A Taxonomy of Custom Computing Machines

Mihai Sima[†‡], Stamatis Vassiliadis[†], Sorin Cotofana[†], Jos van Eijndhoven[‡], Kees Vissers[‡§]

[†]Delft University of Technology – Dept. of Electrical Engineering, Delft, The Netherlands
[‡]Philips Research – Dept. of Information and Software Technology, Eindhoven, The Netherlands
[§] TriMedia Technologies, Inc., Sunnyvale, California, U.S.A.

Phone: +31-(0)40-274-2593 E-mail: mihai@dutepp0.et.tudelft.nl

Abstract— **The need for providing a hardware platform which can be customized on a per-application basis under software control has established the *Reconfigurable Computing* ($\mathcal{RC}$) as a new computing paradigm. A machine employing the $\mathcal{RC}$ paradigm is referred to as a *Custom Computing Machine* (CCM). Till now, the CCMs have been classified according to implementation criteria. As the previous classifications do not seize well the meaning of the $\mathcal{RC}$ paradigm, we propose a classification of the CCMs according to architectural criteria. In order to analyze the phenomena inside CCMs, we introduce a new formalism based on microcode, in which any FPGA-dedicated instruction is executed as a microprogrammed sequence with two basic stages:** `SET CONFIGURATION`**, and** `EXECUTE CONFIGURATION`**. Based on the** `SET/EXECUTE` **formalism, we then propose an architectural-based taxonomy of CCMs.**

Keywords— **Reconfigurable Computing, Custom Computing Machines, Field-Programmable Gate Arrays, microcode, CCMs taxonomy.**

I. Introduction

THE ability of providing a hardware platform which can be transformed under software control has established a new computing paradigm, referred to as *Reconfigurable Computing* ($\mathcal{RC}$) [1], [2], [3], as an emerging technology for more than ten years. According to this paradigm, the main idea in improving the performance of a computing machine is to define custom computing resources on a per-application basis, and to dynamically configure them onto a *Field Programmable Gate Array* (FPGA) [4]. Consequently, virtually infinite hardware can be emulated.

As a general view, a computing machine working under the new $\mathcal{RC}$ paradigm typically includes a *General-Purpose Processor* (GPP) augmented with an FPGA. The basic idea is to exploit both the GPP flexibility to achieve medium performance for a large class of applications, and FPGA capability to implement application-specific computations. Such a hybrid is referred to as a *Custom Computing Machine* (CCM) [5]. The synergism of GPP and FPGA can achieve orders of magnitude improvements in performance over a GPP alone, while preserving the flexibility of the programmed machines over *Application-Specific Integrated Circuits* (ASIC) in implementing a large number of applications. However, the CCM performance in terms of speed and power may still be of orders of magnitude lower than the performance of an ASIC.

Various CCMs have been proposed in the last decade. Former attempts in classifying CCMs used implementation criteria [6], [7], [8], [9], [10]. As the user observes only the architecture of a computing machine [11], the previous classifications do not seize well the implications of the new $\mathcal{RC}$ paradigm as perceived by the user. Therefore, we propose to classify the CCMs according to architectural criteria. In order to analyze the phenomena inside CCMs, we introduce a new formalism based on microcode, in which the execution of an FPGA-dedicated instruction is performed as a microprogrammed sequence with two basic stages: `SET CONFIGURATION`, and `EXECUTE CONFIGURATION`. Based on the `SET/EXECUTE` formalism, we propose an architectural-based taxonomy of CCMs.

The paper is organized as follows. For background purpose, we present the most important issues related to microcode in Section 2, and the basic concepts concerning SRAM-based FPGAs in Section 3. Section 4 introduces a formalism by which the CCM architectures can be analyzed from the microcode point of view, and Section 5 presents the architectural-based taxonomy of CCMs. Section 6 concludes this paper.

II. The Microcode Concept

Figure 1 depicts the basic microprogrammed computer as it is described in [12]. For such a computer, a microprogram in *Control Store* (CS) is associated with each incoming instruction. This microprogram is to be executed on the *Microprogrammed Loop* under the control of the *Sequencer*, as follows:

1. The sequencer maps the incoming instruction code into a control store address, and stores this address into the *Control Store Address Register* (CSAR).

2. The microinstruction addressed by CSAR is read from CS into the *MicroInstruction Register* (MIR).
3. The microoperations specified by the microinstruction in MIR are decoded, and the control signals are subsequently generated.
4. The computing resources perform the computation according to such control signals.
5. The sequencer uses the status information generated by the computing facilities as well as some information originating from MIR to prepare the address of the next microinstruction. This address is then stored into CSAR.
6. If an *end-of-operation* microinstruction is detected, a jump is executed to a microsubroutine which implements the instruction fetch procedure. At the end of the fetch microsubroutine, the new incoming instruction initiates a new cycle of the microprogrammed loop.

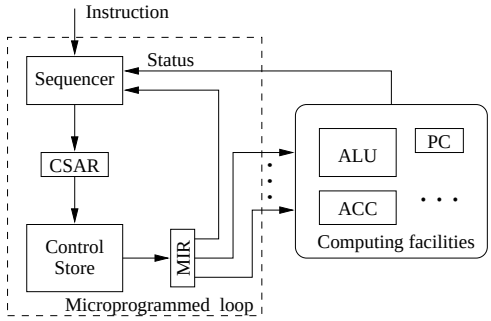

Fig. 1. The basic microprogrammed computer (ALU – Arithmetic and Logic Unit, ACC – Accumulator, PC – Program Counter) – adapted from [12]

The microinstructions may be classified by the number of controlled resources. Given a hardware implementation which provides a number of computing resources (facilities), the amount of explicitly controlled resources during the same time unit (cycle) determines the verticality or horizontality of the microcode as follows:

- A microinstruction which controls multiple resources in one cycle is *horizontal.* In the extreme case, all the resources of the data path are controlled, as it is depicted in Figure 2 – a.
- A microinstruction which controls a single resource is *vertical.* This situation is pictured in Figure 2 – b.

Let us assume we have a *Computing Machine* (CM) and its instruction set. An *implementation* of the CM can be formalized by means of the doublet:

$$\mathrm{CM} = \{\mu\mathrm{P}\,,\, \mathcal{R}\} \tag{1}$$

where μP is the microprogram which includes all the microroutines for implementing the instruction set, and $\mathcal{R}$ is the set of N *computing (micro-)resources* or *facilities*

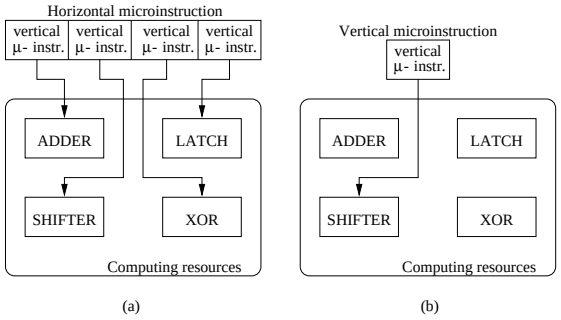

Fig. 2. A classification of microinstructions: (a) – horizontal microinstruction; (b) – vertical microinstruction

which are controlled by the microinstructions in the microprogram:

$$\mathcal{R} = \{r_1\,,\, r_2\,,\, \ldots\,,\, r_N\} \tag{2}$$

Let us assume the computing resources are hardwired. If the microcode[1] is exposed to the user, i.e., the instruction set is composed of microinstructions, there is no way to adapt the architecture to application but by custom-redesigning the computing facilities set, $\mathcal{R}$. When the microcode is not exposed to the user, i.e., a microroutine is associated with each instruction, then the architecture can be adapted by rewriting the microprogram μP.

As the architecture of the vertical microinstructions associated with hardwired computing facilities is fixed, the adaptation procedure by rewriting the microprogram is quite limited. In this way, an instruction is created by threading the operations of fixed (i.e., inflexible) computing facilities rather than generating a full-custom one.

If the resources themselves are microcoded, the formalism recursively propagates to lower levels. Therefore, the implementation of each resource can be viewed as a doublet composed of a *nanoprogram* (nP) and a *nano-resource set* ($n\mathcal{R}$):

$$r_i = \{nP\,,\, n\mathcal{R}\}\,, \quad i = 1, 2, \ldots, N \tag{3}$$

Now it is the rewriting of the nanocode which is limited by the fixed set of nano-resources.

The presence of the reconfigurable hardware opens up new ways to adapt the architecture. Assuming the resources are implemented on a programmable array, adapting the resources to the application is entire flexible and can be performed on-line. In this situation, the resource set $\mathcal{R}$ metamorphoses into a new one, $\mathcal{R}^*$:

$$\mathcal{R} \longrightarrow \mathcal{R}^* = \{r_1^*\,,\, r_2^*\,,\, \ldots\,,\, r_M^*\}, \tag{4}$$

[1] In this presentation, by *microcode* we will refer to both microinstructions and microprogram. The meaning of the microcode will become obvious from the context.

and so does the set of associated vertical microinstructions. It is obvious that writing new microprograms with application-tuned microinstructions is more effective than with fixed microinstructions.

At this point, we want to stress out that the microcode is a *recursive formalism*. The *micro* and *nano* prefixes should be used against an *implementation reference level*[2] (IRL). Once such a level is set, the operations performed at this level are specified by *instructions*, and are under the explicit control of the *user*. Therefore, the operations below this level are specified by *microinstructions*, those on the subsequent level are specified by *nanoinstructions*, and so on.

III. FPGA Terminology and Concept

A device which can be configured *in the field* by the end user is called a *Field-Programmable Device* (FPD) [4]. In a general view, a FPD is composed of two constituents: *Raw Hardware* (Processing Elements (PE) and interconnecting resources) and *Configuration Memory*. The function performed by the FPD is defined by the information stored into configuration memory.

The FPD architectures can be classified in two major classes: *Programmable Logic Devices* (PLD) and *Field-Programmable Gate Arrays* (FPGA). Details on each class can be found for example in [4]. For now, we want to mention only that although both PLD and FPGA devices can be used to implement digital logic circuits, we will pre-eminently above all use the term of FPGA hereafter to refer to a programmable device. The higher logic capacity of FPGAs and the attempts to augment FPGAs with PLD-like programmable logic in order to make use of both FPGA and PLD characteristics, support our choice for this terminology.

Some FPGAs can be configured only once, e.g., by burning fuses. Other FPGAs can be reconfigured any number of times, as their configuration is stored in SRAM.

Initially considered as a weakness due to the volatility of programming data, *in-system* reprogramming capabilities of SRAM-based FPGAs led to the new $\mathcal{RC}$ paradigm. This paradigm assumes that in-system FPGA reconfiguration is performed under software control. In this way, the user can instantly create application-geared computing facilities.

With the new $\mathcal{RC}$ paradigm, complex instructions can be implemented on-the-fly. In this way, applications which are very computational demanding can be efficiently executed. Also, a sequential reconfiguration strategy can be used in order to deal with insufficient configurable hardware. In this way, by swapping the configurations in and out of the FPGA upon demand and in real-time, only the necessary hardware is instantiated at any given time. Consequently, with a limited hardware resource, virtually infinite hardware is emulated.

Unfortunately, a huge reconfiguration data rate is needed to achieve a run-time reconfiguration. For example, following the methodology described in [13], we estimate that for an array of 100 4-input LUT-based processing elements interconnected by a fixed network for the sake of simplicity, a reconfiguration data rate of $16\ \mathrm{Gbit/s}$ is needed if the configuration is to be changed on every cycle with a frequency of 10 MHz.

It is this reconfiguration data rate which constitutes the major drawback of the $\mathcal{RC}$ paradigm. The attempts to overcome this drawback led to different reconfiguration patterns which, in turn, induced the name of the major FPGA architectural classes: *Single-Context*, *Multiple-Context*, *Partial Reconfigurable*.

A *single-context* device typically requires a global reconfiguration even for changing 1 bit of its configuration information. As a single-context FPGA does not make use of any technique to increase the reconfiguration data bandwidth, it can be reconfigured only at a very low rate.

A *multiple-context* FPGA stores multiple layers of configuration information referred to as *contexts*, only one of them being active at a time. An extremely fast context switch is possible, at the expense of a huge transient power consumption. As each layer of the configuration memory can be independently written, the circuit defined by the active configuration layer may continue its execution, while the non-active configuration layers are being reconfigured.

In a *partially reconfigurable* device, means for selective reconfiguration of the array are provided. The portions of the array which are not being configured may continue execution. Consequently, the computation and reconfiguration can be overlapped.

A discussion on choosing the appropriate FPGA architecture is beyond the goal of this paper. More information concerning this problem can be found for example in [14].

IV. FPGA to Microcode Mapping

In this section, we will introduce a formalism by which a CCM architecture can be analyzed from the microcode point of view. This formalism originates in the observation that every instruction of a CCM can be mapped into a microprogram.

As we already mentioned, by making use of the FPGA capability to change its functionality in pursuance of a reconfiguring process, adapting both the functionality of

[2] If it will not be specified explicitly, the IRL will be considered as being the level defined by the instruction set. For example, although the microcode is exposed to the user in the RISC machines, the RISC operations are specified by *instructions*, rather than by microinstructions.

computing facilities and *microprogram in the control store* to the application characteristics are possible with the new $\mathcal{RC}$ paradigm. As the information stored in FPGA's configuration memory determines the functionality of the raw hardware, the dynamic implementation of an instruction on FPGAs can be formalized by means of a microcoded structure. In such a structure, the micro-programmed loop and the FPGA may have a Δ arrangement, as depicted in Figure 3. Both FPGA constituents – *Configuration Memory*, and *Raw Hardware* – are regarded as controlled resources in the proposed formalism. Each of the previously mentioned resources is given a special class of microinstructions: `SET` for configuration memory, and `EXECUTE` for the circuits configured on raw hardware. The `SET` microinstruction initiates the reconfiguration of the raw hardware, and the `EXECUTE` microinstruction launches the operations performed by the circuits configured on the raw hardware.

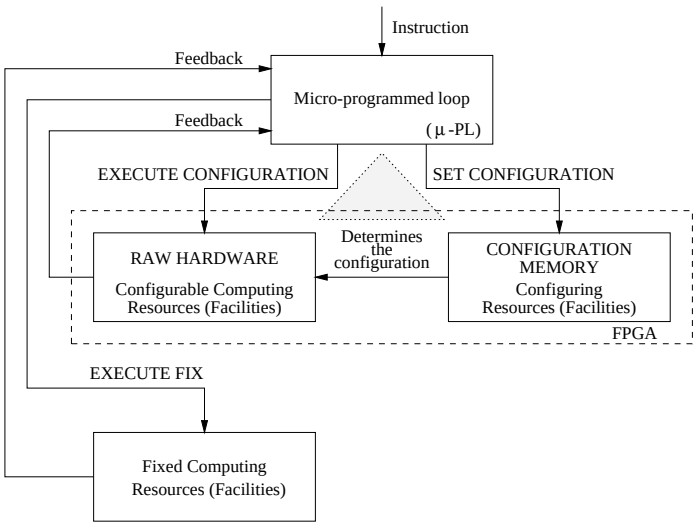

Fig. 3. The microcode concept applied to CCMs. The Δ arrangement

In this way, the execution of an FPGA-dedicated instruction is performed as a microprogrammed sequence with two basic stages: a `SET CONFIGURATION` stage, and an `EXECUTE CONFIGURATION`[3] stage. It is the `SET/EXECUTE` formalism we will use in building the taxonomy of CCMs.

It is worth stressing out that only `EXECUTE FIX` microinstructions can be associated with fixed computing facilities, because such facilities cannot be reconfigured. Also, assuming that a multiple-context FPGA is used, activating an idle context is performed by an `ACTIVATE CONFIGURATION` microinstruction, which is actually a flavor of the `SET CONFIGURATION` microinstruction.

Although we regard the *FPGA configuration memory* as a controlled resource in the proposed formalism, such a resource is not a computing facility in the strict sense. It has a dual status: controlled facility for the micro-programmed loop and controlling unit for the raw hardware. For this reason we will refer to it as a *Configuring Resource*.

Given that a CCM includes both computing and configuring facilities, the statement regarding the verticality or horizontality of the microcode as defined in Section II needs to be adjusted. For a CCM hardware implementation which provides a number of *computing and controlling facilities*, the amount of explicitly controlled *computing or controlling facilities* during the same time unit (cycle) determines the verticality or horizontality of the microcode. Therefore, any of the `SET CONFIGURATION`, `EXECUTE CONFIGURATION`, and `EXECUTE FIX` microinstructions can be either vertical or horizontal, and may participate in a horizontal microinstruction.

Let us set the IRL as being the level of instructions in Figure 3. In the particular case when the microcode is not exposed to the user, an explicit `SET` instruction is not available. Consequently, the system performs by itself the management of the active configuration, i.e., without an explicit control provided by user. In this case, the user "sees" only the FPGA-dedicated instruction which can be regarded as an `EXECUTE CONFIGURATION` microinstruction reflected to the instruction level[4].

Otherwise, when the microcode is exposed to the user, an explicit `SET` instruction is available, and the management of the active configuration becomes the responsibility of the user.

V. A Proposed Taxonomy of CCMs

Before introducing our taxonomy, we would like to overview the previous work in CCM classification.

In [6] two parameters for classifying CCMs are used: *Reconfigurable Processing Unit (RPU) size* (*small* or *large*) and *availability of dedicated RPU local memory*. Consequently, CCMs are divided into four classes. As what exactly means *small* and what exactly means *large* is subject to the complexity of the algorithms being implemented, the differences between classes are rather fuzzy. Also, providing dedicated RPU memory is an issue which belongs to *implementation level* of a machine; consequently, the implications to the *architectural level*, if any, are not clear.

[3] As the differences between software and hardware become fuzzy in the new $\mathcal{RC}$ paradigm, the `EXECUTE CONFIGURATION` terminology can be considered acceptable.

[4] The `EXECUTE FIX` microinstruction is always reflected to the user.

The PE *granularity*, *RPU integration level* with a host processor, and the *reconfigurability of the external interconnection network* are used as classification criteria in [7]. According to the first criterion, the CCMs are classified as *fine-*, *medium-*, and *coarse-grain* PE based systems. The second criterion divides the machines into *dynamic* systems that are not controlled by any external device, *closely-coupled static* systems in which the RPUs are coupled on the processor's datapath, and *loosely-coupled static* systems that have RPUs attached to the host as a coprocessor. According to the last criterion, the CCMs can have a *reconfigurable* or *fixed* interconnection network. This classification is based on the architecture of the programmable arrays themselves and CCM implementation issues rather than CCM architectural criteria.

In order to classify the CCMs, the *loosely coupling* versus *tightly coupling* criterion is used by other members of the CCM community, also [8], [9], [10]. In the loosely coupling embodiment, the RPU is connected via a bus to, and operates asynchronously with the host processor. In the tightly coupling embodiment, the RPU is used as a *functional unit*. This model eliminates the problem of synchronization and reduces the communication latency between host and RPU.

We want to stress out that all the above classifications are build using CCM *implementation* criteria. As the user observes only the architecture of a computing machine, classifying the CCMs according to architectural criteria is more appropriate.

This is why we propose to classify the CCMs according to architectural criteria. As the CCMs are microcoded machines, the criteria we use are:

1. The verticality/horizontality of the microcode.
2. The explicit availability of a `SET` instruction.

According to these criteria, the most well known CCMs can be classified as follows:

1. Vertical microcoded CCMs

(a) With explicit `SET` instruction: PRISM [15], PRISM-II/RASC [16], [17], RISA' [18], RISA" [18], MIPS-derived host + REMARC [19], Garp [20], OneChip-98" [10], URISC [21], Nano-Processor (load-time reconfiguration) [22], Gilson's CCM [23], CCSimP (load-time reconfiguration) [24], Xputer/rALU (load-time reconfiguration) [25].

(b) Without explicit `SET` instruction: PRISC [26], OneChip [9], ConCISe [27], OneChip-98' [10], DISC [28], Multiple-RISA [29], Chimaera [30].

(c) Not obvious information about an explicit `SET` instruction: Virtual Computer [31], Functional Memory [32], NAPA [33].

2. Horizontal microcoded CCMs

(a) With explicit `SET` instruction: CoMPARE [34], Alippi's VLIW [35], RISA"' [18], VEGA [36], RaPiD (load-time reconfiguration) [37], Colt [38], rDPA [39].

(b) Without explicit `SET` instruction: PipeRench [40].

(c) Not obvious information about an explicit `SET` instruction: Spyder [41].

We would like to mention that applying the classification criteria on OneChip-98 machine introduced in [10], we determined that an explicit `SET` instruction was not provided to the user in one embodiment of OneChip-98, while such an instruction was provided to the user in another embodiment. It seems that two architectures were claimed in the same paper. We referred to them as OneChip-98' and OneChip-98".

The same ambiguous way to propose multiple architectures under the same name is employed in [18]. For the Reconfigurable Instruction Set Accelerator (RISA), our taxonomy provides three entries (RISA', RISA", RISA"').

The taxonomy we proposed is architectural consistent, and can be easily extended to embed other criteria. For a complete taxonomy of CCMs we refer the user to [42].

VI. Conclusions

In this paper we proposed a classification of the CCMs according to architectural criteria. Two classification criteria were extracted from a formalism based on microcode. In terms of the first criterion, the CCMs were classified in vertical or horizontal microcoded machines. In terms of the second criterion, the CCMs were classified in machines with or without an explicit `SET` instruction. As future work, we would like to generate new criteria highlighting the structure of the set of the computing resources.

References

[1] W.H. Mangione-Smith and B.L. Hutchings, "Reconfigurable Architectures: The Road Ahead," in *Proc. Reconfigurable Architectures Workshop*, Geneva, Switzerland, 1997, pp. 81–96.

[2] J. Villasenor and W.H. Mangione-Smith, "Configurable Computing," *Scientific American*, pp. 55–59, 1997, http://www.sciam.com/0697issue/0697villasenor.html.

[3] W.H. Mangione-Smith *et al.*, "Seeking Solutions in Configurable Computing," *IEEE Computer*, vol. 30, no. 12, pp. 38–43, 1997.

[4] Stephen Brown and Jonathan Rose, "Architecture of FPGAs and CPLDs: A Tutorial," *IEEE Tran. on Design and Test of Computers*, vol. 13, no. 2, pp. 42–57, 1996.

[5] D.A. Buell and K.L. Pocek, "Custom Computing Machines: An Introduction," *J. Supercomputing*, vol. 9, no. 3, pp. 219–230, 1995.

[6] S.A. Guccione and M.J. Gonzales, "Classification and Performance of Reconfigurable Architectures," in *Proc. 5th Int'l. Workshop on Field-Programmable Logic and Applications*, Oxford, United Kingdom, 1995, pp. 439–448.

[7] B. Radunović and V. Milutinović, "A Survey of Reconfigurable Computing Architectures," in *Proc. 8th Int'l. Workshop on Field-*

Programmable Logic and Application, Tallin, Estonia, 1998, pp. 376–385.

[8] B. Kastrup *et al.*, "Seeking (the right) Problems for the Solutions of Reconfigurable Computing," in *Proc. 9th Int'l. Workshop on Field-Programmable Logic and Applications*, Glasgow, Scotland, 1999, pp. 520–525.

[9] R.D. Wittig and P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1996, pp. 126–135.

[10] J.A. Jacob and P. Chow, "Memory Interfacing and Instruction Specification for Reconfigurable Processors," in *Proc. 7th Int'l. Symp. on Field Programmable Gate Arrays*, Monterey, California, 1999, pp. 145–154.

[11] G.A. Blaauw and F.P. Brooks, Jr., *Computer Architecture. Concepts and Evolution*, Addison-Wesley, 1997.

[12] T.G. Rauscher and P.M. Adams, "Microprogramming: A Tutorial and Survey of Recent Developments," *IEEE Transactions on Computers*, vol. C-29, no. 1, pp. 2–20, 1980.

[13] M. Bolotski *et al.*, "Unifying FPGAs and SIMD Arrays," in *Proc. 2nd Int'l. Workshop on FPGAs*, Berkeley, California, 1994, pp. 1–10.

[14] S. Hauck, "The Roles of FPGA's in Reprogrammable Systems," *Proc. IEEE*, vol. 86, no. 4, pp. 615–638, 1998.

[15] P.M. Athanas and H.F. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11–18, 1993.

[16] M. Wazlowski *et al.*, "PRISM-II Compiler and Architecture," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley, California, 1993, pp. 9–16.

[17] M. Wazlowski, *A Reconfigurable Architecture Superscalar Coprocessor*, Ph.D. thesis, Brown University, Providence, Rhode Island, 1996.

[18] S.M. Trimberger, "Reprogrammable Instruction Set Accelerator," U.S. Patent No. 5,737,631, 1998.

[19] T. Miyamori and K. Olukotun, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1998, pp. 2–11.

[20] J.R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1997, pp. 12–21.

[21] A. Donlin, "Self Modifying Circuitry - A Platform for Tractable Virtual Circuitry," in *Proc. 8th Int'l. Workshop on Field-Programmable Logic and Applications*, Tallin, Estonia, 1998, pp. 199–208.

[22] M.J. Wirthlin *et al.*, "The Nano Processor: A Low Resource Reconfigurable Processor," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley, California, 1994, pp. 23–30.

[23] K.L. Gilson, "Integrated Circuit Computing Device Comprising a Dynamically Configurable Gate Array Having a Microprocessor and Reconfigurable Instruction Execution Means and Method Therefor," U.S. Patent No. 5,361,373, 1994.

[24] Z. Salcic and B. Maunder, "CCSimP – An Instruction-Level Custom-Configurable Processor for FPLDs," in *Proc. 6th Int'l. Workshop on Field-Programmable Logic and Applications*, Darmstadt, Germany, 1996, pp. 280–289.

[25] R.W. Hartenstein *et al.*, "A Novel Paradigm of Parallel Computation and its Use to Implement Simple High-Performance Hardware," *Future Generation Computer Systems*, no. 7, pp. 181–198, 1991/1992.

[26] R. Razdan and M.D. Smith, "A High Performance Microarchitecture with Hardware-Programmable Functional Units," in *Proc. 27th Annual Int'l. Symp. on Microarchitecture – MICRO-27*, San Jose, California, 1994, pp. 172–180.

[27] B. Kastrup *et al.*, "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1999, pp. 92–100.

[28] M.J. Wirthlin and B.L. Hutchings, "A Dynamic Instruction Set Computer," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1995, pp. 99–109.

[29] S.M. Trimberger, "Reprogrammable Instruction Set Accelerator Using a Plurality of Programmable Execution Units and an Instruction Page Table," U.S. Patent No. 5,748,979, 1998.

[30] S. Hauck *et al.*, "The Chimaera Reconfigurable Functional Unit," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1997, pp. 87–96.

[31] S.M. Casselman, "Virtual Computing and the Virtual Computer," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley, California, 1993, pp. 43–48.

[32] A. Lew and R. Halverson, Jr., "A FCCM for Dataflow (Spreadsheet) Programs," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1995, pp. 2–10.

[33] C.R. Rupp *et al.*, "The NAPA Adaptive Processing Architecture," in *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, California, 1998, pp. 28–37.

[34] S. Sawitzki *et al.*, "Increasing Microprocessor Performance with Tightly-Coupled Reconfigurable Logic Arrays," in *Proc. 8th Int'l. Workshop on Field-Programmable Logic and Applications*, Tallin, Estonia, 1998, pp. 411–415.

[35] C. Alippi *et al.*, "A DAG-Based Design Approach for Reconfigurable VLIW Processors," in *Proc. IEEE Design and Test Conf. in Europe*, Munich, Germany, 1999, pp. 778–780.

[36] D. Jones and D.M. Lewis, "A Time-Multiplexed FPGA Architecture for Logic Emulation," in *Proc. IEEE 1995 Custom Integrated Circuits Conf.*, Santa Clara, California, 1995, pp. 487–494.

[37] D.C. Cronquist *et al.*, "Architecture Design of Reconfigurable Pipelined Datapaths," *Advanced Research in VLSI*, pp. 23–40, 1999.

[38] R.A. Bittner and P.M. Athanas, "Wormhole Run-time Reconfiguration," in *Proc. 5th Int'l. Symp. on Field Programmable Gate Arrays*, Monterey, California, 1997, pp. 79–85.

[39] R.W. Hartenstein *et al.*, "A New FPGA Architecture for Word-Oriented Datapaths," in *Proc. 4th Int'l. Workshop on Field-Programmable Logic and Applications*, Prague, Czech Republic, 1994, pp. 144–155.

[40] Srihari Cadambi *et al.*, "Managing Pipeline-Reconfigurable FPGAs," in *Sixth International Symposium on Field Programmable Gate Arrays*, Monterey, California, 1998, pp. 55–64.

[41] C. Iseli and E. Sanchez, "A Superscalar and Reconfigurable Processor," in *Proc. 4th Int'l. Workshop on Field-Programmable Logic and Applications*, Prague, Czech Republic, 1994, pp. 168–174.

[42] M. Sima *et al.*, "A Taxonomy of Custom Computing Machines," Technical report, Delft University of Technology, Delft, The Netherlands, (to be published).