Hardware Compilation for FPGA-based Configurable Computing Machines

Xiaohan Zhu, Bill Lin
University of California, San Diego

Abstract

Configurable computing machines are an emerging class of hybrid architectures where a field programmable gate array (FPGA) component is tightly coupled to a general-purpose microprocessor core. In these architectures, the FPGA component complements the general-purpose microprocessor by enabling a developer to construct application-specific gate-level structures on-demand while retaining the flexibility and rapid reconfigurability of a fully programmable solution. High computational performance can be achieved on the FPGA component by creating custom data paths, operators, and interconnection pathways that are dedicated to a given problem, thus enabling similar structural optimization benefits as ASICs. In this paper, we present a new programming environment for the development of applications on this new class of configurable computing machines. This environment enables developers to develop hybrid hardware/software applications in a common integrated development framework. In particular, the focus of this paper is on the hardware compilation part of the problem starting from a software-like algorithmic process-based specification.

1 Introduction

Traditionally, there has been a strong separation between software running on general-purpose programmable processors and dedicated hardware. Microprocessors can perform many different functions by providing a general mechanism to execute software. They can be programmed to perform different tasks, ranging from the execution of a word processor program to the processing of an audio stream. However, versatility comes at a price: applications implemented in software are usually much slower than their hardware counterparts. In contrast, application-specific hardware circuits provide precisely the functions needed for a specific task. Thus, they can be carefully tuned to the application, resulting in solutions that are often smaller, cheaper, faster, and more power efficient.

DAC 99, New Orleans, Louisiana

The advent of field programmable gate arrays (FPGAs) offers an intriguing alternative. Like a general-purpose microprocessor, FPGAs can be dynamically reconfigured repeatedly to perform many different functions. However, like application-specific hardware, FPGAs can implement specific hardware structures that are tuned to the application. FPGAs consist of large arrays of reconfigurable logic blocks that can be reprogrammed to implement a specific gate-level customization. Thus, FPGAs can serve as a powerful *adaptive computing engine*. Indeed, a number of impressive results have been documented using reconfigurable logic for a number of applications, including RSA encryption [22], DNA sequencing [14], multi-standard video compression [22], automatic target recognition [21], and high energy physics [15], amongst others. Several recent advances have made the case for adaptive computing even more compelling. The densities and performance of FPGAs have increased by several orders of magnitude compared to earlier devices. Currently, FPGA devices with over 1M equivalent gates capacity running at 200MHz clock speeds are emerging. The dynamic reconfigurable times have also dramatically decreased despite the substantial increase in logic capacity. A new generation of devices is emerging that can be dynamically reconfigured at a rate of 250M gates/sec.

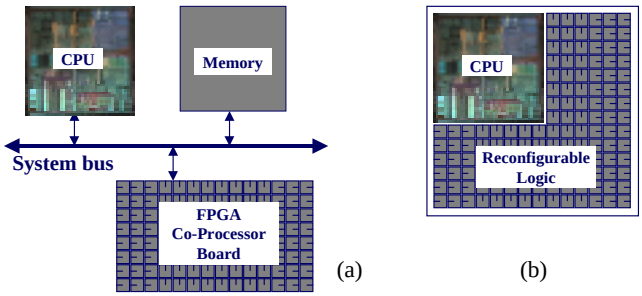


Figure 1: (a) FPGA co-processor board coupled to a microprocessor-based system architecture. (b) FPGA coupled directly to the microprocessor on the same chip.

Recently, a number of hybrid architectures based on the coupling of traditional microprocessors and FPGAs have been proposed, as depicted in Fig. 1. In these architectures, the FPGA component complements the general-purpose processor component by enabling the user to construct application-specific gate-level structures on-demand while retaining the flexibility and rapid reconfigurability of a fully programmable solution. These computing platforms are referred to as *configurable computing machines*. A number of such configurable computing machines have been successfully demonstrated [22, 4, 23, 21, 25, 34]. This earlier generation of configurable comput-

ing machines is based on an add-on FPGA co-processor board approach where the co-processor board is coupled to a microprocessor-based system via a system peripheral bus. Given the increasing scale of integration made possible by deep sub-micron technology, researchers are also exploring FPGA-coupled microprocessors where both the FPGA component and the general-purpose microprocessor are integrated on the same IC [10, 7, 24]. An industrial FPGA-coupled embedded processor from National Semiconductor called the Napa 1000 has already been announced [28].

In this paper, we describe a new programming environment called *Picasso* for the development of applications on configurable computing machines. The programming of these machines will require a new generation of programming tools that will permit programmers without traditional circuit design expertise to develop meaningful hybrid hardware/software applications. To ease programming, the programming language should have the expressive power for describing an entire application, encompassing both the hardware and the software parts. For these reasons, we use a C-like algorithmic programming language that has been extended to support process-based concurrency and inter-process communication. These extensions are based on the CSP [9] model of concurrency and communication.

The organization of this paper is as follows. In Section 2, we introduce the programming model based on the CSP formalism that we use to program hybrid hardware/software applications. In Section 3, we present a high-level overview of the design steps in our compilation trajectory. The focus of this paper is on the hardware compilation part of the problem. Our approach to this problem is described in Section 4. The work presented in this paper has been developed in the framework of a system called *Picasso*. In Section 5, we discuss the status of the project.

2 Programming Model

In this section, we describe a C-like hardware/software programming model, based on the CSP formalism [9]. We believe it has the expressive power to serve as a single executable system specification, encompassing both the hardware and the software parts, so that an application developer can describe an entire application as a single programming activity.

More specifically, our programming language is a process-based programming model. It looks like a C program: the syntactic structure and the expression syntax are nearly identical. However, our model provides language mechanisms not found in C for specifying processes and channel communications, based on the CSP formalism [9]. In addition to its expressive power to handle parallelism and communication, CSP has a rigorously defined semantics along with a well-defined algebra to reason about the concurrent behavior [18], which lends well to formal verification. This section presents a brief overview of our programming model by means of an example.

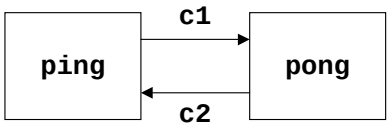


Figure 2: Process model.

Our programs are hierarchically composed of processes that communicate through synchronizing channels. In Fig. 2, a simple example composed of two processes called `ping` and `pong` is illustrated.

```
1.  ping (input chan(int) a, output chan(int) b)
2.  {
3.    int x;
4.
5.    for (;;) {
6.        x = <-a; /* receive */
7.        if(x < 100) x = 10 - x;
8.        else x = 10 + x;
9.        b <-= x; /* send */
10.   }
11. }

12.  pong (input chan(int) c, output chan(int) d)
13.  {
14.    int y, z = 0;
15.
16.    for (;;) {
17.        d <-= 10;/* send */
18.        y = <-c; /* receive */
19.        z = (z + y) % 345; /* send */
20.    }
21.  }

22.  system ( )
23.  {
24.    chan(int) c1, c2;
25.
26.    par {
27.        ping (c2, c1);
28.        pong (c1, c2);
29.    }
30.  }
```

Channels are declared using the `chan` statement, as exemplified in `Line 1`. The unary receive operator, `<-`, receives data on the channel specified as its right operand. The received value may then be manipulated by other operators, e.g. it can be assigned to a variable, as exemplified in `Line 6`. The send operator, `<-=`, transmits the result of the expression provided as its right operand on the channel specified as its left operand, as exemplified in `Line 9`. Basic control-flow constructs, like `if-then-else`, `for-loops`, and `while-loops`, and basic arithmetic and relational operators, like `+`, `-`, `*`, `%`, and `>`, `>=`, `==`, `!=`, are the same as in C. There is also an `alt` construct [9], not used here, that provides a mechanism for non-deterministic execution. Processes can be hierarchically composed to form larger systems, as exemplified by the process `system`. The `par` statement executes the statements in its body in parallel and joins the threads of execution at the end by waiting for all processes to terminate before proceeding. This construct provides a mechanism for invoking concurrency. Finally, other communication constructs such as FIFO-based communication can be programmed on top of the send and receive primitives.

3 Overview of Design Steps

In this section, we present a high-level overview of the design steps in our compilation trajectory. They are comprised of the following: hardware/software partitioning, software compilation, hardware compilation, and hardware/software interfacing. These design steps are described below.

Hardware/Software Partitioning In our programming model, processes provide a natural level of granularity for partitioning. Currently, the assignment of processes to the processor component and the FPGA component is performed by the application developer. This partitioning step arguably requires considerable understanding of the application at hand, which may be best left to the creativity of the developer, as long as the remaining compilation steps can be automated, including hardware/software interfacing. With the remaining compilation steps automated, and the aid of profiling and debugging tools, the developer can quickly experiment with different partitionings.

Software Compilation In the case that only a *single* sequential process is assigned to the processor component, then the software compilation step is straightforward: the sequential process is simply syntactically translated into a C [11] program, which can then be readily compiled to native machine code using conventional optimizing C compilers [1, 19]. In the case where multiple concurrent processes are assigned to the processor component, an embedded multi-threading operating system can be used to support the run-time scheduling of processes, interprocess communication, and context-switching. Again, this involves a relatively straightforward pre-processing step where the processes and communication constructs are syntactically converted to equivalent thread and communication primitives that are supported by the operating system. Alternatively, we have recently developed a *software synthesis* technique that can pre-determine the execution order of operations at compile-time. This technique avoids the need for a run-time operating system. The interested reader can refer to [13] for more details.

Hardware Compilation On the reconfigurable hardware side, well-developed FPGA synthesis tools have been available for some time. Example commercial tools include [30, 31, 26, 33]. These tools perform logic synthesis and technology mapping to cell logic blocks. They are also linked to low-level placement and routing tools. Current FPGA synthesis tools assume a register transfer level description, e.g. using either VHDL [3] or Verilog [2], which essentially describes the behavior on a clock-cycle basis. If we start from an algorithmic behavioral specification, then high-level synthesis steps such as scheduling and resource allocation are needed. A number of high-level synthesis systems have been developed to address this problem [5, 8], including some commercial systems [29, 27]. These systems are based on various control-data-flow-graph (CDFG) models. These models mostly permit only *a single thread of control*, starting from a C-like language. In contrast, our CSP-based model can be used to specify behaviors with *multiple threads of control*. In Section 4, our hardware synthesis approach is further elaborated. The approach is built on top of existing FPGA synthesis tools, using register transfer level VHDL or Verilog as the intermediate representation.

Hardware/Software Interfacing Finally, we need a way to facilitate communication between processes running on the processor and the microprocesses compiled on to the FPGA component. Fortunately, the CSP model provides a high-level abstraction for communication and synchronization. Since the communication architecture coupling the FPGA coprocessor to the microprocessor is predefined, we can solve this communication problem by introducing a library of *channel protocol adapters* that can mediate the communications between the microprocessor and FPGA component. The only requirement is that the library implementation must satisfy the semantics of rendezvous communication. This library implementation can easily be predefined for a given hybrid microprocessor/FPGA architecture.

4 Hardware Compilation

Our approach is based on first compiling the initial CSP specification into an intermediate *Petri net* representation. We then *statically schedule* the operations on the Petri net on to clock cycles via a procedure called *expansion*. The resulting solution is a *single* sequential state machine that can be syntactically mapped to register transfer level VHDL [3] (or Verilog [2]) for logic synthesis and technology mapping to a target FPGA architecture using available FPGA synthesis tools (e.g. [30, 31, 26, 33]).

4.1 Petri Nets and Intermediate Construction

Let $G = \langle P, T, F, m_0 \rangle$ be a Petri net [17], where P is a set of places, T is a set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and $m_0 : P \rightarrow N$ is the initial marking, where N is the set of natural numbers.

The symbols $\bullet t$ and $t\bullet$ define, respectively, the set of input places and the set of output places of transition t. Similarly, $\bullet p$ and $p\bullet$ define, respectively, the set of input transitions and the set of output transitions of place p.

A place p is called a *conflict place* if it has more than one output transition. Two transitions, t_i and t_j are said to be in *conflict* if and only if $\bullet t_i \cap \bullet t_j \neq \emptyset$.

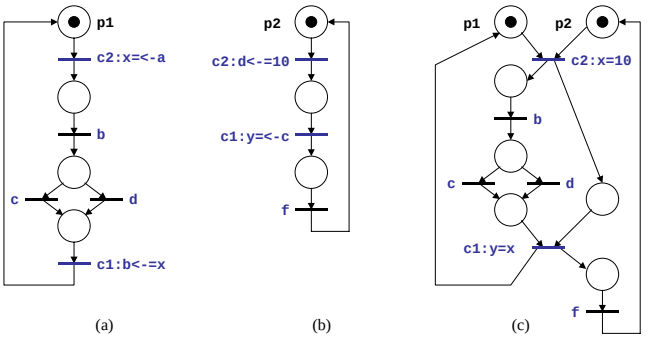
A state, or marking, $m : P \rightarrow N$, is an assignment of a non-negative number to each place. $m(p)$ denotes the number of tokens in the place p. A transition t can fire at marking m_1 if all its input places contain at least one token. The firing of t removes one token from each of its input places and adds a new token to each of its output places, leading to a new marking m_2. This firing is denoted by $m_1 \stackrel{t}{\rightarrow} m_2$.

Given a Petri net G, the reachability set of G is the set of all markings reachable in G from the initial marking m_0 via the reflexive transitive closure of the above firing relation. The corresponding graphical representation is called a reachability graph.

A Petri net G is said to be *live* if $\forall t \in T$, $\exists m$ reachable from the initial marking m_0 such that t is enabled. A Petri net G is said to be *safe* if in every reachable marking, there is at most one token in any place. In this case, we can simply represent each marking $m : P \rightarrow \{0, 1\}$ as a binary assignment.

In [6, 20], a process algebra was developed for constructing a Petri net model from a program of communication processes. Consider again the example shown in Fig. 2. The derived Petri net models for processes `ping` and `pong` are shown in Fig. 3(a) and Fig. 3(b), respectively, along with their initial markings.

Concurrent processes can be composed via parallel composition. In parallel composition, communication actions in fact form synchronization points and are joined together at their common transitions. This is illustrated in Fig. 3(c).



	(a)		(b)		(c)
c2:	x=<-a	c2:	d<-=10	c2:	x=10
b:	(x<0)	c1:	y=<-c	b:	(x<0)
c:	x=10-x	f:	z=(z+y)%345	c:	x=10-x
d:	x=10+x			d:	x=10+x
c1:	b<-=x			c1:	y=x
				f:	z=(z+y)%345

Figure 3: Derived Petri net representations: (a) `ping` (b) `pong` (c) `system = ping || pong`

4.2 Quasi-Static Scheduling and Hardware Generation

In this section, we present a quasi-static scheduling algorithm for Petri nets. It is based on the framework presented in [13] for software synthesis. The algorithm is based on a systematic algorithm for generating *acyclic* Petri net segments from an initial cyclic Petri net representation. For each acyclic Petri net segment, a quasi-static scheduling procedure is applied to schedule the operations (transitions) in that segment. Based on the schedule, a portion of the state machine (representing the control-flow) is generated. After the overall procedure is completed, the resulting state machine is mapped to a register transfer level hardware description for further logic synthesis and technology mapping to the target FPGA architecture.

The procedure for systematically generating acyclic Petri net segment is based on a concept called *maximal expansion.* A maximal expansion is defined with respect to some initial marking m. Starting from this marking, we identify a set of places encountered when a cycle has been reached. These places are referred to as *cut-off places*. Intuitively, the maximal expansion of a Petri net G with respect to a marking m corresponds to the largest *unrolling* of G from m before a cycle has been encountered. Consider the example shown in Fig. 4(a). The corresponding maximal expansion with $m = \langle p1, p2 \rangle$ is shown in Fig. 4(b).

Let G be a Petri net, and let E be a maximal expansion of G with respect to the initial marking m. A marking m_c is said to be a *cut-off marking* if it is reachable from m and no transitions are enabled to fire. The set of cut-off markings is denoted by $CM(E)$. For the example shown in Fig. 4, there are two possible cut-off markings $m_{c_1} = \langle p1', p2' \rangle$ and $m_{c_2} = \langle p3', p4 \rangle$, shown respectively in Fig. 4(c) and Fig. 4(d). From each cut-off marking $m_{c_i} \in CM(E)$, a new maximal expansion segment E_i is generated using m_{c_i} as the initial marking. This iteration terminates when all cut-off markings have already been visited.

In the example shown in Fig. 4, only two expansion segments are needed. From the initial marking $m = \langle p1, p2 \rangle$, the only cut-off markings reachable are $m_c = \langle p1, p2 \rangle$ and $m_c = \langle p3, p4 \rangle$. However, from $m = \langle p3, p4 \rangle$, the only cut-off marking reachable is $m_c = \langle p3, p4 \rangle$ itself, as shown in Fig. 5.

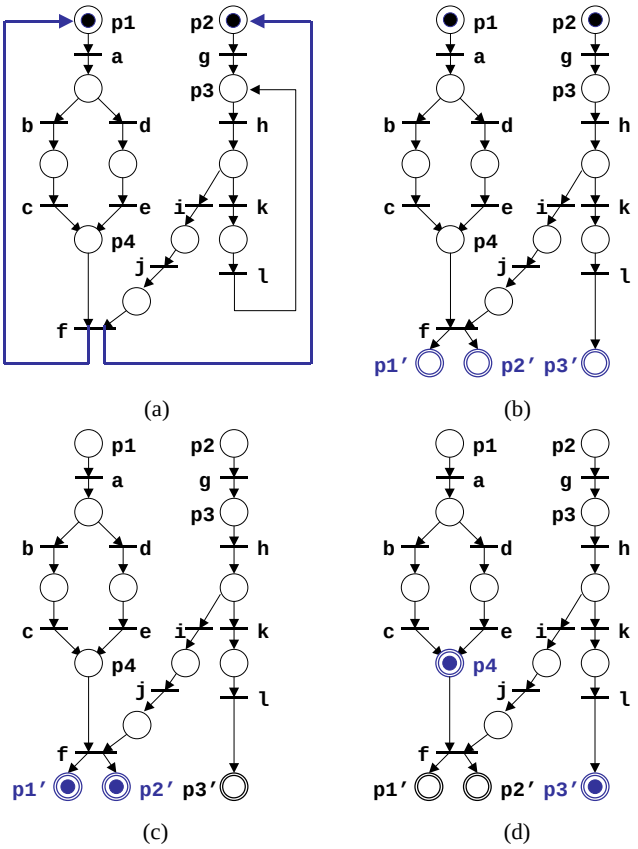


Figure 4: (a) Petri net example. (b) Its maximal expansion. (c) A cut-off marking. (d) Another cut-off marking.

However, in the example shown in Fig. 3, only *one* expansion segment is needed since the only cut-off marking reachable from the initial marking is the initial marking itself (i.e. $m = \langle p1, p2 \rangle$)[1].

Give an expansion segment E, represented as an acyclic Petri net fragment, we perform a *static scheduling* of the operations in that segment. During scheduling, a *step* is assigned to every operation in E. More formally, static scheduling is defined as follows.

Definition 4.1 *Let E be an expansion segment. t_i is said to* precede *t_j in E, denoted as $t_i \prec t_j$, if there is a directed path from t_i to t_j. Let $\pi : T \rightarrow N$, be a* schedule function *that assigns a non-negative integer $\pi(t) \in N$ to every $t \in E$. A schedule is said to be* valid *iff it satisfies the following condition: $\forall t_i, t_j \in E$, if $t_i \prec t_j$, then $\pi(t_i) \leq \pi(t_j)$. If $\pi(t_i) = \pi(t_j)$, then $\forall t_k$ such that $t_i \prec t_k \prec t_j$, $\pi(t_i) = \pi(t_k) = \pi(t_j)$.*

In the case when there exists t_i and t_j, such that $t_i \prec t_j$, but $\pi(t_i) = \pi(t_j)$, it means both operations are executed in the same *clock cycle* combinationally. This technique is referred to as *chaining* in high-level synthesis research [5, 8].

[1] Here, we do not distinguish between p_i and p_i' because they simply denote different instances of the same place.

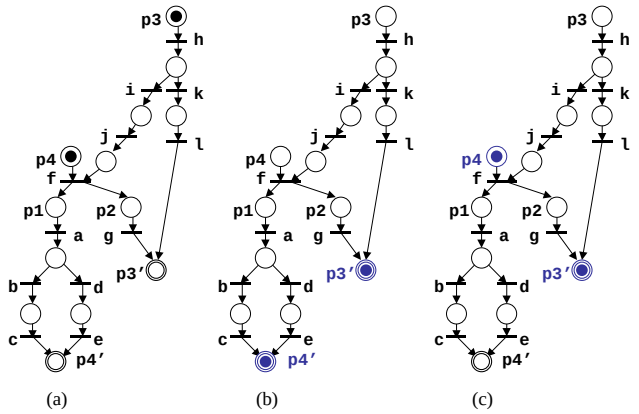


Figure 5: (a) Another maximal expansion. (b) and (c) Cut-off markings.

To illustrate this process, consider the expansion segment shown in Fig. 6(a). A valid schedule is shown. It is not the intention of this paper to discuss in details the different possible scheduling heuristics. The interested reader can refer to [5, 8] for a survey of example techniques.

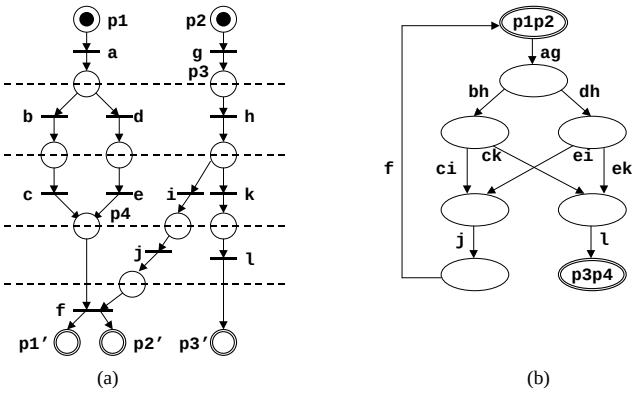


Figure 6: (a) A valid static schedule. (b) Corresponding state machine (control-flow graph) fragment.

Given a schedule π, a state machine fragment SM_π is constructed. The control-flow-graph generation step is based on a traversal of E using Petri net firing rules, but we modify the firing rules so that we proceed in accordance to the levels defined by π. For example, the schedule shown in Fig. 6(a) will result in the state machine fragment depicted in Fig. 6(b).

Currently, the sharing of hardware resources, such as the sharing of arithmetic operators, must be specified manually by the developer via *pragmas*. We have not incorporated automated optimization procedures for this yet. However, we believe resource allocation and sharing heuristics developed in high-level synthesis are applicable to our scheduling framework. This is currently under investigation.

Once the overall state machine has been generated, it can be syntactically translated into a register transfer level VHDL [3] description (or Verilog [2]). Existing FPGA synthesis tools (e.g. [30, 26, 31, 34, 25]) can be used for logic synthesis and technology mapping on to a target FPGA architecture.

5 Status

The hardware compilation method presented in this paper has been developed in the framework of a system called *Picasso*. Our target platform is an FPGA-coupled microprocessor architecture like the ones described in [10, 7, 24, 22, 4, 23, 21, 25, 34]. The overall Picasso system can be used to program both the microprocessor part as well as the FPGA co-processor part. To permit experimentations, we are currently developing a simple simulator for a hypothetical hybrid architecture comprising of a MIPS core and a FPGA component. To build the simulator, we are currently investigating the use of the SPIM instruction-set level simulator for the MIPS R2000/R3000 instruction-set [12] along with a simple cycle-based compiled-code simulator for the reconfigurable hardware part.

6 Summary

In this paper, we described a programming environment for the development of hybrid hardware/software applications for FPGA-coupled configurable computing machines. In particular, we presented a new hardware compilation method that can produce efficient hardware configurations starting from a high-level algorithmic process-based specification.

References

[1] A. V. Aho et al. *Compilers - principles, techniques, and tools*, Reading: Addison-Wesley, 1986.

[2] J. Bhasker. *A Verilog HDL Primer*. Prentice-Hall, 1997.

[3] J. Bhasker. *A VHDL Primer*. Prentice-Hall, 1994.

[4] R. Bittner, P. Athanas. "Wormhole Run-Time Reconfiguration", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ACM, 1997.

[5] R. Camposano and W. Wolf (editors), *Trends in High-Level Synthesis*, Kluwer Academic Publishers, 1993.

[6] G. de Jong, B. Lin. "A communicating Petri net model for the design of concurrent asynchronous modules", *ACM/IEEE Design Automation Conference*, 1994.

[7] A. DeHon. "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century", *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.

[8] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. Van Meerbergen, S. Note, J.A. Huisken, "Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms", *Proceedings of IEEE*, vol.72, no.2, pp.319-335, February 1990.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[10] J. R. Hauser, J. Wawrzynek. "A MIPS Processor with a Reconfigurable Coprocessor" Proc. Symposium on Field-Programmable Custom Computing Machines (FCCM), April 16-18, 1997.

[11] B. W. Kernighan, D. M. Ritchie. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[12] J. Larus, *SPIM* (http://www.cs.wisc.edu/ larus/spim.html), a MIPS R2000/R3000 simulator

[13] B. Lin. "Software synthesis of process-based concurrent programs", *ACM/IEEE Design Automation Conference*, June 1998.

[14] D. P. Lopresti, “P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences”, *Computer*, vol. 20(7), pp. 98-99, 1993.

[15] L. Moll, J. Vuillemin, P. Boucard. “High-energy Physics on DECPeRLe-1 Programmable Active Memory”, *ACM International Symposium on FPGAs*, Monterey, February 1995.

[16] I. Page. “Constructing Hardware-Software Systems from a Single Description”, Submitted to VLSI signal processing, July 1995.

[17] J.L. Peterson. *Petri net Theory and Modeling of Systems*, Prentice Hall, 1981.

[18] A. W. Roscoe, C. A. R. Hoare. “Laws of occam programming”, *Theoretical Computer Science*, 60, 177-229, (1988).

[19] R. M. Stallman, *Using and porting GNU CC*, Free Software Foundation, June 1993.

[20] S. Vercauteren, D. Verkest, G. de Jong, B. Lin, “Derivation of formal representations from process-based specification and implementation models”, *Proc. of ISSS’97*, September 1997.

[21] J. Villasenor, B. Schoner, K. N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, B. Mangione-Smith. “Configurable Computing Solutions for Automatic Target Recognition”, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’96)*, April 1996.

[22] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard. “Programmable Active Memories: Reconfigurable Systems Come of Age”, *IEEE Transactions on VLSI Sysetms*, March 1996, vol. 4, (no.1):56-69.

[23] M. J. Wirthlin, B. L. Hutchings. “DISC: The dynamic instruction set computer”, *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, Proc. SPIE 2607, pp. 92-103 (1995).

[24] R. D. Wittig, P. Chow, “OneChip: An FPGA Processor with Reconfigurable Logic”, *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996.

[25] Altera Corporation (http://www.altera.com), *RIPP10 Programming Board*, California.

[26] Exemplar (http://www.exemplar.com), *Leonardo Spectrum*, Alameda, CA.

[27] Frontier Design System (http://www.frontierd.com), *ART and DSP-Station*, Leuven, Belgium.

[28] National Semiconductor Corporation, *Napa 1000 Data Sheet*, (http://www.national.com/appinfo/milaero/napa1000), 1997.

[29] Synopsys (http://www.synopsys.com), *Behavioral Compiler*, Mountain View, CA.

[30] Synopsys (http://www.synopsys.com), *FPGA Express*, Mountain View, CA.

[31] Synplicity (http://www.synplicity.com), *Synplify*, Sunnyvale, CA.

[32] Virtual Computer Corporation (http://www.vcc.com), California.

[33] Xilinx (http://www.xilinx.com), *Foundation Series Software*, California.

[34] Xilinx Corporation (http://www.xilinx.com), California.