Electronic Notes in Theoretical Computer Science 70 No. 6 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 16 pages

Applying ELAN Strategies in Simulating Processors over Simple Architectures [1]

Mauricio Ayala-Rincón [2]

Departamento de Matemática, Universidade de Brasília
Brasília D.F., Brasil

Rinaldi Maya Neto, Ricardo P. Jacobi [3]

Departamento de Ciência da Computação, Universidade de Brasília
Brasília D.F., Brasil

Carlos H. Llanos [4]

IESB, Brasília D.F., Brasil

Reiner W. Hartenstein [5]

Fachbereich Informatik, Universität Kaiserslautern
Kaiserslautern, Germany

Abstract

The simulation of processors over simple architectures is important for enabling test and verification prior to the expensive implementation involved in the development of new hardware technologies. Arvind's group has illustrated how to describe processors by term rewriting systems and has introduced a technique for proving the correctness of specifications for elaborated processors with respect to basic ones. They propose that the described processors should be simulated over standard hardware description languages such as Verilog, after translating these rewrite descriptions adequately, and not directly over the rewriting specifications. In this work we show how rewriting-logic may be applied for purely rewriting based specification as well as simulation of processors. Furthermore, we show how rewriting based simulation may be used for evaluating the performance of important hardware aspects of processors. Rewriting-logic environments such as ELAN, the one we use here, are sufficiently versatile to allow for adequate specifications and simulations which through easy modifications of the strategies enable a dynamic verification of aspects intrinsically related to hardware properties such as the size and control of reorder buffers and the method of predictions used by speculative processors.

Keywords: *Rewriting strategies and logic, hardware design and verification.*

1 Introduction

In recent years some work on applying rewriting techniques to the design of hardware has been developed. In particular, Arvind's group at MIT has treated the implementation of processors over simple architectures [12,13,1], rewrite based description and synthesis of simple logical digital circuits [8] and description of cache protocols over memory systems [14,16]. Their work has made evident the great capacity and possibilities of rewriting as an effective framework for dealing with simulation and estimation of hardware before expensive physical implementations are done. In this work we discuss the advantages and difficulties of a real rewrite based simulation of descriptions of processors over simple architectures. For this purpose we use the well-known rewriting-logic environment ELAN [5,4].

Our work, as that of Arvind group's in [12,13,1], is focused on the implementation of processors over the $\mathcal{AX}$ RISC architecture. Rules for the processors are specified in the ELAN system and different architectural components such as memory, registers, etc. are discriminated in a natural way, taking advantage of this typed language. Proving the soundness of the processors is thus reduced to proving that they simulate and are simulated by a basic processor. In our ELAN approach the separation between logic and rewriting allows us to define rules for the instructions of the processors and to specify strategies describing architectural characteristics as the size of reorder buffers - ROBs. Unlike the approach of Arvind's group, in our implementations we can simulate the execution of assembly description programs over our rewriting-specified processors; for instance, generation of the Fibonacci sequence, quick-sort, computation of the Knuth-Morris-Pratt jump function, etc., while dynamically changing strategies for estimating the most adequate form of implementing these architectural components. This is all done without translating the rewriting specifications into hardware description languages as suggested by the approach of Arvind's group. After a simulation is performed, these estimations are given by an analysis of the ELAN statistics for the number of times each rewriting rule (i.e. processor instruction) is applied. Other important architectural aspects such as predictions in processors with speculative execution are implemented in their own rewriting rules. Rewrite based simulation of programs in assembly description does not correspond exactly to the execution of these programs over real architectures, since many additional

[1] Work supported by the Brazilian/German cooperation of the CAPES/DFG foundations.
[2] Partially supported by the FEMAT Brazilian foundation. `ayala@mat.unb.br`
[3] `{rinaldi,rjacobi}@cic.unb.br`
[4] `llanos@unb.br`
[5] `hartenst@rhrk.uni-kl.de`

steps are executed in a rewriting based system; in particular, many unsuccessful attempts to apply rewriting rules slow down the simulation. However, ELAN statistics allow for a concrete estimation of how these processors should work in real hardware implementations.

Additionally, we point out some interesting problems inherent in the way rewriting rules (and strategies) are applied in true purely rewrite based systems. In particular, that rules are not naturally applied in a non-deterministic manner; they are selected as the first applicable rule found in the order the rules are defined and applied in the first positions (left-most, inner-most or similarly) that they match over the target term. In our implementation these problems arise when important architectural aspects as *out-of-order execution* of instruction templates over ROBs are to be simulated. Although our implementations are deterministic we comment on how one can overcome these problems in a non-purely rewriting system like ELAN, where some non-deterministic strategies are available.

2 Architecture and processors descriptions

We assume familiarity with the basic concepts of computer architecture and rewriting theory as presented respectively in [7] and [3]. Additionally, we suppose the reader familiar with rewriting-logic environments like ELAN. We briefly describe the $\mathcal{AX}$ RISC architecture and the specifications in ELAN of a basic processor over this architecture and a more elaborated one that allows for speculative execution over a reorder buffer.

2.1 The $\mathcal{AX}$ RISC architecture

$\mathcal{AX}$ is a set of RISC instructions where all memory access is done by *load* and *store* instructions and the arithmetic operations are done over the registers at the register file (*rf*). A sequence of instructions that describes a program is placed at the instruction memory (*im*). The instructions are executed in-order and after each instruction execution the contents of the program counter (*pc*) is incremented by one except for branch instructions (*Jz*).

The set of different instructions of $\mathcal{AX}$, $INST$, is described as:

$$\begin{array}{rcllll} INST & \equiv & r := Loadc(v) \;\| & r := Loadpc & \| & r := Op(r_1, r_2) \;\| \\ & & Jz(r_1, r_2) \quad\;\; \| & r := Load(r_1) & \| & Store(r_1, r_2) \end{array}$$

The load-constant instruction, $r := Loadc(v)$, puts the constant v into the register r. The load-program-counter instruction, $r := Loadpc$, puts the content of the program counter into the register r. The arithmetic-operation instruction, $r := Op(r_1, r_2)$, performs the (abstract) arithmetic operation specified by Op on the operands specified by the registers r_1 and r_2 and puts the result into the register r. The branch instruction $Jz(r_1, r_2)$, sets the program counter to the target instruction address specified by the register r_2 when the

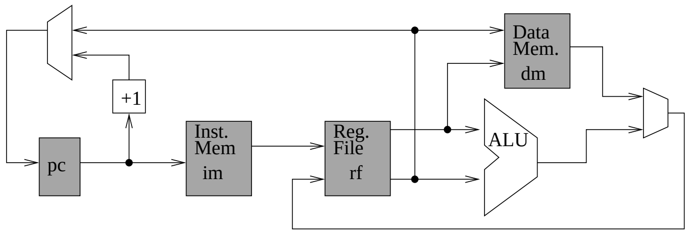

Fig. 1. Description of the basic processor

contents of the register r_1 is zero and increments the program counter by one otherwise. The load instruction, $r := Load(r_1)$, loads the memory cell specified by the register r_1 into the register r. The store instruction, $Store(r_1, r_2)$, stores the contents of the register r_2 into the memory cell specified by the register r_1.

2.2 Basic processor

The operational semantics of the $\mathcal{AX}$ RISC instruction set is defined by a single-cycle, non pipelined, in-order execution processor that we will call the **basic processor**. See figure 1 for a description in register transfer level.

The description of the system `Sys` is given by its memory `m` and processor `Proc`: `Sys(m,Proc)`, the latter consists of the instruction address `ia` of the *pc*, the register file `rf` and the program `prog`: `Proc(ia,rf,prog)`.

The rewriting rules implementing the $\mathcal{AX}$ instructions in ELAN are given in the Table 1. This follows straightforwardly from the operational semantics given earlier of these instructions. We explain the most complex of these rules: the branch instruction *Jz*. All other rules are similarly explained.

Whenever the current instruction of the program `prog` at the position (of the instruction memory *im*) given by the instruction address `ia` is a branch instruction of the form `Jz(r1,r2)`, the program counter should be changed either by the contents of the register `r2` or by `ia+1`. The former, in the case that the contents of the register `r1` equals zero (checked by `valueofReg(r1,rf) == 0`); the last, otherwise (checked by `valueofReg(r1,rf) != 0`). The auxiliary premise `isinstJz(selectinst(prog,ia))` checks whether the current instruction is a branch. The role of the "`where _ :=() _`" commands is to set auxiliary variables.

2.3 Implementation of a processor with speculative execution over a ROB

As in [12,13,1] more sophisticated processors may be described by rewriting rules and then proved *correct* by showing that they are simulated by the basic processor and simulate the basic processor. Here we describe the implementation of a processor that does speculative execution over a ROB. See figure 2. A ROB holds instructions that have been decoded but have not completed their execution. Conceptually, the ROB divides the processor into two asyn-

```
[Loadc] Sys(m,Proc(ia,rf,prog)) => Sys(m,Proc(ia+1,insertRF(rf,r,v),prog))
     where instIa :=() selectinst(prog,ia) if isinstLoadc(instIa)
     where r :=() nameofLoadc(instIa)
     where v :=() valueofLoadc(instIa) end
[Loadpc] Sys(m,Proc(ia,rf,prog)) =>
 Sys(m,Proc(ia+1,insertRF(rf,r,ia),prog))
     where instIa :=() selectinst(prog,ia) if isinstLoadpc(instIa)
     where r :=() nameofLoadpc(instIa) end
[Op] Sys(m,Proc(ia,rf,prog)) => Sys(m,Proc(ia+1,insertRF(rf,r,v),prog))
     where instIa :=() selectinst(prog,ia) if isinstOp(instIa)
     where r1 :=() reg1ofOp(instIa) where r2 :=() reg2ofOp(instIa)
     where r :=() nameofOp(instIa) where v:=() valueofOp(r1,r2,rf) end
[Jz] Sys(m,Proc(ia,rf,prog)) => Sys(m,Proc(nia,rf,prog))
     where instIa :=() selectinst(prog,ia) if isinstJz(instIa)
     where r1:=() reg1ofJz(instIa) where r2:=() reg2ofJz(instIa)
     choose try where nia:=()ia+1 if valueofReg(r1,rf)!=0
            try where nia:=()valueofReg(r2,rf) if valueofReg(r1,rf)==0
     end end
[Load] Sys(m,Proc(ia,rf,prog)) =>
 Sys(m,Proc(ia+1,insertRF(rf,r0,v0),prog))
     where inst :=() selectinst(prog,ia) if isinstLoad(inst)
     where r0 :=() nameofLoad(inst) where v0 :=() getMem(inst,rf,m) end
[Store] Sys(m,Proc(ia,rf,prog)) =>
 Sys(insertMEM(m,valueofReg(rA,rf),valueofReg(rB,rf)),Proc(ia+1,rf,prog))
     where inst :=() selectinst(prog,ia) if isinstStore(inst)
     where rA :=() nameofStoreR1(inst)
     where rB :=() nameofStoreR2(inst) end
```

Table 1
Rewriting rules for the basic processor

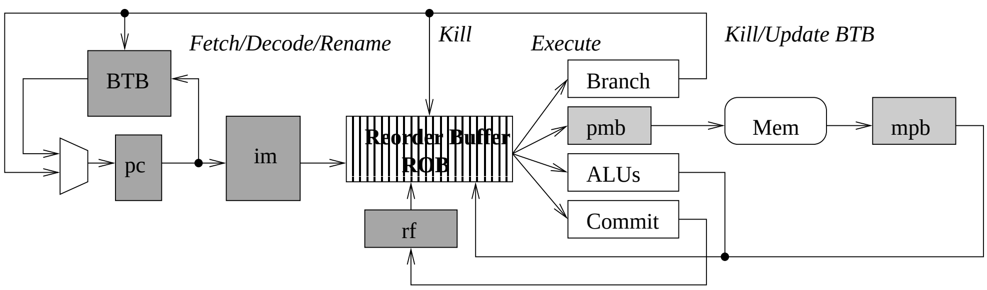

Fig. 2. Description of the speculative processor

chronous parts. The first one fetches the instruction and after decoding and renaming registers, dumps it into the next available slot in the ROB. The ROB slot index serves the purpose of the renaming tag, and the instruction templates in the ROB (ITB) always contain tags or values instead of register names. An instruction template in the ROB can be solved ("executed") if all its operands are available. The second part takes any enabled instruction

[PsOp]
```
Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-Op(v,v1),wf,sf).itbs2,btb,prog)) =>
Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-execOponval(v,v1),wf,sf).itbs2,
btb,prog)) end
```
[PsValueForward]
```
Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-v,wf,sf).itbs2,btb,prog)) =>
Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-v,wf,sf).ValueForward(t(k),v,itbs2),
btb,prog)) if TagExists(t(k),itbs2) end
```
[PsValueCommit]
```
Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-v,Wreg(r),sf).itbs2,btb,prog)) =>
Sys(m,Proc(ia, insertRF(rf,r,v),itbs2,btb,prog))
    if not TagExists(t(k),itbs2) end
```

Table 2
Arithmetic Operation and Value Propagation Rules

out of the ROB and dispatches it to an appropriate functional unit, including the memory system (then "execution" is completed). This mechanism is very similar to the execution mechanism in data flow architectures. Such an architecture may execute instructions out-of-order, especially if functional units have different latencies or there are data dependencies between instructions. Additionally, speculative execution of instruction is allowed. The speculative mechanisms predicts the address of the next instruction to be issued based on the past behavior of the programs. The address of the speculative instruction is determined by consulting a table known as the branch target buffer - BTB, which can be indexed by the current content of the program counter. If the prediction turns out to be wrong, the speculative instruction and all the instructions issued thereafter are abandoned and their effect on the processor state nullified. The BTB is updated according to some prediction scheme after each execution branch resolutions.

As for the basic processor the system `Sys` is described by its memory `m` and processor `Proc`: `Sys(m,Proc)`. But in contrast, the processor consists of the `ia` of the program counter, the register file `rf`, the program `prog` as well as of the ITB (`itb`) and the BTB (`btb`): `Proc(ia,rf,itb,btb,prog)`.

In the sequel we present the ELAN implementation of the rules of the speculative processor and after that we explain the operational semantics of some of these rules. The rules are divided into four classes: arithmetic and value propagation rules; instruction issue rules; branch completion rules and memory access rules. These sets of rewriting rules are presented in the Tables 2, 3, 4 and 5, respectively. Instead of the symbol "`:=`", that is reserved in ELAN, setting in the issued rules at the ITB is denoted by "`|-`".

Arithmetic operation and value propagation rules (Table 2) deal with the computation of arithmetic operations (`PsOp`), the propagation of its results through the ITB (`PsValueForward`) and the exclusion of the instruction template from the ITB when the result had already been solved and committed to the register file (this means the renaming tag it addresses does not occur in the buffer anymore, what is decided by `TagExists(t(k),itbs)`). A value is only committed to the register file when the instruction referencing it is on the head of the ITB, this approach is conservative since it avoids the need to reconstruct the state of the register file in the event of wrong speculations.

Issue rules (Table 3) are those used for the issuing of the instructions which generate templates stored in the ITB. Branch completion rules (Table 4) are those which deal with the resolution of speculations. When a branch instruction is issued the processor has to know which will be the next instruction to be fetched. The next predicted instruction is indicated by the BTB that is an indexed table. Suppose that the program instruction at position `ia` is being issued, the next value of the program counter, called `pia`, is looked up in the BTB using as index the current program counter (`pia :=() getbtb(ia,pia)`) and then the execution resumes at the `pia` value. When the ITB element containing the branch instruction reaches the head of the ITB it is the time to check if the speculation was done correctly or if the processor needs to fix the mistake and restart the execution at the correct program counter value. In the last case, the remaining instructions already in the ITB should be ignored. The rules in the Table 4 deal with this issue. Exemplifying, suppose that the head of the ITB is of the form `ITB(ia,k,Jz(v,nia),wf,Spec(pia))`, the branch completion rule has to check whether the value `v` is zero or not and then, respectively, check whether either the speculated address `pia` coincides with `nia` or with `ia+1`. In this event the prediction has been proved correct and the execution resumes. Otherwise the program counter must be set, respectively, either to the value of `ia+1` or to the correct value of the branch represented by `nia`, depending on whether the wrong speculation was a no jump or a jump, and the ITB must be completely emptied because the remaining instructions should not be executed. These rules also control the updating of the BTB for dynamic speculation (through the rules which define `changebtb`).

The memory access rules (Table 5) `PsLoad` and `PsStore` deal with the ROB and the data memory communication. These rules are applied after the processor has resolved all values of the tags of the instruction templates stored in the ITB by the `PsLoadIssue` and `PsStoreIssue` issue rules, respectively.

2.4 Proving correctness of processors

One useful feature of this rewrite based specification of processors is the possibility of proving the correctness of the implementation of some instruction set describing a processor. This is done by showing that one implementation simulates another in regard of some observation function [12,13,1]. The main idea is to design a function that can extract all the programmer visible states; i.e., the program counter, the register file and the memory from the system. The proof of the correctness of our specification of the speculative processor, here given for completeness of the presentation, follows the lines in [12].

```
[PsLoadcIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,
 Proc(ia+1,rf,insEndITBs(ITB(ia,k,t(k)|-v,Wreg(r),NoSpec),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstLoadc(instIa)
     where r:=() nameofLoadc(instIa)
     where v:=() valueofLoadc(instIa)
     where k:=() lengthof(itbs)+1 end
[PsLoadpcIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,
 Proc(ia+1,rf,insEndITBs(ITB(ia,k,t(k)|-ia,Wreg(r),NoSpec),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstLoadpc(instIa)
     where r :=() nameofLoadpc(instIa)
     where k :=() lengthof(itbs)+1 end
[PsOpIssue]Sys(m,Proc(ia,rf, itbs,btb,prog)) => Sys(m,Proc(ia+1,rf,
 insEndITBs(ITB(ia,k,t(k)|-Op(k1,k2),Wreg(r),NoSpec),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstOp(instIa)
     where r1 :=() reg1ofOp(instIa) where r2 :=() reg2ofOp(instIa)
     where r :=() nameofOp(instIa) where k :=() lengthof(itbs)+1
     where k1 :=() searchforLastTag(r1,rf,itbs)
     where k2 :=() searchforLastTag(r2,rf,itbs) end
[PsJzIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,Proc(pia,rf,
 insEndITBs(ITB(ia,k,Jz(k0,k1),NoWreg,Spec(pia)),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstJz(instIa)
     where r1 :=() reg1ofJz(instIa)
     where r2 :=() reg2ofJz(instIa)
     where k :=() lengthof(itbs)+1
     where k0 :=() searchforLastTag(r1,rf,itbs)
     where k1:=()searchforLastTag(r2,rf,itbs)
     where pia:=()getbtb(ia,btb) end
[PsLoadIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,Proc(ia+1,rf,
 insEndITBs(ITB(ia,k,t(k)|-Load(k1),Wreg(r),NoSpec),itbs),btb,prog))
     where instIa :=() selectinst(prog,ia) if isinstLoad(instIa)
     where r:=()nameofLoad(instIa)
     where r0:=()reg1ofLoad(instIa)
     where k:=()lengthof(itbs)+1
     where k1:=() searchforLastTag(r0,rf,itbs) end
[PsStoreIssue]Sys(m,Proc(ia,rf,itbs,btb,prog)) => Sys(m,Proc(ia+1,rf,
 insEndITBs(ITB(ia,k,Store(k0,k1),NoWreg,NoSpec),itbs),btb,prog))
     where instIa:=()selectinst(prog,ia) if isinstStore(instIa)
     where r0 :=() nameofStoreR1(instIa)
     where r1:=()nameofStoreR2(instIa)
     where k:=()lengthof(itbs)+1
     where k0:=()searchforLastTag(r0,rf,itbs)
     where k1:=()searchforLastTag(r1,rf,itbs) end
```

Table 3
Instruction Issue Rules

[PsJumpCorrectSpec]
```
Sys(m,Proc(ia,rf,ITB(ia1,k,Jz(0,nia),wf,Spec(pia)).itbs,btb,prog)) =>
Sys(m,Proc(ia,rf,itbs,btb,prog)) if pia==nia end
```
[PsJumpWrongSpec]
```
Sys(m,Proc(ia,rf,ITB(ia1,k,Jz(0,nia),wf,Spec(pia)).itbs,btb,prog)) =>
Sys(m,Proc(nia,rf,nilitb,btb1,prog))
    if pia!=nia where btb1:=()changebtb(ia1,nia,btb) end
```
[PsNoJumpCorrectSpec]
```
Sys(m,Proc(ia,rf,ITB(ia1,k,Jz(v,nia),wf,Spec(pia)).itbs,btb,prog)) =>
Sys(m,Proc(ia,rf,itbs,btb,prog)) if v != 0 and pia == ia1+1 end
```
[PsNoJumpWrongSpec]
```
Sys(m,Proc(ia,rf,ITB(ia1,k,Jz(v,nia),wf,Spec(pia)).itbs,btb,prog)) =>
Sys(m,Proc(ia1+1,rf,nilitb,btb1,prog))
    if v != 0 and pia != ia1+1
    where btb1 :=() changebtb(ia1,ia1+1,btb) end
```

Table 4
Branch Completion Rules

[PsLoad]
```
Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-Load(v),wf,sf).itbs,btb,prog))
=> Sys(m, Proc(ia,rf,ITB(ia1,k,t(k)|-v0,wf,sf).itbs,btb,prog))
    where v0 :=() valueofMem(v,m) end
```
[PsStore]
```
Sys(m,Proc(ia,rf,ITB(ia1,k,Store(a,v),wf,sf).itbs,btb,prog))
=> Sys(insertMEM(m,a,v), Proc(ia,rf,itbs,btb,prog)) end
```

Table 5
Memory Access Rules

It is easy to show that the speculative processor simulates the basic one. One basic processor term can be "upgraded" to one of the speculative processor simply by adding an empty ITB and an arbitrary BTB to the processor.

Definition 1 (ITBL) *The Instruction Template Buffer Lift of a basic processor term is defined by*

$$ITBL(\texttt{Sys(m,Proc(ia,rf,prog))}) \equiv \texttt{Sys(m,Proc(ia,rf,nilitb,btb,prog))}$$

where `btb` *is an arbitrary BTB and* `nilitb` *and empty ITB.*

Theorem 1 *Let s and t be system terms of the basic processor. If $s \to^* t$ in the basic processor, then $ITBL(s) \to^* ITBL(t)$ in the speculative processor.*

Proof. Sequences of rules of the speculative processor can simulate each basic processor rule. For example, the `Op` rule in the basic processor can be simulated by consecutively applying the `PsOpIssue`, `PsOp` and `PsValueCommit` rules in the speculative processor; the `Load` rule in the basic processor can be simulated by consecutively applying the `PsLoadIssue`,`PsValueForward`, `PsLoad` and `PsValueCommit` rules in the speculative processor; etc. □

Now we need to define a projection function from the speculative processor to the basic processor. This is not simple because of the partially executed instructions. The approach in [12] is based on flushing instructions in the ITB. The key observation is that during some time of execution over an speculative processor, if no instruction is issued then the ITB will soon become empty. Only instruction issue rules can further expand the ITB. So, we can define another rewriting system which uses the same grammar as the speculative processor and include all its rules except the instruction issue ones.

Definition 2 *The rewriting system R_{ITBF} over terms of the speculative processor is given by the set of rewriting rules* {`PsOp`, `PsValueForward`, `PsValueCommit`, `PsJumpCorrectSpec`, `PsJumpWrongSpec`, `PsNoJumpCorrectSpec`, `PsNoJumpWrongSpec`, `PsLoad`, `PsStore`}.

One can prove that the rewriting system R_{ITBF} is strongly terminating and confluent and that its normal forms have always empty ITBs.

Definition 3 (ITBF) *Let* `Sys(m,Proc(ia,rf,nilitb,btb,prog))` *be the R_{ITBF} normal form of a given term of the speculative processor s. The* instruction template buffer flush *of s, denoted by ITBF(s), is the result of deleting from this R_{ITBF} normal form its empty ITB and its BTB:* `Sys(m,Proc(ia,rf,prog))`.

Theorem 2 *Let s and t be system terms of the speculative processor. If $s \to^* t$, then ITBF(s) $\to^*$ ITBF(t) in the basic processor.*

Speculative processor issue rules	basic processor rules
`PsLoadcIssue`	`Loadc`
`PsLoadpcIssue`	`Loadpc`
`PsOpIssue`	`Op`
`PsJzIssue`	`Jz`
`PsLoadIssue`	`Load`
`PsStoreIssue`	`Store`

Table 6
Correspondence between speculative issue rules and basic processor rules

Proof. The proof is by induction on the number of rewrite steps n on the derivation $s \to^n t$. For $n = 0$ this is obvious. For the inductive step, assume $s \to t$ by applying the rule α. If $\alpha \in R_{ITBF}$, then ITBF(s) and ITBF(t) coincide. If $\alpha \notin R_{ITBF}$, that is α is an instruction issue rule, then we will prove that either ITBF(s) and ITBF(t) coincide or ITBF(s) can be rewritten into ITBF(t) by applying an appropriate basic processor rule.

Suppose $s \to s_1$ by applying a rule $\beta \in R_{ITBF}$. We have two cases:
Case 1. β is a mis-prediction-recover rule: `PsJumpWrongSpec` or `PsNoJumpWrongSpec`. By applying β to t we have $t \to s_1$, since the instruction issuing will be canceled by the mis-prediction-recover rule.
Case 2. β is not a mis-prediction-recover rule. In this case we can notice that α can also be applied to s_1. Suppose that $s_1 \to t_1$ by applying α. If α is `PsValueCommit` and the register to which the value is committed is

referenced as an operand register in the instruction issued by α, then $t \to^* t_1$ by first applying once or twice `PsValueForward`, and then applying the rule β. Otherwise $t \to t_1$ by applying β.

Let s_n be the R_{ITBF} normal form of s. We have two cases to consider:
Case 1. If this normalization from s into s_n invokes one mis-prediction-recover rule, then there are terms s_i, t_i and s_{i+1} such that $s_i \to t_i$, by applying α, $s_i \to s_{i+1}$ and $t_i \to s_{i+1}$ by applying the mis-prediction-recover rule. This implies that s and t have identical R_{ITBF} normal forms.
Case 2. In the other case, by induction, we have that α can be applied to s_n to yield t_n such that $t \to^* t_n$ by applying just R_{ITBF} rules.

Let t_{n+1} be the R_{ITBF} normal form of t_n. Since s_n and t_{n+1} both have an empty ITB, we can easily show that $ITBF(s_n) \to ITBF(t_{n+1})$ by applying the corresponding basic processor rule according to the Table 6. □

3 Benefits of the separation between logic and rewriting in simulating processors

The natural separation in ELAN between rewriting and logic enables the controlled application of rules (i.e., processor instructions) and the adequate simulation of many interesting elements of hardware. For instance, the size of ROBs is one of the basic hardware ingredients of the speculative processor that is controlled by ELAN strategies. In fact, ROBs are controlled by specifying strategies which restrict the number of applications of issue rules. Suppose you want to simulate a ROB of size n, that should completely be filled and emptied alternately. Then the following simple ELAN strategy is used:

$$
\mathit{repeat}* \left(\begin{array}{l} \mathit{first\ one}(\mathit{issue\_rules}); \\ \left. \begin{array}{l} \ \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \\ \qquad\qquad \vdots \\ \ \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \end{array} \right\} n\text{-}1 \\ \mathit{normalise}(\mathit{first\ one}(\mathit{non\_issue\_rules})) \end{array} \right)
$$

Other strategies for handling the ROBs can similarly be specified. For example, for maintaining a ROB of size n filled during the whole execution, one can start as before, but in the subsequent normalization with all non issue rules (R_{ITBF} normalization) these rules should be treated individually. This treatment depends on whether the given non issue rule maintains or decreases the number of instruction templates in the ROB. For instance, since after a wrong branch speculation (rule `PsJumpWrongSpec`) the ROB is emptied the strategy should immediately fill the ROB by applying n issue rules. Below we sketch this strategy showing the case of the rule `PsJumpWrongSpec`.

$$
\mathit{repeat}* \left(\begin{array}{l}
\left. \begin{array}{l}
\mathit{first\ one}(\mathit{issue\_rules}); \\
\left. \begin{array}{l}
\mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \\
\vdots \\
\mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id});
\end{array} \right\} n\text{-}1
\end{array} \right\} \text{Initialization filling the ROB} \\
\mathit{normalise} \left(\mathit{first\ one} \left(\begin{array}{r}
\vdots \left.\right\} \begin{array}{l} \text{treatment of all} \\ \text{other non issue rules} \end{array} \\
\left[\begin{array}{l}
\mathtt{PsJumpWrongSpec}; \\
\mathit{first\ one}(\mathit{issue\_rules}); \\
\left. \begin{array}{l}
\mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \\
\vdots \\
\mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id});
\end{array} \right\} n\text{-}1
\end{array} \right] \\
\vdots \left.\right\} \begin{array}{l} \text{treatment of all} \\ \text{other non issue rules} \end{array}
\end{array} \right) \right)
\end{array} \right)
$$

In contrast to the control of ROBs other interesting aspects of processors as the method of branching prediction are directly controlled by the rewriting rules. The advantages of having ROBs is that instruction templates may be charged and these templates partially executed by the pipeline control. When at a point of the computation, determined by the program counter `ia`, a branch instruction template `Jz(r1,r2)` is charged into the ROB, it is undecided which is the following instruction template to be charged into the ROB, since at this point of the computation the values of the tags associated with the registers `r1` and `r2` are not necessarily resolved. Thus in speculative processors one has to decide which instruction template is the next to be charged according to the contents for the `ia` in the BTB. Well-known dynamic branch prediction schemes are specified by simple rewriting rules. We mention here the *1-bit* and *2-bit dynamic prediction* methods [15]. Initially, any prediction is given in the BTB. For instance, one can give pairs $(1, 2), ..., (j, j + 1), ..., (n, n + 1))$ meaning that after execution of the j^{th} instruction the prediction is to jump to the next instruction $(j + 1^{th})$ of the program. These predictions (i.e., pairs) are only necessary for the addresses of branch instructions in the program. Subsequently, the predictions are modified according to the execution history.

In 1-bit dynamic prediction, the prediction for the n^{th} instruction is actualized according to the next instruction to be executed. Once a prediction fails the corresponding value in the BTB is changed to the correct address of the instruction to be executed.

In 2-bit dynamic prediction, there are four different states of the prediction: *strongly taken, weakly taken, weakly not taken, strongly not taken.* If the state is either *strongly (not) taken* or *weakly (not) taken* and the prediction is correct: "jump" ("next instruction"), then the state is changed to *strongly (not) taken.* If the state is *strongly (not) taken* and the prediction fails: "next instruction" ("jump"), then the state is changed to *weakly (not) taken.* If the state is *weakly (not) taken* and the prediction fails: "next instruction" ("jump"), then the BTB is modified according to the correct address given by the contents of the second register of the branch instruction and the state is changed to *weakly not taken* (*weakly taken*).

	Size	10 ran	10 ord	20 ran	20 ord	30 ran	30 ord	40 ran	40 ord	50 ran	50 ord
1-bit	correct	30	60	109	225	185	490	278	855	401	1320
	wrong	34	34	72	74	114	114	159	154	196	194
2-bit	correct	28	73	120	258	194	543	286	928	407	1413
	wrong	36	21	61	41	105	61	151	81	200	101

Table 7
Elan statistics for quick-sort executed with 1-bit and 2-bit dynamic predictions

Both prediction strategies are specified and simulated by purely rewriting. This is implemented by simple boolean conditions over the branch completion rules: comparisons between `pia` (predicted instruction address), `nia` (next correct instruction address) and `ia+1` (next instruction address in the program) for the four branch completion rules in the Table 4. Once a prediction fails, the BTB is modified by the function `changebtb`, that is specified by purely rewriting and adapted for the two prediction methods.

Furthermore, the performance of different ways to implement proposed processors can be determined by analyzing the ELAN statistics. For instance, one can estimate whether 1-bit performs better than 2-bit prediction for the execution of an assembly description of quick-sort over the speculative processor implemented with the strategy of alternatively filling and emptying the ROB. The total number of wrong and correct predictions (i.e., number of applications of branch completion rules in the Table 4) for ordered (the worst-case of quick-sort) and (the average for) random lists are given in the Table 7. The observation of the differences between the number of wrong predictions for both methods gives an important insight about the advantages of 2-bit over 1-bit prediction, since in the worst-case a wrong prediction flushes the ROB which has been filled with instruction templates over which previous operations have been executed. One can check on the table that the differences between the number of wrong predictions for the two methods is much more significant for ordered lists than for random lists. Consequently, the physical hardware implementation of a processor dedicated to this kind of sorting for random inputs can be performed with the simplest (and cheaper) 1-bit method.

Important aspects like out-of-order execution are not easy to implement in practical purely rewrite based programming environments. In fact, out-of-order execution of instruction templates over a ROB can only be simulated by allowing a truly non-deterministic application of the rewriting rules (i.e., processor instructions) over the ROB during any time of the computation. For allowing out-of-order execution, instead of the usual CONS operator "." of instruction templates and ITBs (which appears as `inst_temp.itbs` in our implementation) a new operator "`#`" is defined for concatenating ITBs and/or instruction templates. Thus ITBs are represented as `itbs1#inst_temp#itbs2` being `itbs1` and `itbs2` lists of instruction templates and `inst_temp` a sole instruction template. The rewriting rules are modified by replacing all their

ITBs with this new representation as we illustrate for the `[PsOp]` rule below:

```
Sys(m,Proc(ia,rf,itbs1#ITB(ia1,k,t(k)|-Op(v,v1),wf,sf)#itbs2,btb,prog)) =>
Sys(m,Proc(ia,rf,itbs1#ITB(ia1,k,t(k)|-execOponval(v,v1),wf,sf)#itbs2,btb,
                                                                  prog)) end
```

By matching the instruction template, `ITB(ia1,k,t(k)|-Op(v,v1),wf,sf)`, the new `[PsOp]` rule can be applied not only at the first but at any position of the current ITB: `itbs1#ITB(ia1,k,t(k)|-Op(v,v1),wf,sf)#itbs2`. In the theory, the rewriting system obtained by modifying all the rules as suggested above enables out-of-order execution, since rewriting rules are applied non-deterministically. But in the practice, in purely rewrite based programming environments, this solution does not work since the application of a rule is decided by searching for either left-most or right-most (inner-most) redices over the ITBs (according to the way the constructor "`#`" is defined) [9].

For rewriting based implementations of a real out-of-order execution mechanism, the availability of true non-deterministic strategies is necessary. With some additional effort, in a rewriting-logic based system as ELAN strategy constructors like *don't know choose* (that gives all possible reducts) can be adapted for simulating the needed non-determinism over the ROBs [17,10,11].

4 Conclusions and future work

We have shown how processors may be specified and their execution simulated over rewriting(-logic) systems. Unlike Arvind's group, who proposes the simulation of the execution of these specifications over standard hardware description languages, we address the simulation of the execution of processors directly over the rewriting specification avoiding the cost of program translation. Furthermore, we have illustrated why the rewriting part as well as the logical part of ELAN are adequate for the simulation of simple hardware components like the method of prediction in speculative processors (done in our case by pure rewriting) and control of the size of ROBs (done in our case by logic strategies). After having specified the rewriting rules for the instruction set of a processor, the intrinsic separation between logic and rewriting in ELAN results in enough versatility for dealing with different conceptions of manipulation of ROBs without additional effort in these rewrite specifications. Additionally, we illustrate how statistics of the application of rewriting rules may be used for estimating and comparing the performance of different processors. Although not done in our implementation, non-deterministic strategies implemented in ELAN also may be shown to be adequate for simulating essential hardware conceptions of these processors like the *out-of-order* execution of instruction templates in ROBs.

Through rewriting-logic one can describe an architecture as precisely as one wants. For example, rules of the speculative processor may be atomized in order to reflect the behavior of lower-level hardware components such as pipelines and functional units of processors like fetch, decode and execution units. Also, the (higher-order) rewriting-logic based simulation of *reconfig-*

urable processors [6], which are non-standard models of computing where two layers of instructions are needed (the one for the instruction set and the other for the processor reconfiguration) is of great interest, since no simulation is possible over standard hardware description languages such as Verilog and VHDL. One of our current goals is to analyze the possibilities of using rewriting for synthesizing (logic components for building) logical circuits for arithmetic operators at their layout level [2]. One of the interesting aspects that emerges from this study is the necessity of new hardware oriented notions of normal forms, since the more adequate algebraic expressions to be transformed into circuits are the ones with *more regularities*. These are consequently the ones that can be implemented with the smallest number of different classes of atomic hardware components, and are not the simplest ones from the algebraic point of view, which is the norm in rewriting.

References

[1] Arvind and X. Shen. Using Term Rewriting Systems to Design and Verify Processors. Technical Report 419, Laboratory for Computer Science - MIT, 1999. Also in IEEE Micro Special Issue on "Modeling and Validation of Microprocessors", 1999.

[2] M. Ayala-Rincón, R. W. Hartenstein, R.P. Jacobi, and C. Llanos. Designing Arithmetic Digital Circuits via Rewriting-Logic. Available at `www.mat.unb.br/~ayala/publications.html`, 2002.

[3] F. Baader and T. Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Electronic Notes in Theoretical Computer Science*, volume 15. Elsevier Science Publishers, 1998.

[5] H. Cirstea and C. Kirchner. Combining Higher-Order and First-Order Computation Using ρ-Calculus: Towards a Semantics of ELAN. In *Frontiers of Combining Systems 2*, Studies on Logic and Computation, 7, chapter 6, pages 95–121. Research Studies Press/Wiley, 1999.

[6] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the DATE 2001 on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.

[7] J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, second edition, 2002.

[8] J. C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. Technical Report 421 A, Laboratory for Computer Science - MIT, 1999. Also in Proc. of the Tenth IFIP International Conference on VLSI - VLSI 1999.

[9] H. Hussmann. *Nondeterminism in Algebraic Specifications and Algebraic Programs.* Birkhäuser, 1993.

[10] H. Kirchner and P.-E. Moreau. Non-deterministic computations in ELAN. In J.L. Fiadeiro, editor, *Recent Developments in Algebraic Specification Techniques, Proc. 13th WADT'98, Selected Papers*, volume 1589 of *LNCS*, pages 168–182. Springer, 1998.

[11] H. Kirchner and P.-E. Moreau. Promoting Rewriting to a Programming Language: A Compiler for Non-Deterministic Rewrite Programs in Associative-Commutative Theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[12] X. Shen and Arvind. Design and Verification of Speculative Processors. Technical Report 400 A, Laboratory for Computer Science - MIT, 1998.

[13] X. Shen and Arvind. Modeling and Verification of ISA Implementations. Technical Report 400 B, Laboratory for Computer Science - MIT, 1998.

[14] X. Shen, Arvind, and L. Rudolph. CACHET: an adaptive cache coherence protocol for distributed shared-memory systems. In *International Conference on Supercomputing*, pages 135–144. ACM, 1999.

[15] D. Sima, T. Fountain, and P. Kacsuck. *Advanced Computer Architectures: a Design Space Approach.* Addison-Wesley, 1997.

[16] J. Stoy, X. Shen, and Arvind. Proofs of Correctness of Cache-Coherence Protocols. In *FME 2001: Formal Methods for Increasing Software Productivity, Int. Symposium of Formal Methods Europe*, volume 2021 of *LNCS*, pages 43–71. Springer, 2001.

[17] M. Vittek. A Compiler for Nondeterministic Term Rewriting Systems. In H. Ganzinger, editor, *Proc. Seventh Int. Conf. on Rewriting Techniques and Applications RTA-96*, volume 1103 of *LNCS*, pages 154–168. Springer, July 1996.