HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform

Massimo Baleani[‡] Frank Gennari[†] Yunjian Jiang[†] Yatish Patel[†] Robert K. Brayton[†]
Alberto Sangiovanni-Vincentelli [†‡]

[‡]PARADES EEIG
Via San Pantaleo 66, 00186 Rome, Italy
mbaleani@parades.rm.cnr.it

[†]Department of EECS
University of California, Berkeley CA 94720
{gennari, wjiang, yatish, brayton, alberto}@eecs.berkeley.edu

ABSTRACT

This paper studies the use of a reconfigurable architecture platform for embedded control applications aimed at improving real time performance. The hw/sw codesign methodology from POLIS is used. It starts from high-level specifications, optimizes an intermediate model of computation (Extended Finite State Machines) and derives both hardware and software, based on performance constraints. We study a particular architecture platform, which consists of a general purpose processor core, augmented with a reconfigurable function unit and data-path to improve run time performance. A new mapping flow and algorithms to partition hardware and software are proposed to generate implementations that best utilize this architecture. Encouraging preliminary results are shown for automotive electronic control examples.

Keywords

CSoC, hw/sw co-design, code generation

1. INTRODUCTION

Configurable system-on-a-chip (CSoC) architectures are emerging as a promising alternative to both ASIC and general purpose processors (GPP), as witnessed by the number of currently available commercial platforms [25]. ASICs suffer from long design cycles, sky-rocketing NRE costs (SEMATECH estimates a cost of $1M for a 0.15$\mu$ mask set) and poor flexibility, while GPPs do not meet performance requirements for demanding applications. The embedding of reconfigurable hardware (eFPGA) expands the range of problems for which post-fabrication solutions are viable. This eliminates the time and money spent in silicon design, fabrication and manufacturing verification.

The wide adoption of these reconfigurable architectures relies on the availability of appropriate methodologies as well as CAD tools to map designs to future multi-million gate devices quickly and to efficiently exploit their flexibility. Particularly, the hardware/software co-design process must determine which portions of the overall specification should be mapped into the reconfigurable logic and which retained on the processor. We discuss the automation of the entire design flow, from high-level specification to hardware and software implementation for control-oriented applications.

While Estrin's "fixed plus variable structure computer" proposed in the early 60's is likely the first reconfigurable computer, the introduction of FPGAs spurred a wealth of research in reconfigurable FPGA-based systems [13]. In the last decade, FPGA-based platforms have achieved significant speedups for a range of applications including data encryption, DNA sequence matching, automatic target recognition, genetic algorithms, image filtering and network processors. However, with increasing system requirements, embedded control applications, such as automotive control, avionics, robotics and industrial plant control processes, are also experiencing performance bottlenecks on traditional micro-controller platforms. Thus reconfigurable hardware opens up new implementation opportunities in this domain.

We use a homogeneous representation for both hardware and software, based on a network of Extended Finite State Machines (EFSMs). This can be captured using several high-level languages with underlying EFSM semantics, such as ESTEREL [9] or State Transition Diagrams. From this common representation we perform hardware/software partitioning and code generation based on performance profiling.

For system implementation, we selected the reconfigurable VLIW RISC core proposed in [7], featuring a 32-bit MIPS core augmented with an FPGA reconfigurable control unit and data-path, which can be customized to issue special instructions reading from and writing to the same MIPS integer registers. A GCC-based software tool-chain for compilation and performance simulation for this architecture was developed, as presented in [23]. It supports arbitrary user defined instructions to be mapped onto the reconfigurable matrix. The user manually identifies and tags certain computation kernels in the source code to be extracted as single FPGA instructions. The tool-set then provides automated support for compilation, assembly, simulation and profiling of the resulting code on this platform.

Starting from the EFSM representation, we provide methods to explore different hw/sw trade-offs based on the profiling of the source

code with the GCC-based tool chain. Thus, we *automatically* derive the C code with critical kernels tagged for FPGA implementation. Our methodology provides a totally automated flow from high-level specifications such as ESTEREL, down to performance-optimized machine code for the reconfigurable target architecture.

The paper is organized as follows. Section 2 reviews some related work in reconfigurable computing and code generation. Section 3 gives our design flow, with details on the model of computation, architecture selection and performance evaluation methods. Section 4 details our method of hw/sw partitioning and code generation. Section 5 gives experimental results followed by conclusions and future work in Section 6.

2. RELATED WORK

A number of research efforts have studied the development strategies and CAD tools for reconfigurable platforms. A popular approach is to produce a unified development environment, and provide a single language that can be effectively mapped to either hardware or software. PRISM [1] accepts generic C code and generates FPGA configurations in a semi-automatic fashion. PRISM analyzed C code to identify C functions that could be implemented with combinational logic. A similar approach to instruction-set augmentation for general purpose computing was proposed for Prisc [22] which considers a finer granularity, i.e. any grouping of instructions instead of entire C functions, for hardware synthesis onto hardware programmable functional units. The GARP architecture and compiler [6] were designed to accelerate loops of general purpose ANSI C programs employing a reconfigurable array as a co-processor. A hw/sw partitioning flow called Nimble was also proposed in [19] for this architecture, which starts from C code and explores the partitioning space for the computation intensive loop kernels. Chameleon Systems reports co-compilation techniques for its CS2000 series [27]. One promising area of research relies on existing compilation techniques targeting VLIW architectures to study the benefits of various functional units and interconnect structures. Adaptive Explicitly Parallel Instruction Computing [26] and Dynamically Variable Instruction Set Architecture [20] represent notable works in this research area.

Our approach starts from a high level abstraction using EFSMs as the formal model to capture system specifications. Any textual or graphical language with underlying EFSM semantics (at present ESTEREL is used as a specification language) can be employed. In the POLIS framework [2] each EFSM is represented by a single state transition table for the control path and a lookup table for the data-path. This does not scale well to designs with large state spaces. Binary Decision Diagrams (BDDs) are used to optimize the state transitions and synthesize the code. This single BDD-based representation cannot be used to perform partitioning between hardware and software at a fine logic computation level.

Other works on Esterel compilation provide efficient ways to translate Esterel programs directly into sequential C code [8, 12, 28]. These techniques do not easily support hw/sw partitioning and cannot be utilized directly for mapping the applications onto a reconfigurable architecture.

For functional evaluation using reconfigurable logic, Sasao *et al.* [24] proposed using a platform that combines FPGAs with sequencing logic to perform logic simulation and showed significant speed up versus a GPP approach. However, only combinational logic functions are considered in their approach.

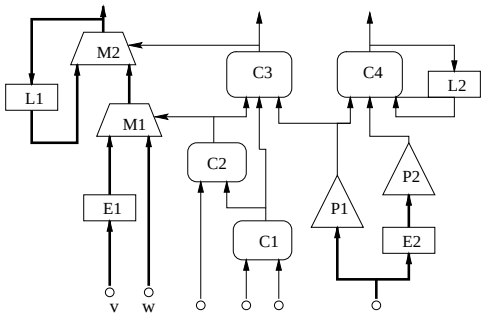

Figure 1: Control-data network

3. DESIGN FLOW

3.1 Model of Computation

We use a network of EFSMs as the fundamental computation model. This is derived from a high-level specification, currently ESTEREL. Technology independent optimization is performed using multi-valued (MV) logic and data-path manipulation.

We use MVSIS [11, 5] for optimization and synthesis. It operates on a control-data network representation. This network has control nodes and data nodes interconnected with two types of variables: MV variables with finite ranges and data variables with unbounded ranges. Each control node is a multi-valued function represented in Multi-valued Decision Diagrams (MDDs) and sum-of-products (SOPs). Data nodes consist of three types: *multiplexers*, *expressions* and *predicates*.

A multiplexer is defined as $f = f(y_c, y_0, \ldots, y_{n-1})$, where y_c is a MV-variable with n values, y_i, ($i \in [0, n-1]$) are data inputs. The output f is assigned to y_i if $y_c = i$. Predicate nodes and expression nodes contain computation of pure data variables, while a predicate outputs an MV variable and an expression outputs a data variable. The data computation is currently modeled as uninterpreted strings, which may be complex computations and function calls. The expressions however must be arithmetic as defined in the semantics of the C language.

Figure 1 shows an example of a control-data network, where bold wires indicate data variables. It consists of four control nodes (C_1-C_4), two multiplexers (M_1, M_2), two predicates (P_1, P_2), two expressions (E_1, E_2) and two latches (L_1, L_2).

A set of technology independent optimization methods are available in MVSIS [5]. These include algorithms extended from binary logic: algebraic decomposition [10], don't care-based simplification [16], elimination, resubstitution; as well as algorithms specifically tuned for multi-valued logic: node pairing and encoding [11].

In the architecture mapping phase, implementation code is automatically generated from this control-data network for the reconfigurable platform. (See Section 4).

3.2 Architectures

Reconfigurable platforms, coupling a programmable logic with a processor core, come in different varieties, differing in processor integration scheme, computing model, and the granularity of the reprogrammable logic. For the computing model, the reconfigurable array may be deployed as an autonomous co/stream processor, dedicated I/O processor, interface glue logic, or as instruction set aug-

mentation. Several computing models may be supported by the same platform, but effectiveness depends on the level of integration between the reconfigurable logic and CPU core. The reconfigurable logic can be placed on an I/O bus [1], can be moved closer to the CPU [15] (akin to a standard floating point unit) or can be integrated into the processor as a reprogrammable functional unit [7, 21, 14].

The integration scheme determines the granularity of the application segments executed on the reconfigurable fabric. Due to the fine granularity of the finite state machine code, we adopted the instruction augmentation computing model and the reconfigurable platform of [7]. Special FPGA operations can be reconfigured and viewed as special instructions in the system ISA. This can be utilized at the C programming level. Since the reconfigurable array is part of the processor's data path, there is minimal communication overhead compared to the coprocessor implementation. A coprocessor platform requires additional cycles to explicitly transfer data to and from the reconfigurable array, thus undermining the performance gain obtained from FPGA instructions.

There are disadvantages of this approach as well. The number of ports on the register file limits the input/output bandwidth of the FPGA array. The control flow of the data path also requires the FPGA array be executed synchronously with the pipeline design. These requirements dictate that only small blocks with few inputs and outputs can be implemented on the FPGA array. Fortunately, this works well in the chosen application space, in that EFSMs contain nodes that have only a few inputs and outputs and perform simple calculations.

3.3 Performance Evaluation

We use the GCC-based performance evaluation tool-chain developed for the target architecture [23]. Given the C code of the target application, the user can tag blocks of C code to be implemented on the reconfigurable array, using a `pragma` directive:

`#pragma` *instr_name opcode delay nout nin outs ins*

where *instr_name* is the mnemonic name of the FPGA instruction; *opcode* is the instruction code used in hardware simulation, which is not relevant in our approach; *delay* is the latency in clock cycles of this instruction; *nouts* and *nins* are the numbers of outputs and inputs; *outs* and *ins* are the lists of output and input variables. The code that follows is interpreted as a C simulation model, which is then ended with another `pragma` directive:

```
#pragma end
```

After the tags are added to the C code, the simulator evaluates the code using the cycle counts specified by the user for the tagged blocks. The profiler returns the number of cycles used to execute each line of code.

Our goal is to automatically partition hardware and software and generate the tags for FPGA instructions, which implements the hardware partition. There are two limitations for an FPGA instruction: (a) it must have no more than 3 inputs and 2 outputs, due to instruction encoding and register file limitations; (b) there is a limited pool of LUTs for FPGA instructions. These are taken into account in the partition algorithm in the next section. There is also a challenge of accurately estimating the cycle count of the FPGA instructions. The most precise method compiles the perspective FPGA instructions into LUTs and calculates the longest path delay versus the clock cycle time of the processor pipeline. In the experiments we use heuristics for the estimation in order to have a quick performance evaluation.

4. CODE GENERATION

The code generation problem is, given a multi-level control-data network, generate efficient code that consists of two portions: software blocks to be executed on the processor and tagged hardware blocks to be implemented as FPGA instructions. The objective is to maximize the overall performance while satisfying appropriate resource constraints, such as RAM and ROM usage and FPGA size.

We solve this problem in two steps: (a) construct maximal regions (clubs) of the network that can be implemented as a single instruction without violating data dependencies, (b) explore the assignment of individual clubs to hardware or software based on performance estimation and profiling. For the final generation of C code, we use an MDD-based code generation approach [17].

4.1 Clubbing

A club is a candidate block of functionality for potential hardware implementation. It is defined as a cluster of nodes, which satisfies the following constraints:

1. It does not contain primary inputs or latches;
2. It consists of either pure control nodes or pure data nodes;
3. Its number of inputs (outputs) does not exceed a predetermined maximum number;
4. It does not introduce combinational cycles among clubs.

We currently do not allow primary inputs to be implemented on the reconfigurable array, because the architecture dictates that all functional inputs be supplied through the register files. However, in some applications it may be worthwhile to consider different architectures that allow this. Latches can be implemented in FPGAs as well, thus the FPGA operations may have states that are kept from instruction to instruction. This involves sequential EFSM partitioning, which is beyond the scope of this paper.

Condition (2) is present because control and data nodes require different sets of logic and data-paths for implementation. Condition (3) is due to instruction encoding and register file limitations. Condition (4) guarantees deterministic and correct functionality.

In [18] Khatri introduced a clustering algorithm, with a similar definition of clubs, for mapping from a logic network to a network of PLAs. Although it does not satisfy our clubbing constraints, our clubbing algorithm, outlined in Figure 2, is based on this.

The network is first optimized and decomposed into small nodes. Routine *Build_levelized_array()* (step 2) then levelizes and sorts the nodes in a depth-first order. In the levelization, the nodes are traversed from inputs to outputs; when a node is added to the array, a recursive function is called on each of the node's fanouts; in the recursion, if all of a node's fanins have previously been added to the array, this node is also added. In essence, a node is inserted to the end of the array immediately after all of its fanins have been

Algorithm [Network Clubbing]:
input: control-data-network N
output: club pool *Result*

```
1   N^1 = Decompose_network (N, M);
2   L = Build_levelized_array (N^1);
3   Foreach node k in L do
4       if check_club(C,k) then
5           C = make_club(C,k);
6           Continue;
7       else
8           Result = Result ∪ C;
9           C = new_club(k);
10      end if
11  end
```

Figure 2: Network Clubbing Algorithm

inserted. Notice that primary inputs and latches are not included in this array.

The main clubbing loop then iterates through each node in this levelized array and checks if the addition of the node to the current club violates any of conditions 1-4 (*check_club()*). If not, the node is added to the club (*make_club()*); otherwise a new club is initiated for this node and the previous club added to the final pool.

4.2 Bit-packing

The synthesis flow supports multi-valued variables in the EFSM model, but in many designs, a majority of the nodes only require a few bits to represent the largest value. Therefore, in order to fully utilize the FPGA instructions with the 3-input and 2-output limitation, we need to bit-pack multiple control variables into a single 32-bit integer allows larger clubs. There is of course overhead cost of assigning (extracting) the proper bits of the 32-bit integer for the inputs (outputs) of the FPGA instructions. However, larger clusters would lead to the elimination of unnecessary intermediate variables, thus saving load/store instructions and memory accesses.

To incorporate bit-packing, the code generation flow described above is modified in two aspects:

1. The clubbing algorithm is modified so that the input/output constraint reflects the number of bits rather than variables. For the selected architecture [7], the input constraint is $32*3$ bits and the output constraint is $32*2$ bits.

2. The code of each club is included with additional temporary variables for the constrained club-inputs and club-outputs. New code is added to align the original input bits into the temporary club input variables before the FPGA instruction, and to extract output bits from the temporary club output variables after the FPGA instruction.

Bit-packing allows each club to have up to 96 binary variables as input, and 64 binary variables as output. The IO constraint of the FPGA instructions for control is then only limited by the data dependencies between control nodes and data nodes. However, bit packing and unpacking are very expensive operations on conventional processors and create overhead on the software partition. Yet they are extremely cheap on the FPGA processor. This again illustrates the trade-off between computation and communication.

Table 1: Benchmark examples

Example	strl	PI	PO	node	MUX	PRED	EXP	latch
Dance	243	1	14	299	73	13	41	38
Display	155	4	6	148	51	3	43	14
Driver	775	85	49	541	118	41	58	71

4.3 Partitioning

We explore different hardware/software trade-offs based on performance profiles of each potential club. This is done in two steps:

1. Obtain the performance profile and FPGA implementation cost (reconfigurable array LUT count) for each club.

2. Find the best hw/sw partition that maximizes performance gain and satisfies the FPGA size constraints.

For the first step, we generate C code for all clubs with no FPGA instructions; we run through the GCC-based tool-chain and obtain an average cycle count for each line of the C code; the cycle count for a club is then the summation of the cycle counts for all its source lines.

A simple algorithm based on the number of input and output bits is used to estimate the number of LUTs required if the club is implemented in hardware. Node patterns that commonly appeared were synthesized using Xilinx logic synthesis tools to determine the LUT counts. Any unidentifiable node types such as function calls were estimated manually.

The second step can be formulated as a Boolean programming problem. For potential FPGA clubs $\{C_1, C_2, \ldots, C_n\}$, let $\{T_1, T_2, \ldots, T_n\}$ be their cycle count from the performance profiling, and let $\{S_1, S_2, \ldots, S_n\}$ be their estimated LUT count for the FPGA implementation. Create a Boolean variable x_i for each club C_i; let $x_i = 1$ represent C_i being implemented in hardware and $x_i = 0$ represent software implementation. The goal is to find the Boolean assignment of the vector $\{x_1, x_2, \ldots, x_n\}$, such that:

$$\begin{aligned} \textit{Maximize}: &\quad \textstyle\sum_1^n x_i \cdot T_i \\ \textit{Constraint}: &\quad \textstyle\sum_1^n x_i \cdot S_i \leq SIZE \end{aligned}$$

In the actual experiments done so far, due to our primitive estimation of cycle count and LUT size, we chose a greedy assignment approach: the clubs are sorted in descending order of their profiled cycle counts; assign, in order, each as FPGA instruction until the FPGA size limit is reached.

After hw/sw partitioning, each chosen FPGA instruction needs an estimated cycle count for performance simulation. The heuristics used assume that small nodes will take 1 cycle. For commonly used clubs, we again run a Xilinx synthesis tool and estimate the resulting FPGA hardware latency. For more complex control clubs, we estimate based on the number of input and output values. Other types of clubs that involve function calls and data computation are estimated manually.

5. EXPERIMENTS

The benchmark set includes a multi-injection driver for engine control systems [3], an automotive cross-display module and a Lego Mindstorms Acrobot [4]. All contain both control and data portions

Table 2: Results from performance simulation

	Clubs	Est. LUTs	Cycles
Shock dance			
No FPGA	0	0	923,384
All Control	75	128	536,729
All Data	76	2596	736,794
Mix	151	2724	406,695
Cross display			
No FPGA	0	0	452,268
All Control	18	18	346,882
All Data	58	2100	306,641
Mix	76	2118	218,146
Injection driver			
No FPGA	0	0	5,263,012
All Control	122	542	4,682,820
All Data	99	3304	2,169,596
Mix	221	3846	1,530,823

in the FSM. They are converted into an intermediate control-data network representation. The corresponding statistics are shown in Table 1, with `strl` for the number of lines in the Esterel source code, `PI` (`PO`) for the number of inputs (outputs), `node` for the total number of control and data nodes, and also the number for each type of data node.

Table 2 show the average number of cycles required to execute the FSM for a set of 500 input vectors, depending on which set of clubs was implemented on the reconfigurable array. Four variations were tried. The first implemented all of the C code on the processor and no FPGA instructions. The second targeted all control nodes to the reconfigurable array; the third targeted all of the data nodes. The fourth targeted all possible clubs to the array while still satisfying the FPGA size limitation.

Generally the results show that performance benefits the most from implementing data nodes on the reconfigurable array. However data nodes require a significantly higher number of LUTs. Further, implementation of complex data computation like multiplication in FPGA may not be as efficient as the arithmetic unit in the GPP core. Further study is needed for more intricate trade-off between FPGA hardware and traditional arithmetic data-path in GPPs.

Due the relatively small size of our benchmark suite, the FPGA size of the target architecture is large enough to implement all potential clubs in these examples. We experimented with smaller FPGA limitations, and the mapping algorithm succeeded in producing significant performance improvement.

The results of adding the bit-packing feature is shown in Table 3. Since only control clubs take advantage of bit-packing, we present results for implementing only control clubs. As a result of relaxed IO constraints, the clubs become larger and more consolidated, (from 122 to 50 for the injection driver example) which leads to more efficient FPGA implementation. Since a large amount of the cycle count lies in data computation, the improvement of the total cycle count due to bit-packing is limited. Yet, it does increase the performance of the control portion, from 20% to 40%.

6. CONCLUSION AND FUTURE WORK

We introduced an automated hw/sw partitioning and code generation flow for control applications on a reconfigurable platform. This flow uses the EFSM model and derives hw/sw implementation automatically from high-level synchronous specifications. Compared to the GPP approach, the reconfigurable array functional unit can

Table 3: Results from bit-packing control clubs

Example	Clubs	Est. LUTs	Cycles
Dance	38	102	366,332
Display	8	12	302,543
Driver	50	442	4,428,876

significantly improve the run time of the applications we targeted. The data computation seems to benefit most from the use of the reconfigurable array. However, more complicated data nodes require significantly larger area on the reconfigurable array than the simpler control nodes.

Using this automated synthesis flow, embedded applications can be smoothly mapped from high-level language specifications like Esterel, down to hw/sw implementation on the reconfigurable architecture platform. However, our approach is limited to the applications that can be modeled and programmed by extended finite state machines.

Our algorithm for clubbing uses a relatively simplistic estimation of the area and delay required for the FPGA implementation. As a result, the hw/sw partitioning may not be optimal, due to inaccurate delay and size information. Future work includes incrementally building a knowledge-base that keeps FPGA implementation costs for sub-components.

The mapping onto other standardized system-on-chip architectures with different FPGA embedding and data-path configuration needs to be explored. Automatic synthesis of the communication scheme between hardware and software, which explores different on-chip communication options, seems to be a promising research direction.

Future work also includes sequential EFSM partitioning, which produces sequential FPGA operations that may have internal states. At a higher-level, decomposing an EFSM into smaller machines with synchronous or asynchronous composition remains an open problem.

7. ACKNOWLEDGMENTS

The authors would like to acknowledge Luciano Lavagno for his insightful discussion and comments. We also thank the anonymous reviewers' comments and suggestion. We are grateful for the support of the SRC under contract 683.004 and the California Micro program and industrial sponsors, Fujitsu, Cadence, and Synplicity.

8. REFERENCES

[1] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, 1993.

[2] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B.Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.* Kluwer Academic Press, 1997.

[3] M. Baleani, M. Conti, A. Ferrari, and A. Sangiovanni-Vincentelli. HW/SW co-design of a multiple injection driver automotive subsystem using a configurable system-on-chip. In *Proc. of the Conf. on Design Automation & Test in Europe*, Mar. 2002.

[4] G. Berry. A dancing lego mindstorms acrobot programmed in esterel. *Technical Report*, 2000.

[5] R. K. Brayton and et al. MVSIS. `http://www-cad.eecs.berkeley.edu/mvsis`.

[6] T. Callahan, J. Hauser, and J. Wawrzynek. The GARP architecture and C compiler. *IEEE Computers.*, 2000.

[7] F. Campi, R. Canegallo, and R. Guerrieri. IP-reusable 32-bit VLIW RISC core. In *European Solid-State Circuits Conference*, Sept. 2001.

[8] S. Edwards. Compiling esterel into sequential code. In *Proc. of the Design Automation Conf.*, June 2000.

[9] The ESTEREL language. [On-line] http://www.esterel.org.

[10] M. Gao and R. K. Brayton. Semi-algebraic methods for multi-valued logic. In *Proc. of the Intl. Workshop on Logic Synthesis*, May. 2000.

[11] M. Gao, J. Jiang, Y. Jiang, Y. Li, S. Singha, and R. K. Brayton. MVSIS. In *Proc. of the Intl. Workshop on Logic Synthesis*, May. 2001.

[12] O. Hainque, L. Pautet, Y. L. Biannic, and E. Nassor. Cronos: a separate compilation toolset for modular esterel applications. *Formal Methods*, 1999.

[13] R. Hartenstein. A decade of reconfigurable computing: A visionary perspective. In *Proc. of the Conf. on Design Automation & Test in Europe*, March 2000.

[14] S. Hauck, T. Fry, M. Hosler, and J. Kao. The Chimaera reconfigurable functional unit. In *the IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1997.

[15] J. R. Hauser and J. Wawrzynek. GARP: A mips processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machine*, Apr. 1997.

[16] Y. Jiang and R. K. Brayton. Don't cares and multi-valued logic network minimization. In *Proc. of the Intl. Conf. on Computer-Aided Design*, Nov. 2000.

[17] Y. Jiang and R. K. Brayton. Logic optimization and code generation for embedded control applications. In *Proc. of the Intl. Symposium on Hardware/Software Co-Design*, Apr. 2001.

[18] S. P. Khatri, R. K. Brayton, and A. Sangiovanni-Vincentelli. Cross-talk immune VLSI design using a network of PLAs embedded in a regular layout fabric. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 412–18, Nov. 2000.

[19] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *Proc. of the Design Automation Conf.*, June 2000.

[20] Proceler. Soft instruction set architectures for embedded computing. [On-line] http://www.proceler.com.

[21] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, May 1994.

[22] R. Razdan, K. Brace, and M. D. Smith. PRISC software acceleration techniques. In *Proceedings of the International Conference on Computer Design*, pages 145–149, October 1994.

[23] 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, Architecture and Synthesis for Embedded Systems*, Nov. 2001.

[24] T. Sasao, M. Matsuura, and Y. Iguchi. A cascade realization of multiple-output function for reconfigurable hardware. In *Proc. of the Intl. Workshop on Logic Synthesis*, May. 2001.

[25] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. A quick safari in the reconfiguration jungle. In *Proc. of the Design Automation Conf.*, June 2001.

[26] S. Talla. *Adaptive Explicitly Parallel Instruction Computing*. PhD thesis, New York University, 2000.

[27] X. Tang, M. Aalsma, and R. Jou. A compiler directed approach to hiding configuration latency in Chameleon processors. In *Proceedings of the 10th International Conference on Field Programmable Logic and Applications (FPL)*, 2000.

[28] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *Proc. of the Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, Nov. 2000.