Generic Processor Modeling for Automatically Generating Very Fast Cycle-Accurate Simulators

Mehrdad Reshadi, Bita Gorjiara, and Nikil D. Dutt, *Senior Member, IEEE*

***Abstract*—Detailed *modeling* of processors is required for validating processor behavior and evaluating parameters such as performance and power consumption. Fast cycle-accurate *simulators* are essential in handling today's complex hardware and software designs at a reasonable time. These problems are challenging enough by themselves and have seen many previous research efforts. Addressing both simultaneously is even more challenging, with many existing approaches focusing on one over another. Abstract models in fast simulators do not provide enough information required for different phases of the design. On the other hand, detailed models are very difficult to generate and result in very slow simulators. In this paper, a modeling approach based on *reduced colored Petri net* (RCPN) is proposed, which has the following three advantages: 1) it is very generic and support a wide range of processor features; 2) it offers a very simple and intuitive yet formal way of modeling pipelined processors; and 3) it can generate high-performance cycle-accurate simulators. RCPN inherits all useful features of *colored Petri nets* while avoiding their exponential growth in complexity. In this paper, it is shown how this approach is general enough to model features such as very long instruction word out-of-order execution, dynamic scheduling, register renaming, hazard detection, and branch prediction. Furthermore, the results of generating cycle-accurate simulators from RCPN models of XScale and StrongArm processors are shown, where an order of magnitude (∼15 times on the average) speedup over the popular SimpleScalar advanced reduced instruction set computing machine simulator is achieved.**

***Index Terms*—Microprocessors, modeling, Petri nets, simulation.**

I. Introduction

EFFICIENT and accurate modeling of processors and fast simulation are critical tasks in the development of both hardware and software during the design of new processors or processor-based systems on a chip. Cycle-accurate simulators simulate the functionality of a program and provide performance metrics such as cycle counts, cache hit ratios, branch prediction accuracy, and different resource utilization statistics. While the increasing complexity of processors has improved their performance, it has had the opposite effect on both the complexity and the performance of the simulators. The proposed techniques for improving the performance of cycle-accurate simulators are usually very complex and sometimes domain or architecture specific. Due to the complexity of these techniques and the complexity of the architecture, generating retargetable high-performance cycle-accurate simulators has become a very difficult task.

To avoid redevelopment of new simulators for new or modified architectures, a retargetable framework uses an architecture model to automatically modify an existing simulator or generate a customized simulator for that architecture. Flexibility and complexity of the modeling approach as well as the simulation speed of generated simulators are important quality measures for a retargetable simulation framework. Simple models are usually limited and inflexible, whereas generic and complex models are less productive and generate slow simulators. A reasonable tradeoff between complexity, flexibility, and simulation speed of the modeling techniques has seldom been achieved in the past. Therefore, automatically generated cycle-accurate simulators were more limited or slower than their manually generated counterparts.

In some cycle-accurate simulators, the architecture is described in terms of individual hardware component units. In these approaches, each component may perform a different functionality based on the binary of instructions. Therefore, similar to real hardware, in such simulators, the behavior of an instruction is executed in different phases at different clock cycles. In this way, more accurate results can be achieved at the expense of simulation performance. To improve the simulation performance, an alternative approach is to describe the architecture in terms of instruction behaviors. Typically, such fast cycle-accurate simulators execute the instruction behavior at once, usually in an execute unit, using a functional simulator in their core to decode instructions and simulate their behaviors. This means that the execution of an instruction is separated from its timing information. The cycle-accurate simulator first detects when to execute an instruction and then uses the functional simulator to execute the complete instruction behavior at once. For example, for a *Load* instruction in the advanced reduced instruction set computing (RISC) machine (ARM) processor, checking the condition, calculating the effective address, requesting data from memory, and reading the data happen in different pipeline stages, whereas a functional simulator may execute all of them at once. This approach is sufficient for the cases where only the timing information of the instructions is needed. However, when accurate processor behavior is required, for example, for verifying the hardware algorithms or estimating the power consumption, more accurate simulation models are necessary. Most often, generating and maintaining

Manuscript received May 10, 2005; revised October 31, 2005 and March 6, 2006. This work was supported in part by National Science Foundation under Grant CCR-0203813 and Grant CCR-0205712. This paper was recommended by Associate Editor M. F. Jacome.

M. Reshadi and N. D. Dutt are with the Center for Embedded Computer Systems and the Donald Bren School of Information and Computer Science, University of California, Irvine, CA 92697 USA (e-mail: reshadi@cecs.uci.edu; dutt@cecs.uci.edu).

B. Gorjiara is with the Center for Embedded Computer Systems and the Henry Samueli School of Engineering, University of California, Irvine, CA 92697 USA (e-mail: gorjiar@cecs.uci.edu).

Digital Object Identifier 10.1109/TCAD.2006.882597

such accurate models are very difficult and time consuming, and the corresponding simulators run very slow.

Colored Petri Net (CPN) [1] is a very powerful and flexible modeling technique and has been successfully used for describing parallelism, resource sharing, and synchronization. It can naturally capture most of the behavioral elements of instruction flow in a processor. However, CPN models of realistic processors are very complex mostly due to incompatibility of a token-based mechanism for capturing data hazards. Such complexity reduces the productivity and results in very slow simulators. In this paper, we present *reduced CPN* (RCPN), a generic modeling approach for generating fast cycle-accurate simulators for pipelined processors. RCPN is based on CPN and reduces the modeling complexity by redefining some of the concepts of CPN and also using an alternative semaphore-based approach for describing data hazards. Therefore, it is as flexible as CPN but far less complex and can support a wide range of architectures. Fig. 1 illustrates the advantages of our approach using an example pipeline block diagram and its corresponding RCPN and CPN models. It is possible to convert an RCPN to a CPN and, hence, reuse the rich varieties of analysis, verification, and synthesis techniques that have been proposed for CPN. The RCPN is intuitive and closely mirrors the processor pipeline structure. RCPN provides necessary information for generating fast and efficient cycle-accurate simulators. For instance, our XScale [3] processor cycle-accurate simulator runs an order of magnitude ($\sim$15 times on the average) faster than the popular SimpleScalar simulator for ARM [2]. This is mainly because most of the information about the flow of instructions can be statically extracted from RCPN. It is an instruction-centric model; i.e., for each instruction in the architecture instruction set, RCPN defines *where* its next state is, *when* the instruction can advance, and *what* should be done on the way from one state to another. In other words, once the instruction is decoded, everything about its behavior in time (clock cycle) and space (pipeline stage) will be known. In our modeling approach, RCPN captures the schedule of instructions (controller of processor) and their main behavior. Other components such as cache and predictors are described in a library, and the RCPN has references to them to control the flow of instructions. To further improve the productivity and efficiency of the model, we group similar instructions and describe their behavior using instruction templates.

In this paper, Section II summarizes the related works. Section III describes the RCPN model, and Section IV demonstrates the RCPN model of several processor features and examples including the one in Fig. 1. Section V describes how RCPNs can be converted to CPNs. Section VI explains the simulation engine and optimizations that are possible because of RCPN. Section VII shows the experimental results, and Section IX concludes this paper.

II. Related Work

Detailed microarchitectural simulation has been worked on for many years, and several models and techniques have been proposed to automate the process and improve the performance of the simulators. In interpretive simulators, instructions of the program are fetched, decoded, and executed at run time. Compiled simulators improve the performance of simulation by first analyzing, decoding, and compiling the instructions of the program into one or more functions and then executing the corresponding functions at run time. Compiled simulators are much faster than interpretive simulators. However, they are more complex and sometimes less flexible.

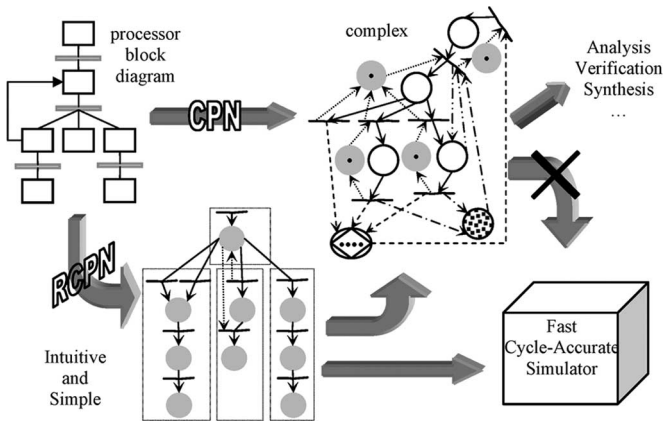

Fig. 1. Advantages of RCPN: intuitive fast simulation.

An alternative to detailed simulation is analytical modeling. These techniques such as those reported in [4], [38], and [39] avoid the detailed execution of program and estimate the performance and other quality measures by extracting analytical formulations that model the behavior of application. Such techniques have speed advantages over detailed simulation and can provide valuable insight. However, these techniques are not as accurate as detailed simulators and, hence, are out of the scope of this paper.

In hardware-centric approaches, the architecture is modeled and described in terms of individual hardware component units. In these approaches, each component may perform a different functionality based on the binary of instructions. In such cases, the simulator executes the hardware components. Therefore, the complete behavior of an instruction is actually executed by the collective behavior of different components at different times. Since the behavior of each component can be described independent of other components in the system, they can be reused in different designs, and hence, the hardware-centric approaches are usually easier to maintain and expand. However, the corresponding simulation engines are typically event-driven because they have to simulate multiple parallel components without having any global picture of the system functionality. Therefore, these simulators are usually slow. Hardware-centric approaches such as BUILDABONG [14] and MIMOLA [13] model the architectures at register transfer level and lower levels of abstraction. This level of abstraction is not suitable for complex microprocessor modeling and results in very slow cycle-accurate simulators. Similarly, ASim [15] and Liberty [16] model the architectures by connecting hardware modules through their interfaces. Emphasizing reuse, they use explicit port-based communication that increases the complexity of these models and have a negative impact on the simulation speed. SystemC [17] is a C++ library that supports the discrete event model of computations and uses explicit port-based

communication scheme as well, suffering from a similar degradation in cycle-accurate simulation performance. UPFAST [19] takes a hardware-centric approach and abstracts the ports and connections away to improve productivity as well as efficiency for synthesized simulators. However, since all pipeline hazards need to be resolved explicitly by the user, modeling superscalar processors with complex control is difficult in this approach. Chang *et al.* [20] have proposed a hardware-centric approach that implicitly resolves pipeline hazards in the cost of an order of magnitude slow down in simulation performance.

In operation-centric approaches, the architecture is described in terms of its instructions or subinstruction operations. The high-level behavior description of these operations is usually linked to one or more hardware components. Operation-centric approaches provide a higher level view of architecture compared to hardware-centric approaches. Therefore, the operation-centric architecture models can be used for code generation and can also lead to faster architectural simulation. Zhu and Gajski [37] have proposed a compiled simulation technique for generating instruction-set simulators. They define a simulation code generation interface that decouples the host processor from the target processor. The instruction set of target processor is described in terms of functions of the simulation code generation interface. Each host processor implements this interface by either emitting C code or host machine assembly code. In this way, the target program is statically decoded in to C code or host assembly. The result is then compiled on the host machine to create the executable program of the compiled simulator. This approach creates very fast instruction-set simulators but cannot support detailed cycle-accurate simulation of the processor. EXPRESSION [9], ISDL [8], and nML [7] are operation centric, and they automate the development of code generators. However, they are not suitable for generating detailed cycle-accurate simulators. EXPRESSION implicitly captures the pipeline structure through the notions of control and data transfer paths [10]. ISDL and nML do not explicitly support detailed pipeline control-path specification. The Sim-nML [11] language is an extension to nML to enable cycle-accurate modeling of pipelined processors. Sim-nML extends nML by adding a *resource usage model* that assigns a sequence of resources $(R_0, R_1, \ldots, R_n)$ to each instruction. During simulation, as soon as resource R_{i+1} becomes free, the instruction releases resource R_i and holds the next resource in the sequence (R_{i+1}). While generating slow simulators, the proposed language cannot describe processors with complex pipeline control mechanisms due to the simplicity of the underlying instruction sequencer. For example, a forwarding path between two units in a pipeline cannot be modeled in this scheme. An instruction may not proceed based on only the availability of a forwarding path; it needs to know the exact value on the path at the exact clock cycle. SimC [12] is based on a machine description in ANSI C that describes the behavior of instructions in each clock cycle. The program is statically decoded, and different stages of instructions are combined to create a new program, in C, which represents the behavior of the processor in each clock cycle. The corresponding compiled simulator is created by compiling the source code of the new program into an executable program that simulates the original program. Architectural features such as dynamic scheduling or variable timing (e.g., for memory accesses) are not easily supported in this approach, and it also has limited retargetability. Park *et al.* [21] have proposed an operation-centric technique called "early pipeline evaluation" for simulating simple RISC-like pipelines. In this technique, instructions execute independently and separately. The effects of concurrency are simulated by maintaining two separate pro cessor states, resulting in a complex modeling of stalls and hazards. Furthermore, since instructions are executed independently, features such as data forwarding and dynamic scheduling cannot be modeled with this technique.

SimpleScalar [2] is a tool set with significant usage in the computer architecture research community, and its cycle-accurate simulators have good performance. It uses a fixed architectural model with limited flexibility through parameterization. Babel [18] was originally designed for retargeting the GNU binary tools such as assembler and linker and has also been used for retargeting the SimpleScalar simulator. MicroLib [35] is a recent attempt that advocates modular simulation for processors and aims to provide an open library of simulatable architectural components that can be used in other simulation frameworks such as SystemC and SimpleScalar. For example, Perez *et al.* have reported the results of their experiments with different data cache module added to SimpleScalar [36]. FastSim [22] uses the "fast-forwarding" technique to perform an order of magnitude (i.e., 5–12 times) faster than SimpleScalar. Fast-forwarding is one of very few techniques with such a high performance; however, generating simulators based on this technique is very complex. To decrease this complexity, the Facile language [23] has been proposed to automate the process. However, the automatically generated simulators suffer a significant loss of performance compared to FastSim and run only 1.5 times faster than SimpleScalar. Besides, modeling in Facile requires more understanding of fast-forwarding simulation technique rather than the actual hardware being modeled.

LISA [24] uses the L-chart formalism to model the operation flow in the pipeline and simplifies the handling of structural hazards. It has been used to generate fast interpretive [33] and compiled simulators [32]. In LISA, each instruction is described in terms of several single-cycle operations that execute in their corresponding pipeline stage. The schedule of these operations is specified statically and explicitly with the L-chart formalism in LISA. Upon execution of an instruction, the operation sequencer in the simulator schedules all of the operations of that instruction on their corresponding resources in proper clock cycles considering the resource availabilities. The flexibility of LISA is limited by the L-chart formalism [34], and to the best of our knowledge, no LISA-based model for out-of-order architectures has been reported.

Qin and Malik have proposed the operation state machine (OSM) [25], which models a processor in two layers, namely: 1) the hardware layer and 2) the operation layer. The hardware layer captures the functionality and connectivity of hardware components, simulated by a discrete event simulator. The operation layer captures the flow of operations in the hardware using finite state machines (FSMs). The FSMs communicate with the underlying hardware components by exchanging tokens (events) through token management interfaces (TMIs), which

define the meaning of these tokens. The number of OSMs in this model depends on the number of pipeline paths that operations can flow through. For example, to model a RISC processor such as StrongARM [27], only one OSM is used, and thus, most of the functionalities of processor are implemented by TMIs and hardware layer rather than the OSMs. The generated simulators in this model run as fast as SimpleScalar.

Petri nets have been successfully used to model and analyze pipelined processes. They are very flexible and powerful and additionally enable many formal analyses. Simple Petri net models are also easy to visualize. CPNs [1] simplify the Petri nets by allowing the tokens to carry data values. However, for complex designs such as processor pipeline, their complexity grows exponentially, which makes modeling very difficult and significantly reduces simulation performance. Because of these problems, there has been very few attempts to practically use Petri net (or its extensions) for modeling pipelined processors. Petri net utility tools [28] are a set of tools allowing user to manage the complexity of a Petri net model of a processor. The Petri net models of the processor in this tool set are statistical and cannot be used for detailed simulations. For example, the user specifies that an instruction may right back its results with a probability of 0.2. The simulator of this model generates a trace of the processor state at different times, and a statistical tool analyzes this trace to calculate higher level concepts such as processor utilization.

In this paper, we propose the RCPN model that benefits from Petri net features while being simple and capable of deriving high-performance cycle-accurate simulators. It is an instruction-centric approach and captures the behavior of instructions in each pipeline stage at every clock cycle via modified and enhanced CPN concepts. Unlike Facile and fast-forwarding, writing RCPN models does not require any specific simulation knowledge. In terms of modeling capabilities, OSM and LISA are the closest to RCPN; however, RCPN is more formal and concise and captures a wider range of architectural features.

III. RCPN

To describe the behavior of a pipelined processor, operation latencies, and data, control and structural hazards must be captured properly. A token-based mechanism such as CPN can easily model variable operation latencies and basic structural and control hazards. Because of architectural features such as register overlapping, register renaming, and feedback paths, capturing data hazards using a token-based mechanism is very complex and difficult. In RCPN, we redefine the concepts of CPN to make it more suitable for processor modeling and fast simulation. As for the data hazards, we use a separate mechanism that is explained in Section III-A.

The goal of CPN is to simplify the complexity of standard Petri net and enhance its modeling capability. In standard Petri net, tokens are simple indicators that show if a condition is true or false. In CPN, a token can carry complex values from a type-set. While maintaining the CPN properties, RCPN further simplifies it to make it applicable for complex processor designs. RCPN has a cleaner structure that is easier to understand and also results in much faster processor simulators. Fig. 2 shows a very simple pipeline structure and its CPN and RCPN models. There are two latches and four units in this pipeline. In Fig. 2(b), circles show *places* (states), boxes show *transitions* (functions), and black dots represent *tokens*. In CPN, a transition is enabled when it has one token of proper type on each of its input arcs. An enabled transition can *fire* and remove tokens from its input places and generate tokens for its output places. In Fig. 2(b), whenever U_1 is enabled, it removes the token of L_1 and puts it in P_1. Then, U_2 or U_4 can fire and put one token back into L_1. In other words, whenever a token is in place L_1, it means that latch L_1 in Fig. 2(a) is available, and unit U_1 can send it a new instruction; and whenever a token is in place P_1, it means that an instruction is available in latch L_1, and unit U_2 or U_4 can use it. The number and complexity of the circular loops in the CPN of a typical pipeline grow very rapidly with its size. These loops not only make the CPN models very complex but are also the main obstacle in generating high-performance simulators.

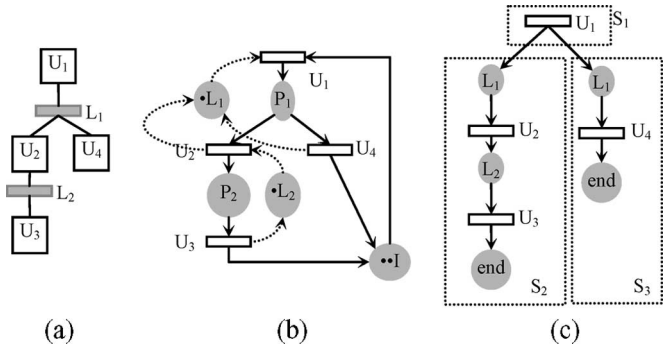

Fig. 2. (a) Example pipeline structure and (b) its CPN and (c) RCPN models.

The RCPN is based on the same concept as CPN; i.e., when a *transition* is enabled, it fires and removes *tokens* from the input *places* and generates tokens for the output places. To make RCPN more suitable for processor modeling we redefine these elements of CPN as follows.

1) *Places:* A place shows the state of an instruction. To each place, a pipeline stage is assigned. A pipeline stage is a latch, reservation station, or any other storage element in the pipeline that an instruction can reside in. For each pipeline stage that an instruction may go through, there will be at least one place in the model. Each pipeline stage has a *capacity* parameter that determines how many tokens (instructions) can reside in it at any time. We assume that when instructions finish, they go to a final virtual pipeline stage, called *end*, with unlimited capacity. The places to which this virtual final stage is assigned represent the final state of the corresponding instructions. In RCPN, each place is shown with a circle in which the name of the corresponding pipeline stage is written. Places with similar name share the capacity of the same pipeline stage. The tokens of a place are stored in the corresponding pipeline stage.

2) *Transition:* A transition represents the functionality that must be executed when the instruction changes its state (place). This functionality is executed (fired) when the transition is enabled. A transition is enabled when its *guard* condition is true, and there are enough tokens of proper types on its input arcs, and the pipeline stages of the output places have enough capacity to accept new tokens. A transition can directly access

nonpipeline units such as branch predictor, memory, cache, etc. The transition may use the functionality of these units to determine the type, value, and delay of tokens that it sends to its output places.

3) *Arc:* An arc is a directed connection between a place and a transition. An arc may have an expression that converts the set of tokens that pass through the arc. For deterministic execution, each output arc of a place has a priority that shows the order at which the corresponding transitions can consume the tokens and become enabled. The arc priorities are added to the model to simplify the guard conditions of the transitions. Since RCPN is used for modeling processors and the behavior of a processor is always deterministic, the guard conditions will be exclusive. Without arc priorities, the guard conditions should explicitly check the order at which the tokens are consumed. Arc priorities simplify these conditions in the model. In this way, the guard conditions of transitions in the model mostly deal with dynamic situations that depend on processor status at run time. During simulation, these guard conditions are automatically expanded in order to consider static conditions such as arc priorities. The expanded conditions determine the actual order of execution of transitions and, hence, flow of instructions at run time.

4) *Token:* There are two groups of tokens, namely: 1) *reservation* tokens that carry no data (their presence in a place indicates the occupancy of the place's corresponding pipeline stage) and 2) *instruction* tokens that carry complex data, depending on the type of the instruction.

Instruction tokens are the main focus of the model since each instruction token represents an instance of an instruction being executed in the pipeline. In other words, RCPN describes how an individual instruction flows through stages of the pipeline. For each type of instruction, it describes *where* the next stage is, *when* instruction can advance, and *what* should be done during transition. In any RCPN, there is one instruction-independent subnet that generates the instruction tokens, and for each instruction type, there is a corresponding subnet that distinctively describes the behavior of instruction tokens of that type. For example, Fig. 2(c) shows the RCPN model of the simple pipelined data path shown in Fig. 2(a). The model is divided into three subnets, i.e., S_1, S_2, and S_3. Subnet S_1 describes the instruction-independent portion that generates two types of instruction tokens. In this manner, only one type of instruction token can flow in each subnet. Note that as long as state L_1 has room for a new token, transition U_1 can fire. In fact, because of our new definition of "transition enable," an RCPN model can start with a transition as well as a place. Any subnet can generate instruction tokens and send them to their corresponding subnet. This is equivalent to instructions that generate multiple microoperations in a pipeline (e.g., the Multiple LoadStore instruction in XScale).

A delay may be assigned to a place, a transition, or a token. The delay of a place determines how long a token should reside in a place before it can be considered for enabling an output transition. The delay of a transition expresses the execution delay of the functionality of that transition. The delay of a token overwrites the delay of its containing place and has the same effect. By changing the delay of a token, a transition can indirectly change the delay of its output place. In Section IV-B, we show that these simple elements of model are enough to describe behavior of instructions in a very long instruction word (VLIW) machine.

Usually, in microprocessors, the instructions that flow through a similar pipeline path have similar binary format as well. In other words, the instructions that go through the same functional units have similar fields in their binary format. Therefore, a single decoding scheme and behavior description can be used for such group of instructions, which we refer to as an *operation class*. An operation class describes the corresponding instructions by using *symbols* to refer to different fields in their binary code. A symbol can refer to a constant, a μ-operation, or a register. Using these symbols, for each operation class, an RCPN subnet describes the behavior of the corresponding instructions. During instruction decode, the actual values of these symbols are determined. Therefore, by replacing the symbols with their values, a customized version of the corresponding RCPN subnet is automatically generated for individual instances of instructions. Fig. 4 shows examples of such operation classes. This way of grouping instructions and using templates for simplifying models and improving simulation performance of instruction-set simulators is explained in [5] and [6]. We use a similar approach to simplify our RCPN models and generate cycle-accurate simulators. The structure of templates that are produced from RCPN models are the same as those created in [5] and [6]. However, instead of having just one function that simulates the functional behavior of the instruction, we add multiple functions that simulate the behavior of instructions in different clock cycles. Section VI explains the details of detecting and constructing these functions.

A. Capturing Data Hazards in RCPN

To capture data hazards, we need to know when registers can be read or updated and if an instruction is going to update a register and what its state is at any time. In many processors, registers may overlap,[1] hence modifying one may affect the others. On the other hand, generally, instructions use different pipeline stages to read source operands, calculate results, or update destination operands. Therefore, instructions must be able to hold register values after reading or before updating registers.

Addressing all these issues with a token-based mechanism is very complicated, and hence, in RCPN, we use an alternative approach that explicitly supports a lock/unlock (semaphore) mechanism for accessing registers, allocates temporary locations for register values, and supports overlapping registers. As Fig. 3 shows, we model registers in three levels.

1) *Register file:* It defines the actual data storages, register renaming, and pointer to instructions that will write to a register. There may be multiple register files in a design.

2) *Register:* Each register has an index and points to proper storages of the register file. Multiple registers can point to the same storage areas to represent overlapping.

3) *Register reference (RegRef):* Each RegRef points to a register and has an internal storage for storing the register value.

[1]For example, overlapping register banks in ARM or register windows in Scalable Performance ARChitecture (SPARC).

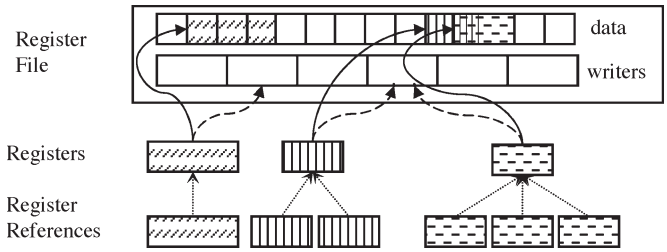

Fig. 3. Register structure in RCPN.

A symbol in an operation class that points to a register is replaced by a proper RegRef during decode. In fact, RegRefs represent the pipeline latches that carry instruction data in real hardware. During simulation, this is almost equivalent with renaming registers for each individual instruction. RegRefs' internal values are used in the computations, and the instructions access and update registers through RegRefs' interfaces. The interface is fixed and includes the following commands: $canRead()$, which returns true if register is allowed to be read; $canRead(s)$, which returns true if the instruction that is going to update the corresponding register is in state s at the time of call; $read()$, which reads the values of corresponding register and stores it in the internal storage of RegRef; $canWrite()$, which returns true if the register is allowed to be written; $reserveWrite()$, which assigns the current RegRef pointer and its containing instruction as the writers of the corresponding register; $writeback()$, which writes the internal value of the RegRef to the corresponding register and may reset its writer pointers; and $read(s)$, which, instead of reading the value of the corresponding register, reads the internal value of the writer RegRef whose containing instruction is in state s at the time of call. The $read(s)$ interface provides a simple and generic means of modeling data forwarding through feedback or bypass paths.

In RCPN, data hazards are explicitly captured using Boolean interfaces such as *canRead* in the guard condition of arcs as well as non-Boolean interfaces such as *read* in the transitions. These pairs of interfaces must be used properly to ensure correctness of the model. Whenever $read()$, $reserveWrite()$, or $read(s)$ appears in a transition, $canRead()$, $canWrite()$, or $canRead(s)$ must appear in the guard condition of its input arc, respectively.

The implementation of these interfaces may vary based on architectural features such as register renaming. For example, in a typical implementation of these interfaces, a transition first checks $r.canWrite()$ to check write-after-write and write-after-read (WAR) hazards for accessing register r. Then, it calls $r.reserveWrite()$ to prevent future reads or writes. After calling $r.writeback()$ in another transition, register r can be safely read or written. In RCPN, a symbol in an operation class that points to a constant is replaced by a *Const* object during decode. The Const object provides the same interface as of RegRef with proper implementation. For example, its $canRead()$ always returns true, its $writeback()$ does nothing, and so on. In this way, data hazards can be uniformly captured using symbols in the operation class.

In Section IV, we show by example how RCPN can capture structural/data/control hazards and operation latencies and how it can model processor features such as out-of-order execution/completion, VLIW, multi-issue processors, branch prediction, dynamic scheduling, register renaming, and in-order completion.

```
Branch {
   condition: {Register | Constant}
   offset: {Register | Constant}
};
Integer {
   op: {Add | Sub | Mul | Div | ...}
   d, s1: {Register}
   s2 : {Register | Constant}
};
FP {
   op: {Add | Sub | Mul | Div | ...}
   d, s1: {Register}
   s2 : {Register | Constant}
};
LoadStore {
   L: {true | false}
   r: {Register}
   addr: {Register | Constant}
};
```

Fig. 4. Operation classes of a simple ISA.

IV. RCPN Model of Processor Features by Example

In this section, we explain three different examples that cover most of the features that can be found in today's processors. We assume that all of the example processors use the simple instruction set in Fig. 4. This instruction set is described by four operation classes, i.e., *Branch*, *Integer*, *FP* (floating point), and *LoadStore*. Each operation class consists of symbols whose actual value is determined during instruction decode. For example, the L symbol in LoadStore is a Boolean symbol and is true for load and false for store instructions. Similarly, the *offset* symbol in Branch will be decoded to a register or a constant (RegRef or Const as described in Section III-A). After an instruction instance is decoded, the actual values of these symbols can be accessed in the transitions of the RCPN. For an Integer instruction token t, $t.op$, or $t.d$ represents the values of the corresponding symbols. Note that some of these symbols are functions. For example, $t.d = t.op(t.s_1, t.s_2)$ applies the corresponding arithmetic operation to sources s_1 and s_2 and stores the computation result in symbol d. Additionally, tokens may have some default attributes. For example, $t.delay$ and $t.pc$ show the delay of the token and the address of the instruction token, respectively.

In the RCPN models of the following examples, circles show the places, which are labeled by the name of their corresponding pipeline stage. The boxes show the functionality of transitions, and the codes above them show their guard conditions. The guard conditions are written in the form of $[cond_1, cond_2, \ldots]$, which is equivalent to $cond_1 \wedge cond_2 \wedge \cdots$.

A. Simple Processor With Out-of-Order Execution/Completion

Fig. 5(a) shows the block diagram of a representative out-of-order completion processor with data forwarding. To show the flexibility of the model, we assume that the forwarding

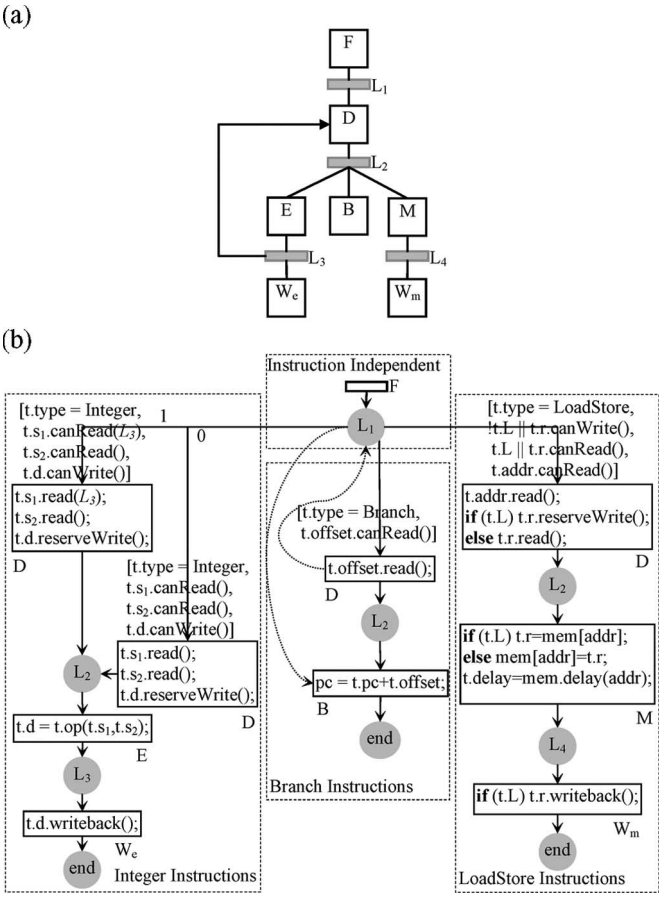

Fig. 5. (a) Out-of-order execution processor and (b) its RCPN subnets.

path is used only for the first source operand of Integer instructions, i.e., s_1. Fig. 5(b) shows the complete RCPN model of the processor, which contains one instruction-independent subnet and three instruction-specific subnets. The instruction-independent subnet fetches one instruction per cycle and sends it to L_1.

The arcs that come out of L_1 have guard functions that check the source and destination operands for data hazards using the specified interface. To model the feedback path, two arcs with different priorities come out of place L_1 and enter the Integer Instruction subnet. If the first arc, with priority 0, cannot read the value of first source operand, then the second arc, with priority 1, checks if the writer instruction of operand s_1 is in the pipeline stage L_3. If that is the case, then source operand s_1 is loaded by destination operand of the instruction residing in L_3. After reading the source operands and reserving the destination operand for writing, the result is calculated in transition E and stored in the internal value of the destination d. This value is finally written back in transition W_e.

In the Branch Instruction subnet, the dotted arcs represent reservation tokens. Therefore, in this example, when a branch instruction is issued, it stalls the fetch unit by occupying latch L_1 with a reservation token and disabling the fetch transition. In the next cycle, this token is consumed, and the fetch unit is resumed. Note that, as an alternative, we can flush the L_1 and L_2 latches in transition B instead, or use a branch predictor as shown in the next example.

The LoadStore Instruction subnet demonstrates the use of token delay in transition M to model the variable delay of memory (cache). It also shows how part of the RCPN model is customized based on the value of symbol L. This is a convenient feature for grouping the instructions and simplifying the model. However, to avoid any negative impact on the simulation speed, only static symbols, whose value is known during instruction decode, should be used. By using partial evaluation, such symbols can be optimized away during subnet customization. If in Fig. 5 latch L_4 is replaced by L_3 in the model, the result will simply be an in-order-execution RISC processor.

B. VLIW and Multi-issue Processors

This section presents modeling of a simple two-slot VLIW processor. We assume that the architecture can issue one integer and one floating point instruction in each cycle and that it has a branch prediction unit (BPU). Since in VLIW, all the data hazards are handled by a compiler, no data dependency checking is necessary in the RCPN model. Fig. 6(a) shows the pipeline structure of each instruction slot. Fig. 6(b) shows the RCPN model of this architecture, which has four subnets, namely: 1) Instruction Independent, 2) Integer Instruction, 3) Branch Instruction, and 4) FP Instruction. The Instruction Independent subnet reads the instructions, and if they are not NOP, then it generates and fires their tokens. In the Integer Instruction subnet, the operands are simply fetched, executed, and written back to the destination register. In the Branch Instruction subnet, the offset is read from the register file and the BPU is updated. We assume that BPU implements two functionalities, namely: 1) *getNext*, which either predicts the next pc or returns $pc + 8$, and 2) *update*, which updates BPU's internal state based on the outcome of branch. In case of branch miss prediction, the L_1 and L_4 latches are flushed, and the pc is reloaded with the correct address. The internal state machine of BPU can also be expanded and added to the model. The floating point subnet is similar to the integer subnet.

For a multi-issue processor, the fetch unit is similar to the example given above, except that instead of generating a fixed number of tokens, it generates up to a limited number of tokens based on the availability of resources. The model of the rest of the processor can be similar to our previous example in Section IV-A or can use advance techniques such as dynamic scheduling and register renaming, which we describe in Section IV-C.

C. Tomasulo Algorithm

This section describes how RCPN can be used in modeling complex processor features such as dynamic scheduling, reservation stations, register renaming, bus structural hazard, and data broadcasting. Fig. 7(a) shows an architecture that has one integer unit and one floating point unit, where each one has three reservation stations. In this architecture, after an instruction is fetched, it goes through the following three steps: 1) *Issue:* if there is an empty reservation station and there is no read-after-write (RAW) hazard, available operands are read from the register file to the reservation station. If no reservation

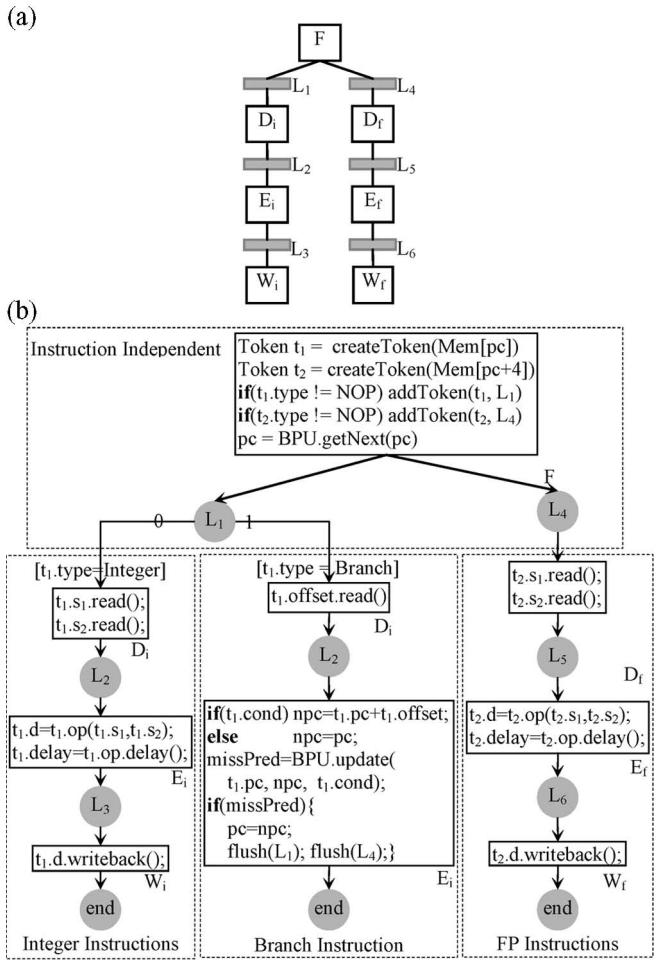

Fig. 6. (a) VLIW pipeline and (b) its RCPN model.

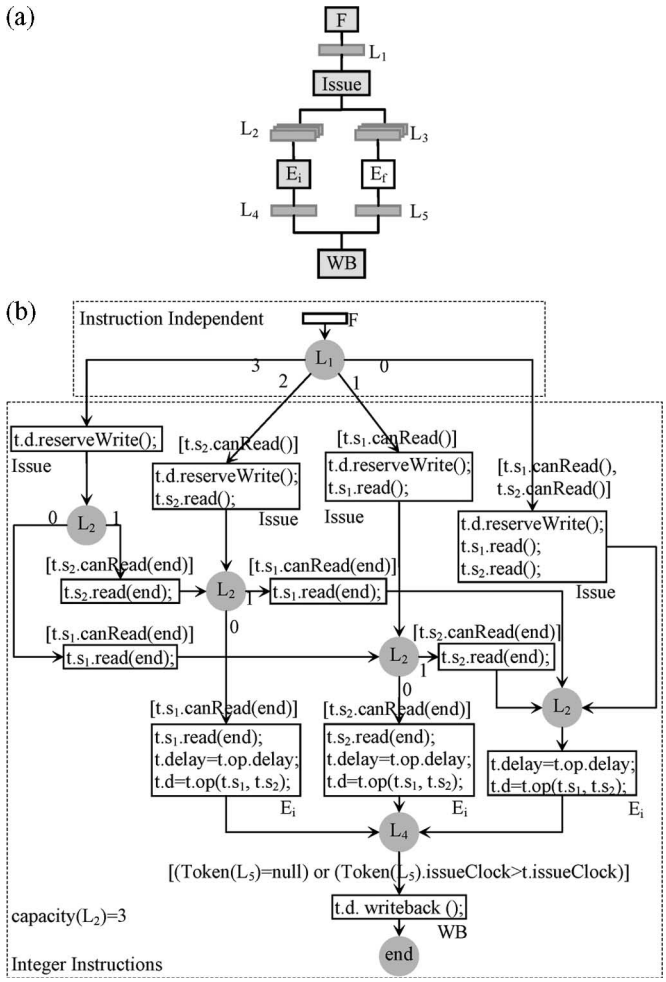

Fig. 7. (a) Architecture with reservation station using (b) the Tomasulo algorithm for integer pipeline.

station is available, the instruction (unit) is stalled until a station is freed. 2) *Execute:* if one or more of the operands is not yet available, then the common data bus (CDB) is monitored, and as soon as the operand becomes available, it is stored in the corresponding reservation station. Whenever both operands are available, the execution of the operation starts. 3) *Write Back:* when execution is finished, it checks for the availability of CDB and then writes on the bus. In case that both integer and floating point units finish the execution at the same time, the older instruction has higher priority for writing on the CDB [26]. Fig. 7(b) shows the RCPN model of the Integer Instruction subnet. After an instruction is issued into the reservation station, depending on the number of operands that are initially available and the order that unavailable operands become available, the instruction transitions between four possible states, all corresponding to reservation station L_2. The states show that both operands are read, only $s1$ or $s2$ is read, or none of them are read. The available operands are read using the $read()$ method, and the value is stored inside the instruction token. Storing the operand value implements the register renaming technique and avoids the WAR hazard. Furthermore, to avoid RAW hazard, the destination register is reserved by the instruction, and future instructions that use this register must wait until it is released and its value put on CDB. The instructions waiting for an operand must monitor the status of the source instruction in each clock cycle using the $canRead(end)$ method. If the source instruction is in the *end* stage, then it must have put the results on the bus, and so the operand can be read. If both operands are read and the execution unit is not busy (i.e., L_4 can accept a new token), then the instruction executes and generates its results; otherwise, the instruction waits in L_2. The generated results must be broadcasted over the shared CDB, and therefore, the model must be able to detect the structural hazard on the bus. Here, the bus is shared between instructions sitting in latches L_4 and L_5. For Integer instructions, the guard expression of the write back transaction must examine the status of L_5 before execution. If there is no token in L_5, then the token in L_4 proceeds; otherwise, the issue times of the tokens are compared, and the older one will proceed.

In general, to model a reservation station, its *capacity* and *behavior* must be captured. The *capacity* affects the structural hazards in the architecture and is modeled by the capacity of the pipeline stage assigned to the corresponding places in the RCPN. The *behavior* updates the state of instruction while it is in the reservation station and is modeled inherently by transitions and places. Modeling register renaming is very simple in RCPN because instruction tokens carry the value of registers in the RegRefs, which play the role of renamed registers in the actual hardware. This example also shows

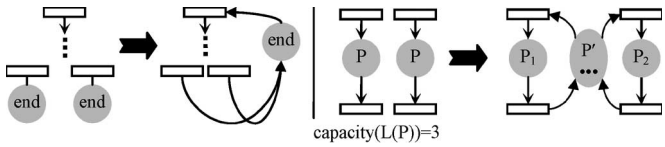

Fig. 8. Converting RCPN places to CPN places.

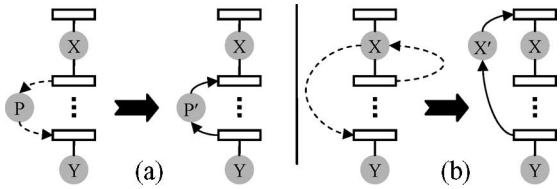

Fig. 9. Converting reservation tokens.

how to arbitrate the conflicts for accessing shared resources using guard expressions. While RCPN is inherently a parallel model, any kind of sequencing is simply implemented through guards, as shown for the in-order completion of the example given earlier. Finally, the $read(s)$ and $canRead(s)$ interfaces of RegRefs are enough to model forwarding paths and data broadcasting in RCPN.

V. Converting RCPN to CPN

To show that RCPN models maintain the formality of standard CPNs, in this section, we briefly explain how RCPNs can be converted to CPNs. In CPN, the capacity of a place is not part of the model, and transitions are enabled whenever there are enough tokens of proper type in their input places, irrespective of the number of tokens in their output places. Furthermore, the priority of output arcs in RCPN must be converted to proper conditions added to the guards of corresponding transitions. Note that because of these priorities, RCPN models are deterministic, whereas CPN models may be nondeterministic. Therefore, any RCPN model can be converted to an equivalent CPN, but the reverse may not be always possible. Note that processors must have a deterministic behavior, and hence, they can be represented in RCPN.

A. Converting Places

In RCPN, homonymous places share the capacity of a pipeline stage. Suppose that $L(P)$ represents the corresponding pipeline stage of place P and $n(P)$ represents the number of instances of such place. For conversion, we need one resource place P' to represent the resource $L(P)$ and $k = n(P)$ places, $P_1 \dots P_k$, to represent instances of P in the RCPN model. The initial number of tokens in place P' in CPN is equal to the capacity of place $L(P)$ in RCPN. All instances of the *end* places in RCPN are replaced with one *end* place in CPN. Initially, this place holds as many tokens as the maximum possible number of concurrent instructions in the pipeline. This place also provides an input arc for the initial transition of the RCPN model. Fig. 8 shows how extra arcs must be added to model the shared resource.

B. Converting Arc Priorities

If G_i is the guard condition of a transition connected to an arc with priority i, the guard condition of this transition in the CPN model G' is defined as follows:

$$G' = G_i \wedge not(G_{i-1}) \wedge \cdots \wedge not(G_0).$$

C. Converting Reservation Tokens

A reservation token in RCPN occupies a place P to disable its input transitions by reducing the capacity. In CPN, this should be done by consuming tokens. As we explained in Section V-A, for every place P, there will be a place P' in the CPN that represents the corresponding pipeline stage of P. Therefore, whenever a reservation token is sent to a place P in an RCPN model, in the corresponding CPN model, a token must be consumed from the corresponding place P'.

One special case is when in an RCPN model a transition sends a reservation token to its own input place. This means that, when transition is fired, it consumes a token from its input place, and therefore, the capacity of the place increases. At the same time, the transition also sends a reservation token to its input place, and therefore, the capacity of the place must decrease. Considering the conversion rule of places, in the corresponding CPN model, such a transition will neither consume nor generate a token for the resource place. Fig. 9(a) shows an example of the general case, and Fig. 9(b) shows an example of the special case.

D. Converting RegRef Interfaces

Instead of using a token-based mechanism, RCPN models data hazards by its model of registers and register references. When converting to CPN, the interfaces of registers and register references in RCPN must be converted to proper token manipulations (token production/consumption).

CPN provides a generic mechanism to assign types and values to the tokens and to use and modify them through conditions and expressions. More specifically, arcs can have expressions that determine what type of tokens and with what value these tokens can pass through that arc. This can be used to manipulate the tokens of specific registers. To convert the register accesses in RCPN to CPN, we need to perform the following steps.

1) We create a place RF in CPN for the register file and initialize it with the proper number and type of tokens. For each register in the system, there will be one token initially stored in the register file.
2) A *canRead* condition and a *read* method always appear together in a transition. For each *canRead* that appears in the guard condition of a transition, we add an arc from the RF place to the corresponding transition in the CPN, and for each *read* in the transition, we add an arc from the corresponding transition in CPN to the RF place. These arcs have proper expressions that determine the type and the value of the register token that is being used. Fig. 10(a) and (b) shows examples of this conversion.

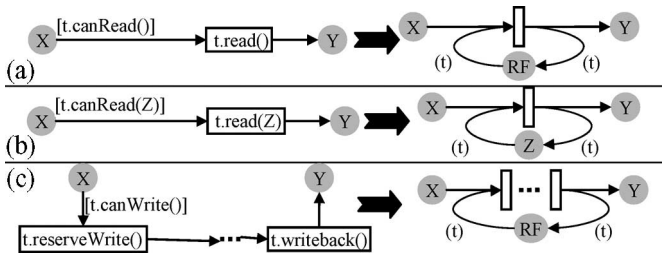

Fig. 10. Converting RegRef interfaces.

3) A *canWrite* in RCPN means that the corresponding transition in CPN checks for the availability of a register token in RF and a *reserveWrite* means that the transition CPN consumes that register token from RF. The token may not be returned back to the register file until its value is written back. Therefore, a *writeback* in RCPN means that the corresponding transition returns the register token to RF. Fig. 10(c) shows an example of this conversion.

The actual implementation of RegRefs also depends on whether the processor is implementing register renaming or not. In case of register renaming, the conversion of RCPN to CPN requires to carefully define token types, arc variables, etc. It is rather a complex process and requires a good understanding of CPN concepts that can be found in [1].

VI. Cycle-Accurate Simulation

RCPN can generate very fast cycle-accurate simulators. Like any other Petri net model, an RCPN model can be simulated by determining the enabled transitions and executing them concurrently. Searching for enabled transitions and handling concurrency can be very time consuming in generic Petri net models, especially if there are too many places and transitions in the design. However, a more careful look at the RCPN model reveals some of its properties that can be utilized to simplify these two tasks and speed up the simulation significantly.

Of the two groups of tokens in RCPN, reservation tokens carry no data and are used only to show unavailability of resources. Since transitions represent the functionality of an instruction between two pipeline stages, reservation tokens alone cannot enable them. Therefore, only places that have an instruction token may have an enabled output transition. While a place may be connected to many transitions in different subnets, an instruction token only goes through transitions of the subnet corresponding to its type. In other words, based on the type of an instruction token, only a subset of output transitions of a place may be enabled. Since the structure of the RCPN model is fixed during simulation, for every place and instruction type, the list of transitions that may be enabled can be statically extracted from the model before simulation begins. This list is sorted based on the priorities of output arcs of the place and is processed accordingly. Therefore, because of this property of RCPN, not only can we significantly reduce the overhead of searching for enabled transitions but we can also generate a compiled simulator for an RCPN model of a processor.

For every instruction-dependent subnet of an RCPN, we generate a C++ template class. This class stores the values of instruction symbols and has a function for each place that the instruction (token) can reside in. These places are, in fact, the places in that subnet as well as some of the places in the instruction-independent subnet of the model. At run time, the first time that an instruction is executed, a customized instance of the template class is created and stored in a software cache. A pointer to this object is used in the instruction token. During simulation, in every clock cycle, the instruction tokens of a place are processed by calling the proper function of template class that correspond to the place where the instruction is stored. Fig. 11 shows the template class that implements the Integer Instruction subnet in the RCPN model in Fig. 5. The functions $L1$, $L2$, and $L3$ correspond to the places that the instruction token can reside in. In each of these functions, the firing mechanism of the output transitions of the corresponding place is implemented. The order at which the output transitions are processed is fixed and is determined by the priorities of output arcs. For example, in function $L1$ in Fig. 11, the first capacity of the output place $L2$ and the guard condition of the output arc with priority 0 is checked. If $L2$ can receive a token and the guard condition is true, then the code of the transition is executed, and the instruction token is moved from place $L1$ to place $L2$. Otherwise, if the guard condition of the output arc with priority 1 is true, the corresponding transition is fired. If none of the guard conditions are true, the instruction token remains in that place and is processed again in the next clock cycle. The parameters and symbols of the template class in Fig. 11 are defined based on the instruction description in Fig. 4. The details of defining and initializing these symbols during instruction decode is explained in [6].

Using C++ templates forces the compiler to apply aggressive partial evaluation optimizations on the instructions. Furthermore, each instruction is decoded only once, and the decoded instruction is stored in a software cache while its pointer is passed around. In this way, we avoid redecoding of instruction in different pipeline stages and in future executions of the instruction. These three optimizations (i.e., reducing the search for enabled transitions to only output transitions of a place in a subnet, using template classes to utilize partial evaluation, and caching the decoded instructions) significantly improve the simulation performance. In the rest of this section, we also explain how we reduce the overhead of handling concurrency.

In RCPN, enabled transitions execute in parallel; tokens are simultaneously read from input places at the beginning of a processor cycle and then, in parallel, written to the output places at the end of the cycle. Therefore, the simulator must ensure that the variables representing such places are all read before being written during a cycle. The usual and computationally expensive solution is to model such places using a two-list algorithm (similar to master/slave latches). This approach uses two token storages per place—one of them is read from and the other is written to in the course of a cycle. At the end of the cycle, the tokens in the written-to storage are copied to the read-from storage.

In general, we can ensure that all tokens from the previous cycle are read from before being written to by evaluating all places (or their corresponding pipeline stages) in reverse topological order. Therefore, only very few places that are

```
template <class T1, class T2, class T3> class IntegerInstructions
{
public:
  T1 op;
  T2 d, s1;
  T3 s2;
  …
  void L1()  //called when instruction is in place L1
  {
    if ( PLACE_HAS_CAPACITY(L2) && s1.canRead() &&
      s2.canRead() && d.canWrite())
    {
      s1.Read();
      s2.Read();
      d.reserveWrite();
      CHANGE_PLACE(L1, L2);
    }
    else if ( PLACE_HAS_CAPACITY(L2) &&
          s1.canRead(PLACE(L3)) && s2.canRead() &&
          d.canWrite())
    {
      s1.Read(PLACE(L3));
      s2.Read();
      d.reserveWrite();
      CHANGE_PLACE(L1, L2);
    }
  }

  void L2()  //called when instruction is in place L2
  {
    if (PLACE_HAS_CAPACITY(L3))
    {
      d = op(s1, s2);
      CHANGE_PLACE(L2, L3);
    }
  }

  void L3()  //called when instruction is in place L3
  {
    if (PLACE_HAS_CAPACITY(END)) //END may have unlimited
capacity
    {
      d.writeBack();
      CHANGE_PLACE(L3, END);
    }
  }
}
```

Fig. 11. Template class implementing the Integer Instruction subnet in Fig. 5.

referenced in a circular way, usually because of feedback paths like state L_3 in Fig. 5, need to implement a two-list algorithm. The resulting code is considerably faster since it avoids the overheads of managing two storages in the two-list algorithm. Note that in CPN, this well-known optimization is not applicable because all the resource sharings are modeled with circular loops.

Fig. 12 shows the main body of our simulation engine. In the main loop, after updating the places that implement the two-list algorithm, all places are processed in reverse topological order. At the end of each iteration, the instruction-independent subnet of the model, which is responsible for fetching instructions and generating the instruction tokens, is executed.

VII. Experiments

To evaluate the RCPN model, we modeled both StrongArm [27] and XScale [3] processors using the ARM7 instruction set. StrongArm has a simple five-stage pipeline. XScale is an in-order-execution out-of-order-completion processor with

```
//main body of simulation engine
 P = list of places in reverse topological order;
 while program not finished
   foreach place p in {places that implement two-list algorithm}
     mark written tokens as available for read in p;
   endfor
   foreach place p in P
     foreach instruction token t in place p
     call t.p();
     endfor
   endfor
   execute the instruction independent subnet of RCPN;
   increment cycle count;
 Endwhile
//end main body
```

Fig. 12. Main body of simulation engine.

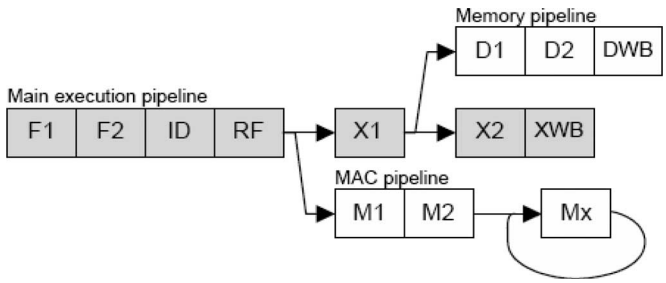

Fig. 13. XScale pipeline.

a pipeline structure shown in Fig. 13. The ARM instruction set was implemented using six operation classes describing instruction symbols and their binary coding. Except for the operation classes, it took only one man-day for StrongArm and only three man-days for XScale to develop both the RCPN models and the simulators.

To evaluate the performance of the simulators, we chose benchmarks from the MiBench [29] (blowfish, crc), MediaBench [30] (adpcm, g721), and SPEC95 [31] (compress, go) suites. These benchmarks were selected because they use very few simple system calls (mainly for I/O) that should be translated into host operating system calls in the simulator. We used *arm-linux-gcc* to generate the binary code of the benchmarks. The compiler only uses ARM7 instruction set, and therefore, we only needed to model those instructions. The simulators were run on a Pentium 4/1.8 GHz/512 MB RAM.

Fig. 14 compares the performance of the simulators generated from the RCPN model with that of SimpleScalarArm. The first bar for each benchmark shows the performance of the SimpleScalarArm simulator. This simulator implements StrongArm architecture, and we disabled all the extra options and used simplest parameter values to improve simulation performance. On the average, this simulator executes 600 kHz. The second and third bar for each benchmark shows the performance of our simulator for XScale and StrongArm processor models, respectively. These simulators execute 8.2 and 12.2 MHz on the average. SimpleScalar uses a fixed architecture for any processor model. Therefore, the complexity and performance of the simulator are almost similar across different models. On the other hand, RCPN models are true to the modeled processor, and hence, the performance of the generated simulators may vary, depending on the complexity

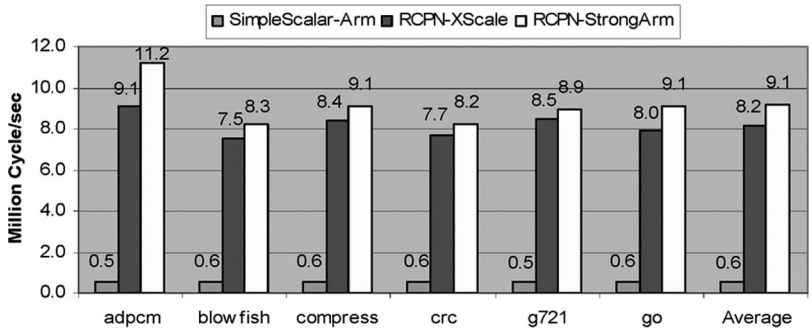

Fig. 14. Simulation performance (in megahertz).

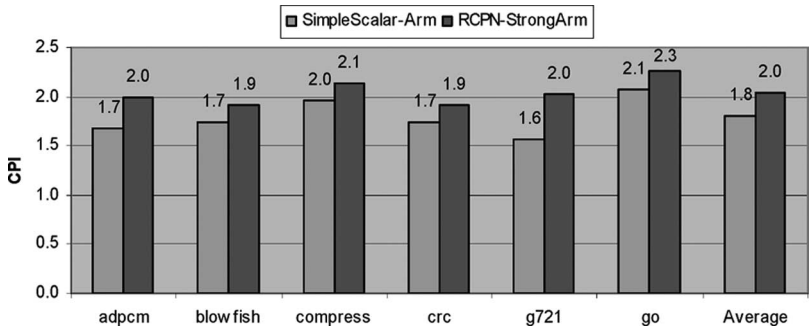

Fig. 15. Clock per instruction (CPI).

of the simulated processors. The StrongArm has a very simple pipeline structure compared to that of XScale. Therefore, not only are there fewer places and transitions in the RCPN model of StrongArm but instructions may also have far less number of possible states in the places or possible execution paths in the units (see Section VIII-A). In other words, the RCPN model of StrongArm is simpler than the RCPN model of XScale, and hence, the corresponding simulator runs slightly faster than that of XScale.

Fig. 15 shows the CPI values of SimpleScalarArm and our StrongArm simulator (RCPN-StrongARM). This figure shows that although our simulator runs significantly faster than SimpleScalar, the CPI values of the two simulators are almost similar ($\sim$10% difference).

SimpleScalarArm is based on a superscalar architecture simulator engine that assumes a typical five-stage pipeline processor architecture [40]. Its machine description is, in fact, a set of parameters that forces the engine to behave "as close as" possible to the target processor. However, since the pipeline structure of the simulator is not exactly the same as the target (i.e., the modeled processor), SimpleScalarArm itself may not be very accurate. In our simulator, the pipeline structure is, in fact, faithfully modeled with an RCPN model. In order to obtain a fair comparison of the two simulators, we set the parameters of both simulators so that the effects of different models becomes minimal (e.g., we disabled cache, branch prediction, etc. to simulate only the core processor). Clearly, adding the same peripheral models to both simulators should have a similar impact (i.e., in terms of performance and accuracy). Our main goal was to show that we can achieve much higher speed, and we included the CPI as an indication (but not absolute proof) that we modeled the processor correctly. Nevertheless, as SimpleScalarArm developers also mention in their documents, some of the microarchitectural details are not available in the manual and must be derived from other sources such as "compiler writers guide." Different interpretations of the information can also lead to variations in the models used in their simulator and ours. For example, disabling the branch predictor in SimpleScalar means that after fetching the branch instruction, program counter is incremented and the next instructions are fetched and later pipeline is flushed if needed. In other words, disabling the branch predictor in SimpleScalar means that branch instructions are *always* predicted as not taken. In our simulator, we stall the pipeline until the branch is executed. In this way, the number of executed instructions as well as CPIs may vary slightly between the two simulators.

The RCPN-based modeling approach does not impose any limitation on capturing instruction schedules. Therefore, by providing accurate models, the results of generated simulators can be fairly accurate. Such models are usually obtained by comparing the simulation results against a base simulator or the actual hardware and refining the model information.

VIII. Discussion

A. Model Complexity

Similar to general Petri net models, the complexity of RCPN directly depends on the number of places and transitions. A more complex RCPN can slow down the decoding and execution of instructions. As explained in Section III, each transition in an RCPN model describes one execution path in a unit for one class of instructions captured by a subnet. In the worst case, for n instructions in the instruction set and m units in the pipeline, we might have $n \times m$ transitions. This case

happens only if every instruction goes through every unit in the pipeline during execution. However, typically, the instruction set is divided into groups of instructions that go through only a subset of units in the pipeline. For example, in Fig. 5(b), only LoadStore instructions go through unit M, and therefore, there is only one transition for unit M in the LoadStore Instruction subnet. On the other hand, when an instruction goes through a unit, depending on the dynamic status of the pipeline, different functionalities may be executed for that instruction in that unit. For each of these functionalities, there may be a separate transition in the RCPN subnet to which the instruction belongs. For example, there are two separate transitions for unit D in the Integer Instruction subnet in Fig. 5(b). Similarly, in Fig. 7(b), there are four transitions that correspond to unit *Issue*. In general, the upper bound of the number of transitions is given by

$$\#\text{transitions} \leq \sum_{i \in \text{InstructionClasses}} \sum_{u \in \text{Units}} \text{number\_of\_execution\_paths}(i, u).$$

A place in RCPN shows the state of an instruction in a pipeline stage. For example, in Fig. 5(a), after an instruction is decoded, it goes to pipeline stage L_2, and it has only one state: *decoded*. In other words, only one place in each subnet is enough to capture the sate of an instruction after going through any transition corresponding to unit D. On the other hand, in Fig. 7, an instruction may reside in pipeline stage L_2 while being in one of the following four possible states: 1) both operands are available; 2) only the left operand is available; 3) only the right operand is available; and 4) none of the operands are available. Correspondingly, there are four places in Fig. 7(b), representing the state of an instruction residing in stage L_2. In general, the upper bound of the number of places is given by the equation shown at the bottom of the page.

Modeling an instruction set architecture (ISA) involves two separate steps, namely: 1) modeling the binary decoding and 2) modeling the behavior of instructions. In [6] and [41], we have discussed the complexity of modeling the binary decoding of other processors such as SPARC and Pentium using a template-based approach. In addition to such binary code complexities, the instructions in complex instruction set computing or DSP architectures usually have a complex behavior as well. Typically, one instruction in such processors performs multiple operations. However, such behavior is also seen in the ARM ISA, where, for example, a single instruction may check a condition, set flag bits, perform an arithmetic operation, and shift the result. Since RCPN is based on CPN, it is general enough to model any kind of instruction behavior. In this paper, we also showed the effect of different microarchitectural features on the modeling effort. We expect that for a DSP-like architecture, the modeling effort should be similar.

B. RCPN Versus Other Comparable Approaches

We could not compare the speed of our simulators with any LISA simulators because we did not have access to either a LISA simulator or any published results comparing LISA with a publicly available simulator that could be used as a common comparison point. In terms of modeling capabilities, in addition to the architectures that can be captured in LISA, RCPN also covers out-of-order processors. To the best of our knowledge, LISA does not provide any formal mechanism for modeling processors. In contrast, RCPN provides a formal yet intuitive modeling approach.

From the modeling point of view, the RCPN and OSM models are comparable. However, OSM uses very few FSMs, e.g., only one FSM for StrongARM, and captures the pipeline through these FSMs and TMI software components. RCPN uses multiple subnets, each equivalent to an OSM, to explicitly capture the pipeline control. For example, there are six RCPN subnets in the StrongArm model. Only for capturing data hazards, RCPN relies on the fixed interface software components. Therefore, a larger part of processor behavior is captured formally in RCPN than in OSM. In other words, the nonformal part of OSM model (i.e., TMIs) is large enough that it needs a separate event-driven simulation engine; but the nonformal part of the RCPN model is a set of very simple functions for accessing registers. Nevertheless, RCPN-based simulators run an order of magnitude faster than OSM-based ones.

Our simulators are as fast as FastSim although we use simple but very effective optimizations and FastSim uses the very complex fast-forwarding technique.

On the average, our XScale simulator runs 15 times faster than SimpleScalar. We can summarize the causes of this high performance as follows.

1) Because of RCPN features, we can reduce the overheads of supporting concurrency and searching for enabled transitions.
2) We apply partial evaluation optimization to customize the instruction-dependent subnets for each instruction instance and, hence, improve their performance.
3) In RCPN, when an instruction token is generated, the corresponding instruction is decoded and stored in the token. Since the token carries this information, we do not need to redecode the instruction in different pipeline stages to access its data. Furthermore, the tokens are cached for later reuse in the simulator.

C. Debugging and Extending RCPN Models

Since the simulator is generated automatically, debugging of the implementation of the simulator is (eventually) eliminated. Only the model itself must be debugged/verified. As described in [6], using operation classes and templates makes debugging

$$\#\text{places} \leq \sum_{i \in \text{InstructionClasses}} \sum_{p \in \text{PipelineStages}} \text{number\_of\_possible\_states}(i, p)$$

much simpler. Since RCPN is formal and can be converted to standard CPN, formal methods can be used for analyzing the models as well.

Depending on the type of extension, RCPN models can be typically reused and easily extended. For example, when adding a new instruction, if the flow of instruction in the processor is already modeled by an instruction-dependent subnet, then only the symbols and their binary decoding needs to be added to the corresponding operation class of the subnet. If the flow of the new instruction is not already modeled, then a new subnet can be easily added to an existing RCPN model with minimum interaction with other parts of the model. Depending on the situation, adding a new functional unit can be as easy as adding new places in proper subnets and connecting them to proper transitions. As a final example, in order to extend the model of an out-of-order processor from three-way to four-way, first a proper instruction-dependent subnet must be added to the model, either by duplicating an available subnet or adding a new one. Then, the instruction-independent subnet must be updated to fetch more instructions and the capacity of the corresponding pipeline stages must be increased to account for the extra fetched instructions.

It is also possible to extend the transitions of an RCPN processor model to include call-back functions to other tools. For example, the application programming interface of a profiler can be included in the transitions to collect information such as component utilization, I/O values of units, and execution frequency of different types of instructions. Furthermore, a call back to a software debugger can temporarily transfer the control to another software tool that monitors the state of the processor and possibly report it to the user.

D. Other Benefits of RCPN

As it was explained in Section I, the first step in generating an efficient retargetable simulation framework is to have an efficient modeling approach. In this paper, we showed that RCPN is an efficient and intuitive modeling approach for capturing pipelined processors. To create a complete retargetable framework, in the next step, we need to develop a language that describes the processor based on RCPN and also develop the set of tools for analyzing the description and generating the simulation source code. Since RCPN is formal, the corresponding processor descriptions can also be used for automatic test generation and verification of pipeline structure.

IX. Conclusion

In this paper, we presented the RCPN for capturing pipelined processors and generating fast cycle-accurate simulators. Conceptually, RCPN is similar to other Petri net models but is far less complex. It has two major contributions, namely: 1) it provides an accurate, generic, efficient, and formal way for modeling processors and 2) it generates very high-performance cycle-accurate simulators. RCPN models are very intuitive to generate because they are similar to the pipeline block diagram of the processor. Our cycle-accurate simulators, for both StrongArm and XScale processors, run about 15 times on the average faster than SimpleScalar for ARM, although XScale has a relatively complex out-of-order pipeline.

The use of CPN concepts in RCPN makes it very suitable for different design analysis and verification purposes. The clean distinction between different types of tokens and data hazard mechanism in addition to the structure of RCPN can be used to extract the necessary information for deriving retargetable compilers. The future direction of our research is to address these issues as well as extract fast functional simulators from the same detailed RCPN models.

References

[1] K. Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. New York: Springer-Verlag, 1997.

[2] The SimpleScalar project homepage. [Online]. Available: http://www.simplescalar.com

[3] *Intel XScale Microarchitecture for the PXA250 and PXA210 Applications Processors, User's Manual*, Intel Co., Feb. 2002. [Online]. Available: http://www.intel.com/design/intelxscale/

[4] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proc. ISCA*, Jun. 2004, pp. 338–349.

[5] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation: A technique for fast and flexible instruction set simulation," in *Proc. DAC*, Jun. 2003, pp. 758–763.

[6] ——, "An efficient retargetable framework for instruction-set simulation," in *Proc. Int. Conf. Hardware/Software Codesign and Syst. Sysnthesis*, Oct. 2003, pp. 13–18.

[7] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. DATE*, Mar. 1995, pp. 503–507.

[8] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *Proc. DAC*, Jun. 1997, pp. 299–302.

[9] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Proc. DATE*, Mar. 1999, pp. 485–490.

[10] P. Grun, A. Halambi, N. Dutt, and A. Nicolau, "RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions," in *Proc. ISSS*, 1999, pp. 44–50.

[11] V. Rajesh and R. Moona, "Processor modeling for hardware software codesign," in *Proc Int. Conf. VLSI Des.*, Jan. 1999, pp. 132–137.

[12] F. Engel, J. Nührenberg, and G. P. Fettweis, "A generic tool set for application specific processor architectures," in *Proc. Int. Workshop Hardware/Software Codesign (CODES)*, May 2000, pp. 126–130.

[13] G. Zimmerman, "The MIMOLA design system: A computer-aided processor design method," in *Proc. DAC*, 1979, pp. 53–58.

[14] J. Teich and R. Weper, "A joined architecture/compiler environment for ASIPs," in *Proc. Int. Conf. CASES*, 2000, pp. 26–33.

[15] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan, "Asim: A performance model framework," *IEEE Computer*, vol. 35, no. 2, pp. 68–76, Feb. 2002.

[16] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *Proc. Int. Symp. Microarchitecture*, Nov. 2002, pp. 271–282.

[17] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Boston, MA: Kluwer, 2002.

[18] W. Mong and J. Zhu, "A retargetable micro-architecture simulator," in *Proc. DAC*, Jun. 2003, pp. 752–757.

[19] S. Onder and R. Gupta, "Automatic generation of microarchitecture simulators," in *Proc. IEEE Int. Conf. Comput. Languages*, 1998, pp. 80–89.

[20] F. S. Chang and A. Hu, "Fast specification of cycle-accurate processor models," in *Proc. ICCD*, 2001, pp. 488–492.

[21] I. Park, S. Kang, and Y. Yi, "Fast cycle-accurate behavioral simulation for pipelined processors using early pipeline evaluation," in *Proc. ICCAD*, 2003, pp. 138–141.

[22] E. Schnarr and J. R. Larus, "Fast out-of-order processor simulation using memoization," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Syst.*, 1998, pp. 283–294.

[23] E. Schnarr, M. Hill, and J. R. Larus, "Facile: A language and compiler for high-performance processor simulators," in *Proc. Conf. PLDI*, Jun. 2001, pp. 321–331.

[24] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "LISA—Machine description language for cycle-accurate models of programmable DSP architectures," in *Proc. DAC*, 1999, pp. 933–938.
[25] W. Qin and S. Malik, "Flexible and formal modeling of microprocessors with application to retargetable simulation," in *Proc. DATE*, Mar. 2003, pp. 556–561.
[26] J. Hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach*, 3rd ed. San Mateo, CA: Morgan Kaufman, 2003.
[27] Digital Equipment Corporation, *Maynard, Digital Semiconductor SA-110 Microprocessor Technical Reference Manual*, 1996.
[28] R. Razouk, "The use of Petri Nets for modeling pipelined processors," in *Proc. DAC*, 1988, pp. 548–553.
[29] MiBench Embedded Benchmark Suit homepage. [Online]. Available: http://www.eecs.umich.edu/mibench/
[30] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. Int. Symp. Microarchitecture (MICRO)*, 1997, pp. 330–335.
[31] The Standard Performance Evaluation Corporation (SPEC) benchmarks homepage. [Online]. Available: http://www.spec.org
[32] R. Leupers, J. Elste, and B. Landwehr, "Generation of interpretive and compiled instruction set simulators," in *Proc. ASP-DAC*, Jan. 1999, pp. 339–342.
[33] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr, "A universal technique for fast and flexible instruction-set architecture simulation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 12, pp. 1625–1639, Dec. 2004.
[34] V. Zivojnovic, S. Pees, and H. Meyr, "LISA—Machine description language and generic machine model for HW/SW co-design," in *Proc. IX VLSI Signal Process.*, 1996, pp. 127–136.
[35] The MicroLib project homepage. [Online]. Available: http://www.microlib.org/
[36] D. G. Pérez, G. Mouchard, and O. Temam, "MicroLib: A case for the quantitative comparison of micro-architecture mechanisms," in *Proc. Int. Symp. Microarchitecture (MICRO)*, 2004, pp. 43–54.
[37] J. Zhu and D. D. Gajski, "An ultra-fast instruction set simulator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 10, no. 3, pp. 363–373, Jun. 2002.
[38] G. Beltrame, C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, and V. Trianni, "Modeling assembly instruction timing in superscalar architectures," in *Proc. ISSS*, 2002, pp. 132–137.
[39] ——, "Dynamic modeling of inter-instruction effects for execution time estimation," in *Proc. ISSS*, 2001, pp. 136–141.
[40] The SimpleScalar Processor Architecture. [Online]. Available: http://www.simplescalar.com/docs/simple_tutorial_v4.pdf
[41] M. Reshadi, P. Mishra, and N. Dutt, "A retargetable framework for instruction-set architecture simulation," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 5, no. 2, pp. 431–452, May 2006.

Mehrdad Reshadi received the B.S. (hons.) degree in computer engineering from Sharif University of Technology, Tehran, Iran, in 1997, and the M.S. degree in computer engineering from the University of Tehran, Tehran, in 2000. He is currently working toward the Ph.D. degree at the Center for Embedded Computer Systems, University of California, Irvine.

He is also with the Donald Bren School of Information and Computer Science, University of California. His research interests include embedded system design automation, microprocessor modeling and simulation, computer architecture, high-level synthesis, design specification techniques, and specification-based compilation and simulation.

Mr. Reshadi received the Best Paper Award at the International Conference on Hardware/Software Codesign and System Synthesis in 2003.

Bita Gorjiara received the B.S. and M.S. degrees in computer engineering from the University of Tehran, Tehran, Iran, in 1998 and 2001, respectively. She is currently working toward the Ph.D. degree at the Center for Embedded Computer Systems, University of California, Irvine.

She is also with the Henry Samueli School of Engineering, University of California. Her research interests include low-power custom processors design, multicore system design, on-chip communication, and design automation of embedded systems.

Nikil D. Dutt (S'82–M'89–SM'96) received the Ph.D. degree in computer science from the University of Illinois, Urbana-Champaign, in 1989.

He is currently a Professor of computer science and electrical engineering and computer science with the University of California, Irvine (UCI) and is affiliated with the following centers at UCI: Center for Embedded Computer Systems, Center for Pervasive Communications and Computing, and California Institute for Telecommunications and Information Technology. His research interests are embedded systems design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems.

Dr. Dutt received the best paper awards at CHDL89, CHDL91, VLSIDesign2003, CODES+ISSS 2003, and ASPDAC-2006. He currently serves as the Editor-in-Chief of the Association for Computing Machinery (ACM) Transactions on Design Automation of Electronic Systems and as an Associate Editor of the ACM Transactions on Embedded Computer Systems. He was an ACM SIGDA Distinguished Lecturer during 2001–2002 and an IEEE Computer Society Distinguished Visitor for 2003–2005. He has served on the steering, organizing, and program committees of several premier CAD and embedded system design conferences and workshops, including ASPDAC, CASES, CODES+ISSS, DATE, ICCAD, ISLPED, and LCTES. He also serves or has served on the advisory boards of ACM SIGBED and ACM SIGDA and is the Vice-Chair of IFIP WG 10.5.