. Micheli. Automatic instruction set extension and utilization for embedded processors. In *Proc. of the Intl. Conf. on Application-Specific Systems, Architectures, and Processors*, 2003.

[14] R. Razdan and M. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. of the Intl. Symposium on Microarchitecture*, 1994.

[15] A. L. Rosa, L. Lavagno, and C. Passerone. A software development tool chain for a reconfigurable processor. In *Proc. of the Intl. Conf. on Compilers, Architectures and Synthesys for Embedded Systems*, 2001.

[16] M. Valenti and J. Sun. The UMTS turbo code and efficient decoder implementation suitable for software-defined radios. In *Intl. Journal of Wireless Information Networks*, Oct. 2001.

[17] M. Vaya and J. Cavallaro. Viturbo: A reconfigurable architecture for viterbi and turbo decoding. In *Proc. of the Intl. Conf. on Acoustics, Speech and Signal Processing*, 2003.

[18] L. Yanbing, T. Callahan, R. Harr, and U. Kurkure. Hardware-software co-design of embedded reconfigurable architectures. In *Proc. of the Design Automation Conference*, 2000.

[19] Z. Ye, N. Shenoy, and P. Banerjee. A C compiler for a processor with a reconfigurable functional unit. In *Proc. of the Intl. Symposium on Field Programmable Gate Arrays*, 2000.