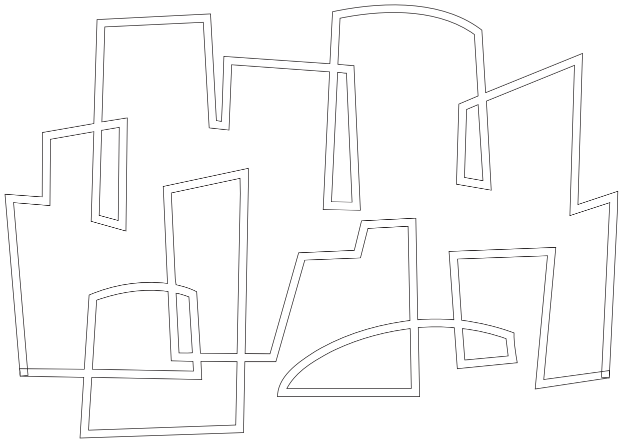
Design and Verification of Speculative Processors

Xiaowei Shen, Arvind

In Proceedings of the Workshop on
Formal Techniques for
Hardware and Hardware-like
Systems Architecture, 1998, June

Computation Structures Group
Memo 400B



Design and Verification of Speculative Processors

Computation Structures Group Memo 400 (B)
June 20, 1997

Xiaowei Shen and Arvind
xwshen, arvind@lcs.mit.edu

To appear in Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden, June 1998.

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft Huachuca contract DABT63-95-C-0150.

Design and Verification of Speculative Processors

Xiaowei Shen and Arvind
xwshen, arvind@lcs.mit.edu

June 20, 1997

Abstract

We define $\mathcal{AX}$, a simple RISC instruction set, by specifying its operational semantics using term rewriting systems (TRS). We then give another TRS that models an $\mathcal{AX}$ implementation which permits out-of-order and speculative instruction execution. The correctness of the speculative implementation is proved by showing that the two TRS's can simulate each other with regards to some observation function. Our method facilitates understanding of important micro-architectural differences without delving into low-level implementation details. For example, we will show that an ISA implementation that is correct for uniprocessor systems is not necessarily so for multiprocessor systems.

1 Introduction

We have introduced a novel approach based on Term Rewriting Systems (TRS) to describe ISA implementations in [9]. A computer system and its components are described as terms generated by a context-free grammar, and the operational behavior of the ISA is specified as a set of rules for rewriting the terms that represent the system or its components. Term rewriting systems [4] are convenient for describing parallel systems, and can be used to prove the correctness of an implementation with respect to a specification.

This paper is an extension of the work presented in [9] where we described and proved the correctness of a processor with register renaming and out-of-order instruction execution capabilities. Here we deal with an implementation with speculative instruction execution capability and employ a slightly different proof technique. The proof is useful for showing the correctness of the processor implementation even in the multiprocessor setting.

Formal verification of microprocessors has gained considerable attention in recent years. For example, Burch and Dill [1] described a technique which automatically compares a pipelined implementation to an architectural specification and produces debugging information for incorrect processor design. Levitt and Olukotun [5] proposed a methodology that iteratively de-constructs a pipeline by merging adjacent pipeline stages thereby allowing verifications to be done in a number of easier steps. Windley [10] presented a case study which uses abstract theories to hierarchically verify microprocessor implementations formalized in HOL.

Windley's methodology is similar to ours, in the sense that the correctness theorem states the implementation implies the behavior specification. The most critical step in the proof is the definition of the abstract mapping function to map the states of one system into the states of the other system. Our proofs and mapping functions are simple and intuitive, perhaps because of the use of TRS's.

Though the reader is the ultimate judge, we believe that our descriptions of micro-architectures are more precise than what one may find in a modern textbook [3]. It is the clarity of these descriptions that lets us study the impact of features such as write buffers on multiprocessors. In fact part of the motivation for this work came from one of the author's experience in teaching computer architecture.

We give a brief introduction to TRS in Section 2. In Sections 3 and 4 we define the $\mathcal{AX}$ instruction set and specify its operational semantics using a simple in-order execution processor ($\mathcal{P}_{\mathcal{B}}$). These sections have been lifted verbatim from [9] and are included here to make the paper self-contained. In Section 5 we present an implementation of $\mathcal{AX}$ that allows out-of-order and speculative execution ($\mathcal{P}_{\mathcal{S}}$). Then in Section 6 we formally prove the correctness of $\mathcal{P}_{\mathcal{S}}$ by showing that $\mathcal{P}_{\mathcal{B}}$ and $\mathcal{P}_{\mathcal{S}}$ can simulate each other. In Section 7 we demonstrate that $\mathcal{P}_{\mathcal{B}}$ and $\mathcal{P}_{\mathcal{S}}$ have the same observable behavior in multiprocessor systems. In Sections 8 and 9 we discuss aggressive implementations of memory operations and their impact on the behavior of concurrent programs in multiprocessor systems. Finally we briefly discuss some related work-in-progress.

2 Term Rewriting Systems

A term rewriting system is defined as a tuple (S, R, S_0), where S is a set of terms, R is a set of rewriting rules, and S_0 is the set of initial terms ($\mathrm{S}_0 \subseteq \mathrm{S}$). In the architectural context, the terms and rules of a TRS represent states and state transitions, respectively. The general structure of rewriting rules is as follows:

$$\begin{array}{ll} & s_1 \quad \textit{if} \;\; p\,(s_1) \\ \longrightarrow & s_2 \end{array}$$

where s_1 and s_2 are terms, and p is a predicate.

A rule can be used to rewrite a term if its left-hand-side pattern matches the term or one of its subterms, and the corresponding predicate is true. If several rules are applicable, then any one of them can be applied. If no rule is applicable, then the term cannot be rewritten any further and is said to be in *normal form*.

We use $\mathrm{C}[\![\;]\!]$ to represent a *context*, which is a term with a "hole" that can be filled by a term. $\mathrm{C}[\![s]\!]$ refers to the term in which the hole is filled by term s.

We say term s_1 can be rewritten to term s_2 in one rewriting step ($s_1 \longrightarrow s_2$), if there exist a context $\mathrm{C}[\![\;]\!]$ and terms s_1' and s_2' such that $s_1 = \mathrm{C}[\![s_1']\!]$ and $s_2 = \mathrm{C}[\![s_2']\!]$, and s_1' can be rewritten to s_2' according to some rewriting rule.

We say term s_1 can be rewritten to term s_2 in zero or more rewriting steps ($s_1 \twoheadrightarrow s_2$), if either $s_1 = s_2$, or there exists a term s' such that $s_1 \longrightarrow s'$ and $s' \twoheadrightarrow s_2$.

A term s is *legal* if there exists $s_0 \in \mathrm{S}_0$ such that $s_0 \twoheadrightarrow s$. Since we are only interested in legal terms, we will drop the qualifier "legal" in our discussion.

INST	$\equiv$	$\mathsf{r} := \mathsf{Loadc}(\mathsf{v})$	*Load-constant Instruction*
	$[\!]$	$\mathsf{r} := \mathsf{Loadpc}$	*Load-program-counter Instruction*
	$[\!]$	$\mathsf{r} := \mathsf{Op}(\mathsf{r}_1, \mathsf{r}_2)$	*Arithmetic-operation Instruction*
	$[\!]$	$\mathsf{Jz}(\mathsf{r}_1, \mathsf{r}_2)$	*Branch Instruction*
	$[\!]$	$\mathsf{r} := \mathsf{Load}(\mathsf{r}_1)$	*Load Instruction*
	$[\!]$	$\mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2)$	*Store Instruction*

Figure 1: $\mathcal{AX}$ Instruction Set

A TRS is *confluent* if, for any term s_1, if $s_1 \twoheadrightarrow s_2$ and $s_1 \twoheadrightarrow s_3$, then there exists a term s_4 such that $s_2 \twoheadrightarrow s_4$ and $s_3 \twoheadrightarrow s_4$.

A TRS is *strongly terminating* if, for any term, it can always be rewritten to a normal form using any rewriting strategy.

3 $\mathcal{AX}$ Instruction Set Architecture

$\mathcal{AX}$ is a minimalist RISC instruction set (see Figure 1), in which all arithmetic operations are performed on registers and only the Load and Store instructions can access memory. Semantically instructions are executed strictly according to the program order: the program counter is incremented by one each time an instruction is executed except for the Jz instruction, where the program counter is set appropriately according to the branch condition. The informal meaning of the instructions is as follows:

The load-constant instruction $\mathsf{r} := \mathsf{Loadc}(\mathsf{v})$ puts constant v into register r. The load-program-counter instruction $\mathsf{r} := \mathsf{Loadpc}$ puts the content of the program counter into register r. The arithmetic-operation instruction $\mathsf{r} := \mathsf{Op}(\mathsf{r}_1, \mathsf{r}_2)$ performs an arithmetic operation on operands specified by registers r_1 and r_2, and puts the result into register r. The branch instruction $\mathsf{Jz}(\mathsf{r}_1, \mathsf{r}_2)$ sets the program counter to the target instruction address specified by register r_2 if register r_1 contains value zero (otherwise the program counter is simply increased by one). The load instruction $\mathsf{r} := \mathsf{Load}(\mathsf{r}_1)$ reads the memory cell specified by register r_1, and puts the data into register r. The store instruction $\mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2)$ writes the content of register r_2 into the memory cell specified by register r_1.

Throughout the paper, we use '$[\!]$' as meta notation in grammars to separate disjuncts. We use 'a' and 'ia' to represent a data address and an instruction address, respectively. We use 'r' as a register name, 't' as a register renaming tag, 'u' and 'v' as values, and 'tv' as either a register renaming tag or a value. Subscripts will be used to distinguish domain elements whenever necessary. To avoid unnecessary complications, we assume that the instruction address space is disjoint from the data address space, so that self-modifying code is forbidden.

SYS	$\equiv$	$\mathsf{Sys}(\mathrm{MEM},\ \mathrm{PROC})$	*System*
MEM	$\equiv$	$\epsilon \ \ [\!] \ \ \mathsf{Cell(a,v)} \mid \mathrm{MEM}$	*Memory*
PROC	$\equiv$	$\mathsf{Proc}(\mathrm{PC},\ \mathrm{RF},\ \mathrm{PROG})$	*Processor*
PC	$\equiv$	ia	*Program Counter*
RF	$\equiv$	$\epsilon \ \ [\!] \ \ \mathsf{Reg(r,v)} \mid \mathrm{RF}$	*Register File*
PROG	$\equiv$	$\epsilon \ \ [\!] \ \ \mathsf{Inst(ia,}\ \mathrm{INST}) \mid \mathrm{PROG}$	*Program*

Figure 2: $\mathcal{P}_\mathcal{B}$ Model

4 $\mathcal{P}_\mathcal{B}$ Model: Operational Semantics of $\mathcal{AX}$

In this section, we define the operational semantics of the $\mathcal{AX}$ instruction set with respect to $\mathcal{P}_\mathcal{B}$ (base processor), a single-cycle, non-pipelined, in-order execution model. The grammar of $\mathcal{P}_\mathcal{B}$ is given in Figure 2. The system has two components, a memory and a processor. The memory consists of a set of memory cells, where each memory cell has an address and a value. The processor consists of a program counter, a register file, and a program. The program counter holds the address of the instruction to be executed. The register file is a set of registers, where each register has a register name and a value. The program is a set of instructions, in which each instruction is associated with an instruction address.

In our notation, '|' is a constructor that is commutative and associative. We use 'ϵ' to represent the empty term, and '-' to represent the wild-card term that can match any term. We assume that instructions in a program have distinct instruction addresses, and use notation $\mathsf{prog[ia]}$ to refer to the instruction with instruction address ia in the program prog. We assume that addresses in the memory are pairwise distinct, and so are register names in the register file. Notation $\mathsf{m[a]}$ refers to the content of memory cell a, and notation $\mathsf{m[a := v]}$ represents memory m with memory cell a updated with value v. Similarly, notation $\mathsf{rf[r]}$ refers to the content of register r, and notation $\mathsf{rf[r := v]}$ represents the register file that differs from rf only in the content of register r.

In the initial system term, the program counter is the address of the first instruction to be executed, and all registers and memory cells have the undefined value '$\perp$'. The following rewriting rules specify the operational semantics of the $\mathcal{AX}$ instruction set:

Loadc Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \ \ \mathsf{prog[ia]} = \mathsf{r := Loadc(v)} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf[r := v],\ prog)} \end{array}$$

Loadpc Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \ \ \mathsf{prog[ia]} = \mathsf{r := Loadpc} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf[r := ia],\ prog)} \end{array}$$

Op Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \ \ \mathsf{prog[ia]} = \mathsf{r := Op(r_1, r_2)} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf[r := v],\ prog)} \quad \textit{where} \ \ \mathsf{v} = \underline{\mathsf{Op}}\mathsf{(rf[r_1], rf[r_2])} \end{array}$$

Jz-Jump Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{Jz(r_1,r_2)} \quad \textit{and} \quad \mathsf{rf[r_1]} = 0 \\ \longrightarrow & \mathsf{Proc(rf[r_2],\ rf,\ prog)} \end{array}$$

Jz-NoJump Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{Jz(r_1,r_2)} \quad \textit{and} \quad \mathsf{rf[r_1]} \neq 0 \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf,\ prog)} \end{array}$$

Load Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ \ Proc(ia,\ rf,\ prog))} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{r := Load(r_1)} \\ \longrightarrow & \mathsf{Sys(m,\ \ Proc(ia{+}1,\ rf[r := m[a]],\ prog))} \quad \textit{where} \quad \mathsf{a} = \mathsf{rf[r_1]} \end{array}$$

Store Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ \ Proc(ia,\ rf,\ prog))} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{Store(r_1,r_2)} \\ \longrightarrow & \mathsf{Sys(m[a := rf[r_2]],\ \ Proc(ia{+}1,\ rf,\ prog))} \quad \textit{where} \quad \mathsf{a} = \mathsf{rf[r_1]} \end{array}$$

Notice the memory access rules involve both the processor and the memory (i.e. the system), while other rules only deal with the processor. Notation $\underline{\mathsf{Op}}(\mathsf{v_1}, \mathsf{v_2})$ represents the result of operation Op with operands $\mathsf{v_1}$ and $\mathsf{v_2}$.

5 $\mathcal{P}_S$ Model: An Implementation with Speculative Execution

Micro-architectures that do Tomasulo style register renaming maintain a register renaming table and a set of instruction template buffers (ITBs) to hold instructions that have been issued and assigned register renaming tags but have not yet completed execution. An instruction template buffer usually holds an instruction in a form where all register names have been appropriately replaced by either renaming tags or values. An arithmetic operation in the ITBs can be executed if all its operands are available. A natural consequence of register renaming is that instructions can be executed in a different order from the program order.

In addition to the out-of-order execution, most contemporary microprocessors also permit speculative execution of instructions. The speculation mechanism is restricted to speculate only the address of the next instruction to be issued. (Several researchers have suggested mechanisms to speculate on memory values as well but none of these have been implemented so far; we do not consider such mechanisms in this paper). 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 have to be abandoned and their effect on the processor state must be nullified. The BTB is updated according to some prediction scheme after each branch resolution.

We assume that the BTB produces the correct next instruction address for all non-branch instructions. The correctness of the speculative processor is not contingent upon how the BTB is maintained. However, different prediction schemes can give rise to very different misprediction rates and thus have profound influence on the performance. We will not discuss the BTB any further because the branch prediction strategy is completely orthogonal to the mechanisms for speculative execution.

SYS	$\equiv$	Sys(MEM, PROC)	*System*
MEM	$\equiv$	ϵ $[]$ Cell(a,v) | MEM	*Memory*
PROC	$\equiv$	Proc(PC, RF, ITBs, BTB, PROG)	*Processor*
PC	$\equiv$	ia	*Program Counter*
RF	$\equiv$	ϵ $[]$ Reg(r, v) | RF	*Register File*
ITBs	$\equiv$	ϵ $[]$ ITB(ia, IT, WF, SF) $\oplus$ ITBs	*Instruction Template Buffers*
IT	$\equiv$	$\mathsf{t} := \mathsf{tv}_1$	
	$[]$	$\mathsf{t} := \mathsf{Op}(\mathsf{tv}_1, \mathsf{tv}_2)$	
	$[]$	$\mathsf{Jz}(\mathsf{tv}_1, \mathsf{tv}_2)$	
	$[]$	$\mathsf{tv} := \mathsf{Load}(\mathsf{tv}_1)$	
	$[]$	$\mathsf{Store}(\mathsf{tv}_1, \mathsf{tv}_2)$	*Instruction Template*
WF	$\equiv$	Wreg(r) $[]$ NoWreg	*Write Flag*
SF	$\equiv$	Spec(ia) $[]$ NoSpec	*Speculation Flag*
tv	$\equiv$	t $[]$ v	*Tag or Value*
PROG	$\equiv$	ϵ $[]$ Inst(ia, INST) | PROG	*Program*

Figure 3: $\mathcal{P}_\mathcal{S}$ Model

Any processor that permits speculative execution has to make sure that either a speculative instruction does not modify the programmer visible state until it can be "committed", or save enough of the processor state when the speculation begins so that the correct state can be restored in case the speculation turns out to be wrong. Our implementation uses a mixture of these two ideas: speculative instructions cannot modify the register file or memory until it can be determined that the prediction is correct, but can update the program counter. Both the current and the speculated (next) instruction address are recorded in the instruction template buffer so that later the correctness of speculation can be determined and the correct program counter can be restored in case the branch prediction turns out to be wrong.

The grammar of $\mathcal{P}_\mathcal{S}$, an implementation of $\mathcal{AX}$ that allows out-of-order and speculative execution, is given in Figure 3. Two new components, ITBs and BTB, have been incorporated into the processor. The ITBs is maintained as an ordered queue using the constructor '$\oplus$', which is associative but not commutative. Initially the ITBs is empty. Each buffer in ITBs contains an instruction template, the associated instruction address, and some extra information needed for register writeback (w-flag) and speculation resolution (s-flag). The w-flag records the destination register to which the result needs to be committed when the instruction is completed, while the s-flag holds the speculated (next) instruction address which is used to determine the correctness of the prediction.

5.1 Instruction Issue Rules

Each time an instruction is issued, the program counter is set to the address of the next instruction to be issued. For non-branch instructions, the program counter is simply incremented by one. Speculative execution happens when a Jz instruction is issued: the program counter is then set to the instruction address obtained by consulting the BTB entry corresponding to the address of the Jz instruction.

When an instruction is issued, an instruction template for the issued instruction is created in the ITBs. If the instruction is to modify a register, an unused renaming tag is used to rename the destination register. The destination register is recorded in the w-flag so that the register can be updated when the instruction execution completes. For a Jz instruction, the s-flag holds the speculated (next) instruction address. This speculative address is later compared to the resolved branch target address to determine if the speculation was correct. The following table summarizes how the w-flag and s-flag are set at the instruction issue stage (some bits in the ITBs can be saved by merging the two flags in the implementation).

instruction type	flag setting	
	w-flag	s-flag
$\mathsf{r := Loadc(v)}$	Wreg(r)	NoSpec
$\mathsf{r := Loadpc}$	Wreg(r)	NoSpec
$\mathsf{r := Op(r_1, r_2)}$	Wreg(r)	NoSpec
$\mathsf{Jz(r_1, r_2)}$	NoWreg	Spec(pia)
$\mathsf{r := Load(r_1)}$	Wreg(r)	NoSpec
$\mathsf{Store(r_1, r_2)}$	NoWreg	NoSpec

At the time of instruction fetch, the tag or value of each operand register is found by searching the ITBs from the youngest buffer (rightmost) to the oldest buffer (leftmost) until an instruction template containing the referenced register is found. If no such buffer exists in the ITBs, then the most up-to-date value resides in the register file. The following lookup procedure captures this idea:

$$\textit{Def}\quad \mathsf{lookup(r,\ rf,\ itbs_1 \oplus ITB(\text{-},\ t := \text{-},\ Wreg(r),\ \text{-}) \oplus itbs_2)} \equiv \mathsf{t} \quad \textit{if}\ \mathsf{Wreg(r)} \notin \mathsf{itbs_2}$$

$$\textit{Def}\quad \mathsf{lookup(r,\ rf,\ itbs)} \equiv \mathsf{rf[r]} \quad \textit{if}\ \mathsf{Wreg(r)} \notin \mathsf{itbs}$$

In the following instruction issue rules, t represents an unused tag (i.e., $\mathsf{t} \notin \mathsf{itbs}$), and $\mathsf{tv_1}$ and $\mathsf{tv_2}$ represent the tag or value corresponding to the operand registers $\mathsf{r_1}$ and $\mathsf{r_2}$, respectively (i.e., $\mathsf{tv_1} = \mathsf{lookup(r_1,\ rf,\ itbs)}$, $\mathsf{tv_2} = \mathsf{lookup(r_2,\ rf,\ itbs)}$).

$\mathcal{P}_S$-Loadc-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ itbs,\ btb,\ prog)} \quad \textit{if}\ \mathsf{prog[ia] = r := Loadc(v)} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf,\ itbs \oplus ITB(ia,\ t := v,\ Wreg(r),\ NoSpec),\ btb,\ prog)} \end{array}$$

$\mathcal{P}_S$-Loadpc-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ itbs,\ btb,\ prog)} \quad \textit{if}\ \mathsf{prog[ia] = r := Loadpc} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf,\ itbs \oplus ITB(ia,\ t := ia,\ Wreg(r),\ NoSpec),\ btb,\ prog)} \end{array}$$

$\mathcal{P}_S$-Op-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ itbs,\ btb,\ prog)} \quad \textit{if}\ \mathsf{prog[ia] = r := Op(r_1, r_2)} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf,\ itbs \oplus ITB(ia,\ t := Op(tv_1, tv_2),\ Wreg(r),\ NoSpec),\ btb,\ prog)} \end{array}$$

$\mathcal{P}_S$-Jz-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ itbs,\ btb,\ prog)} \quad \textit{if}\ \mathsf{prog[ia] = Jz(r_1, r_2)} \\ \longrightarrow & \mathsf{Proc(pia,\ rf,\ itbs \oplus ITB(ia,\ Jz(tv_1, tv_2),\ NoWreg,\ Spec(pia)),\ btb,\ prog)} \\ & \qquad \textit{where}\ \ \mathsf{pia = btb[ia]} \end{array}$$

$\mathcal{P}_S$-Load-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ itbs,\ btb,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia] = r := Load(r_1)} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf,\ itbs \oplus ITB(ia,\ t := Load(tv_1),\ Wreg(r),\ NoSpec),\ btb,\ prog)} \end{array}$$

$\mathcal{P}_S$-Store-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ itbs,\ btb,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia] = Store(r_1, r_2)} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf,\ itbs \oplus ITB(ia,\ Store(tv_1, tv_2),\ NoWreg,\ NoSpec),\ btb,\ prog)} \end{array}$$

In any implementation, there are a finite number of instruction template buffers and renaming tags. Instruction issue has to be stalled if all instruction template buffers are occupied, or no unused renaming tag is available to rename the destination register. This availability checking can be easily modeled, and we leave it as a simple exercise for the interested reader.

5.2 Arithmetic Operation and Value Propagation Rules

The arithmetic operation rule states that an arithmetic operation in the ITBs can be performed if both operands are available. It assigns the result to the corresponding tag.

$\mathcal{P}_S$-Op Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ itbs_1 \oplus ITB(ia_1,\ t := Op(v_1, v_2),\ wf,\ sf) \oplus itbs_2,\ btb,\ prog)} \\ \longrightarrow & \mathsf{Proc(ia,\ rf,\ itbs_1 \oplus ITB(ia_1,\ t := v,\ wf,\ sf) \oplus itbs_2,\ btb,\ prog)} \\ & \qquad \textit{where} \quad \mathsf{v = \underline{Op}(v_1, v_2)} \end{array}$$

There are two value propagations rules, the forward rule and the commit rule. The forward rule sends the value of a tag to other instruction templates, while the commit rule writes the value to the destination register and retires the corresponding renaming tag. Notation $\mathsf{itbs_2[v/t]}$ means that one or more appearances of tag t in $\mathsf{itbs_2}$ are replaced by value v.

$\mathcal{P}_S$-Value-Forward Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ itbs_1 \oplus ITB(ia_1,\ t := v,\ wf,\ sf) \oplus itbs_2,\ btb,\ prog)} \quad \textit{if} \quad \mathsf{t \in itbs_2} \\ \longrightarrow & \mathsf{Proc(ia,\ rf,\ itbs_1 \oplus ITB(ia_1,\ t := v,\ wf,\ sf) \oplus itbs_2[v/t],\ btb,\ prog)} \end{array}$$

$\mathcal{P}_S$-Value-Commit Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ ITB(ia_1,\ t := v,\ Wreg(r),\ sf) \oplus itbs,\ btb,\ prog)} \quad \textit{if} \quad \mathsf{t \notin itbs} \\ \longrightarrow & \mathsf{Proc(ia,\ rf[r := v],\ itbs,\ btb,\ prog)} \end{array}$$

It is worth noting that the register file is modified by the oldest instruction template after it has forwarded the value to all the buffers in the ITBs that reference its tag. Restricting the register writeback to just the oldest instruction in the ITBs eliminates output (write-after-write) hazards, and protects the register file from being polluted by incorrect speculative instructions.

Also notice the implementation would be correct even without the commit rule. However, an unbounded number of instruction template buffers and renaming tags would then become necessary. An instruction template buffer cannot be freed until the tag in the template has been retired, and the tag for the destination register for Loadc, Loadpc, Op or Load instruction cannot be retired until the value of the tag has been comitted to the register file.

5.3 Branch Completion Rules

The branch completion rules determine if the branch prediction was correct by comparing the speculated instruction address and the resolved branch target instruction address. If the two do not match (indicating that the speculation was wrong), all instructions issued after the branch instruction are aborted, and the program counter is set to resume the program execution from the new branch target instruction. In the following branch completion rules, btb' represents the BTB which has been updated according to some prediction algorithm.

$\mathcal{P}_{\mathcal{S}}$-Jz-Jump-CorrectSpec Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1,\ \mathsf{Jz}(0,\mathsf{nia}),\ \mathsf{wf},\ \mathsf{Spec}(\mathsf{pia})) \oplus \mathsf{itbs}_2,\ \mathsf{btb},\ \mathsf{prog}) \\ & \qquad\qquad \textit{if} \ \ \mathsf{pia} = \mathsf{nia} \\ \longrightarrow & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{itbs}_2,\ \mathsf{btb}',\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{S}}$-Jz-Jump-WrongSpec Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1,\ \mathsf{Jz}(0,\mathsf{nia}),\ \mathsf{wf},\ \mathsf{Spec}(\mathsf{pia})) \oplus \mathsf{itbs}_2,\ \mathsf{btb},\ \mathsf{prog}) \\ & \qquad\qquad \textit{if} \ \ \mathsf{pia} \neq \mathsf{nia} \\ \longrightarrow & \mathsf{Proc}(\mathsf{nia},\ \mathsf{rf},\ \mathsf{itbs}_1,\ \mathsf{btb}',\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{S}}$-Jz-NoJump-CorrectSpec Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1,\ \mathsf{Jz}(\mathsf{v},\text{-}),\ \mathsf{wf},\ \mathsf{Spec}(\mathsf{pia})) \oplus \mathsf{itbs}_2,\ \mathsf{btb},\ \mathsf{prog}) \\ & \qquad\qquad \textit{if} \ \ \mathsf{v} \neq 0 \quad \textit{and} \ \ \mathsf{pia} = \mathsf{ia}_1{+}1 \\ \longrightarrow & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{itbs}_2,\ \mathsf{btb}',\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{S}}$-Jz-NoJump-WrongSpec Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1,\ \mathsf{Jz}(\mathsf{v},\text{-}),\ \mathsf{wf},\ \mathsf{Spec}(\mathsf{pia})) \oplus \mathsf{itbs}_2,\ \mathsf{btb},\ \mathsf{prog}) \\ & \qquad\qquad \textit{if} \ \ \mathsf{v} \neq 0 \quad \textit{and} \ \ \mathsf{pia} \neq \mathsf{ia}_1{+}1 \\ \longrightarrow & \mathsf{Proc}(\mathsf{ia}_1{+}1,\ \mathsf{rf},\ \mathsf{itbs}_1,\ \mathsf{btb}',\ \mathsf{prog}) \end{array}$$

We also refer to the *$\mathcal{P}_{\mathcal{S}}$-Jz-Jump-WrongSpec* and *$\mathcal{P}_{\mathcal{S}}$-Jz-NoJump-WrongSpec* rules as misprediction-recover rules.

5.4 Memory Access Rules

The memory access rules are very conservative in the sense that a memory operation can execute only when the Load or Store is the oldest instruction template buffer in the ITBs. This effectively prohibits any speculative Store instruction from modifying the memory incorrectly.

$\mathcal{P}_{\mathcal{S}}$-Load Rule

$$\begin{array}{ll} & \mathsf{Sys}(\mathsf{m},\ \ \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{ITB}(\mathsf{ia}_1,\ \mathsf{t}{:=}\mathsf{Load}(\mathsf{a}),\ \mathsf{wf},\ \mathsf{sf}) \oplus \mathsf{itbs},\ \mathsf{btb},\ \mathsf{prog})) \\ \longrightarrow & \mathsf{Sys}(\mathsf{m},\ \ \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{ITB}(\mathsf{ia}_1,\ \mathsf{t}{:=}\mathsf{m}[\mathsf{a}],\ \mathsf{wf},\ \mathsf{sf}) \oplus \mathsf{itbs},\ \mathsf{btb},\ \mathsf{prog})) \end{array}$$

$\mathcal{P}_{\mathcal{S}}$-Store Rule

$$\begin{array}{ll} & \mathsf{Sys}(\mathsf{m},\ \ \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{ITB}(\mathsf{ia}_1,\ \mathsf{Store}(\mathsf{a},\mathsf{v}),\ \mathsf{wf},\ \mathsf{sf}) \oplus \mathsf{itbs},\ \mathsf{btb},\ \mathsf{prog})) \\ \longrightarrow & \mathsf{Sys}(\mathsf{m}[\mathsf{a}{:=}\mathsf{v}],\ \ \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs},\ \mathsf{btb},\ \mathsf{prog})) \end{array}$$

If precise interrupt is not a concern, a simple optimization can allow a Load instruction to be performed if it is the oldest memory access instruction in the ITBs (but not necessarily in the oldest instruction template buffer). Similarly a Store instruction can be performed if there is no unresolved Jz or other memory access instructions in front of it in the ITBs. More aggressive memory access implementations and their impact on the program behavior in multiprocessor systems are discussed in Sections 8 and 9.

6 Correctness Proof of $\mathcal{P}_\mathcal{S}$ Model

In this section, we prove that the $\mathcal{P}_\mathcal{S}$ model is a correct implementation of the $\mathcal{AX}$ instruction set by showing that $\mathcal{P}_\mathcal{B}$ and $\mathcal{P}_\mathcal{S}$ can simulate each other in regards to some observation function. A natural observation function is the one that can extract all the programmer visible state, i.e., the program counter, the register file and the memory from the system. One can think of an observation function in terms of a print instruction that prints a part or the whole of the programmer visible state. If model A can simulate model B, then model A should be able to print whatever model B prints during the execution of any program.

6.1 Simulation of $\mathcal{P}_\mathcal{B}$ by $\mathcal{P}_\mathcal{S}$

It is trivial to show that $\mathcal{P}_\mathcal{S}$ can simulate $\mathcal{P}_\mathcal{B}$. A $\mathcal{P}_\mathcal{B}$ term can be "lifted" to a $\mathcal{P}_\mathcal{S}$ term by adding an empty ITBs and an arbitrary BTB to the processor.

Definition 1 ITBL (instruction-template-buffer-lift)
ITBL(Sys(m, Proc(ia, rf, prog))) $\equiv$ Sys(m, Proc(ia, rf, ϵ, btb, prog))
where btb is an arbitrary BTB.

Theorem 2 Let s_1 and s_2 be system terms in $\mathcal{P}_\mathcal{B}$. If $s_1 \twoheadrightarrow s_2$ in $\mathcal{P}_\mathcal{B}$, then ITBL(s_1) $\twoheadrightarrow$ ITBL(s_2) in $\mathcal{P}_\mathcal{S}$.

Proof: The following table illustrates the sequence of $\mathcal{P}_\mathcal{S}$ rules that can simulate each $\mathcal{P}_\mathcal{B}$ rule. For example, applying the *Op* rule in $\mathcal{P}_\mathcal{B}$ can be simulated by consecutively applying the $\mathcal{P}_\mathcal{S}$-*Op-Issue*, $\mathcal{P}_\mathcal{S}$-*Op* and $\mathcal{P}_\mathcal{S}$-*Value-Commit* rules in $\mathcal{P}_\mathcal{S}$.

$\mathcal{P}_\mathcal{B}$ rule	Sequence of $\mathcal{P}_\mathcal{S}$ rules with the same effect
Loadc	$\mathcal{P}_\mathcal{S}$-*Loadc-Issue*, $\mathcal{P}_\mathcal{S}$-*Value-Commit*
Loadpc	$\mathcal{P}_\mathcal{S}$-*Loadpc-Issue*, $\mathcal{P}_\mathcal{S}$-*Value-Commit*
Op	$\mathcal{P}_\mathcal{S}$-*Op-Issue*, $\mathcal{P}_\mathcal{S}$-*Op*, $\mathcal{P}_\mathcal{S}$-*Value-Commit*
Jz-Jump	$\mathcal{P}_\mathcal{S}$-*Jz-Issue*, $\mathcal{P}_\mathcal{S}$-*Jz-Jump-CorrectSpec* or $\mathcal{P}_\mathcal{S}$-*Jz-Issue*, $\mathcal{P}_\mathcal{S}$-*Jz-Jump-WrongSpec*
Jz-NoJump	$\mathcal{P}_\mathcal{S}$-*Jz-Issue*, $\mathcal{P}_\mathcal{S}$-*Jz-NoJump-CorrectSpec* or $\mathcal{P}_\mathcal{S}$-*Jz-Issue*, $\mathcal{P}_\mathcal{S}$-*Jz-NoJump-WrongSpec*
Load	$\mathcal{P}_\mathcal{S}$-*Load-Issue*, $\mathcal{P}_\mathcal{S}$-*Load*, $\mathcal{P}_\mathcal{S}$-*Value-Commit*
Store	$\mathcal{P}_\mathcal{S}$-*Store-Issue*, $\mathcal{P}_\mathcal{S}$-*Store*

6.2 Simulation of $\mathcal{P}_\mathcal{S}$ by $\mathcal{P}_\mathcal{B}$

It is a bit tricky to define a projection function from $\mathcal{P}_\mathcal{S}$ to $\mathcal{P}_\mathcal{B}$ because of the partially executed or speculatively executing instructions. We consider two approaches for the projection function; it captures the system state either after all the partially executed instructions are aborted, or after all the partially executed instructions are completed.

A mapping based on killing instruction in ITBs The effect of partially executed instructions in the $\mathcal{P_S}$ model can be nullified by setting the program counter to the address of the oldest instruction in the ITBs and aborting all the instructions in the ITBs. In other words, we can push the system state back as if the outstanding instructions in the ITBs had never been issued. Notice just the program counter need to be restored, because only the oldest instruction in the ITBs can write to the register file or the memory, and the corresponding instruction template buffer is freed immediately after the write.

Definition 3 ITBK (instruction-template-buffer-kill)

$$\mathsf{ITBK(\ Sys(m,\ \ Proc(ia,\ rf,\ } \epsilon, \mathsf{\ btb,\ prog))\ )}$$
$$\equiv \ \mathsf{Sys(m,\ \ Proc(ia,\ rf,\ prog))}$$
$$\mathsf{ITBK(\ Sys(m,\ \ Proc(ia,\ rf,\ ITB(ia_1,\ it,\ wf,\ sf)} \oplus \mathsf{itbs,\ btb,\ prog))\ )}$$
$$\equiv \ \mathsf{Sys(m,\ \ Proc(ia_1,\ rf,\ prog))}$$

A mapping based on flushing instructions in ITBs In $\mathcal{P_S}$, with instruction issue stalled, the ITBs will sooner or later become empty as instruction execution proceeds. In other words, we can always push the system state forward as if the outstanding instructions in the ITBs have all been completed. This motivates us to define another rewriting system $\mathcal{R_{ITBF}}$ which uses the same grammar as the $\mathcal{P_S}$ model and includes all the $\mathcal{P_S}$ rules except the instruction issue rules.

Definition 4 $\mathcal{R_{ITBF}} \equiv$ { *$\mathcal{P_S}$-Op, $\mathcal{P_S}$-Value-Forward, $\mathcal{P_S}$-Value-Commit, $\mathcal{P_S}$-Jz-Jump-CorrectSpec, $\mathcal{P_S}$-Jz-Jump-WrongSpec, $\mathcal{P_S}$-Jz-NoJump-CorrectSpec, $\mathcal{P_S}$-Jz-NoJump-WrongSpec, $\mathcal{P_S}$-Load, $\mathcal{P_S}$-Store* }

It can be shown by simple induction and case analysis that for any $\mathcal{P_S}$ system term, rewriting with respect to $\mathcal{R_{ITBF}}$ terminates within a finite number of steps, and always reaches the same normal form regardless of the order in which the rules are applied. In the TRS jargon, $\mathcal{R_{ITBF}}$ is said to be strongly terminating and confluent. It can be furthermore proved that the ITBs in the normal form is always empty.

We define ITBF(s) as "compute the normal form of s with respect to $\mathcal{R_{ITBF}}$ and then delete the empty ITBs and the BTB".

Definition 5 ITBF (instruction-template-buffer-flush)

Let $\mathsf{Sys(m,\ \ Proc(ia,\ rf,\ } \epsilon, \mathsf{\ btb,\ prog))}$ be the normal form of s with respect to $\mathcal{R_{ITBF}}$.

$$\mathsf{ITBF}(s) \ \equiv \ \mathsf{Sys(m,\ \ Proc(ia,\ rf,\ prog))}$$

ITBF is similar to the mapping function used in [9] to prove the correctness of an out-of-order processor. A proof that $\mathcal{P_B}$ can simulate $\mathcal{P_S}$ using ITBF is given in the Appendix. Here we present a proof based on ITBK, because the idea underlying this mapping function also works for multiprocessor systems.

Theorem 6 Let s_1 and s_2 be system terms in $\mathcal{P_S}$. If $s_1 \twoheadrightarrow s_2$ in $\mathcal{P_S}$, then $\mathsf{ITBK}(s_1) \twoheadrightarrow \mathsf{ITBK}(s_2)$ in $\mathcal{P_B}$.

Proof: By induction on rewriting steps. Assume $s_1 \longrightarrow s_2$ in $\mathcal{P}_\mathcal{S}$ by applying rule α. There are several cases on α:

- α is an instruction-issue rule. Then $\mathsf{ITBK}(s_1) = \mathsf{ITBK}(s_2)$.
- α is the $\mathcal{P}_\mathcal{S}$-*Op* or the $\mathcal{P}_\mathcal{S}$-*Value-Forward* rule. Then $\mathsf{ITBK}(s_1) = \mathsf{ITBK}(s_2)$.
- α is the $\mathcal{P}_\mathcal{S}$-*Value-Commit* rule. Let s_1 be the term:

  $\mathsf{Sys(m,\ \ Proc(ia,\ rf,\ ITB(ia_1,\ t{:=}v,\ Wreg(r),\ sf)} \oplus \mathsf{itbs,\ btb,\ prog))}$

  It can be shown that $\mathsf{prog[ia_1]}$ must be one of the following instructions:

  - $\mathsf{r{:=}Loadc(v)}$. Then $\mathsf{ITBK}(s_1) \longrightarrow \mathsf{ITBK}(s_2)$ by applying the *Loadc* rule in $\mathcal{P}_\mathcal{B}$.
  - $\mathsf{r{:=}Loadpc}$. We can prove that $\mathsf{v} = \mathsf{ia_1}$, therefore, $\mathsf{ITBK}(s_1) \longrightarrow \mathsf{ITBK}(s_2)$ by applying the *Loadpc* rule in $\mathcal{P}_\mathcal{B}$.
  - $\mathsf{r{:=}Op(r_1, r_2)}$ for some $\mathsf{r_1}$ and $\mathsf{r_2}$. We can prove that $\mathsf{v} = \underline{\mathsf{Op}}(\mathsf{rf[r_1], rf[r_2]})$, therefore, $\mathsf{ITBK}(s_1) \longrightarrow \mathsf{ITBK}(s_2)$ by applying the *Op* rule in $\mathcal{P}_\mathcal{B}$.
  - $\mathsf{r{:=}Load(r_1)}$ for some $\mathsf{r_1}$. We can prove that $\mathsf{v} = \mathsf{m[rf[r_1]]}$, therefore, $\mathsf{ITBK}(s_1) \longrightarrow \mathsf{ITBK}(s_2)$ by applying the *Load* rule in $\mathcal{P}_\mathcal{B}$.

- α is a branch completion rule. Then there are two cases:

  - if the Jz instruction is not at the head of the ITBs, then $\mathsf{ITBK}(s_1) = \mathsf{ITBK}(s_2)$;
  - if the Jz instruction is at the head of the ITBs. Let s_1 be the term:

    $\mathsf{Sys(m,\ \ Proc(ia,\ rf,\ ITB(ia_1,\ Jz(v, nia),\ wf,\ sf)} \oplus \mathsf{itbs,\ btb,\ prog))}$

    It can be shown that $\mathsf{prog[ia_1]} = \mathsf{Jz(r_1, r_2)}$ for some $\mathsf{r_1}$ and $\mathsf{r_2}$. We can prove that $\mathsf{v} = \mathsf{rf[r_1]}$ and $\mathsf{nia} = \mathsf{rf[r_2]}$, therefore, $\mathsf{ITBK}(s_1) \longrightarrow \mathsf{ITBK}(s_2)$ by applying the *Jz-Jump* or the *Jz-NoJump* rule in $\mathcal{P}_\mathcal{B}$, depending on if v is 0 or not.

- α is the $\mathcal{P}_\mathcal{S}$-*Load* rule. Then $\mathsf{ITBK}(s_1) = \mathsf{ITBK}(s_2)$.
- α is the $\mathcal{P}_\mathcal{S}$-*Store* rule. Let s_1 be the following system term:

  $\mathsf{Sys(m,\ \ Proc(ia,\ rf,\ ITB(ia_1,\ Store(a, v),\ wf,\ sf)} \oplus \mathsf{itbs,\ btb,\ prog))}$

  It can be shown that $\mathsf{prog[ia_1]} = \mathsf{Store(r_1, r_2)}$ for some $\mathsf{r_1}$ and $\mathsf{r_2}$. We can prove that $\mathsf{a} = \mathsf{rf[r_1]}$ and $\mathsf{v} = \mathsf{rf[r_2]}$, therefore, $\mathsf{ITBK}(s_1) \longrightarrow \mathsf{ITBK}(s_2)$ by applying the *Store* rule in $\mathcal{P}_\mathcal{B}$.

Therefore, if $s_1 \longrightarrow s_2$ in $\mathcal{P}_\mathcal{S}$, then $\mathsf{ITBK}(s_1) \twoheadrightarrow \mathsf{ITBK}(s_2)$ in zero or one rewriting steps in $\mathcal{P}_\mathcal{B}$. By induction, if $s_1 \twoheadrightarrow s_2$ in $\mathcal{P}_\mathcal{S}$, then $\mathsf{ITBK}(s_1) \twoheadrightarrow \mathsf{ITBK}(s_2)$ in $\mathcal{P}_\mathcal{B}$. □

7 Multiprocessor Systems

In this section, we show that the $\mathcal{P}_\mathcal{B}$ and $\mathcal{P}_\mathcal{S}$ can simulate each other in multiprocessor systems. A multiprocessor system consists of a shared memory and a set of processors.

$$\begin{array}{lcll} \text{SYS} & \equiv & \mathsf{Sys}(\text{MEM},\ \text{PG}) & \textit{System} \\ \text{PG} & \equiv & \epsilon \ \ [\!] \ \ \text{PROC} \mid \text{PG} & \textit{Processor Group} \end{array}$$

We can define multiprocessor systems $\mathcal{MP}_\mathcal{B}$ and $\mathcal{MP}_\mathcal{S}$ based on $\mathcal{P}_\mathcal{B}$ and $\mathcal{P}_\mathcal{S}$, respectively. In $\mathcal{MP}_\mathcal{B}$, each processor is a $\mathcal{P}_\mathcal{B}$ processor. All $\mathcal{P}_\mathcal{B}$ rules are preserved in $\mathcal{MP}_\mathcal{B}$, except the memory access rules are changed slightly to reflect the fact that the memory is shared by a number of processors.

$\mathcal{MP}_\mathcal{B}$-Load Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ Proc(ia,\ rf,\ prog)\mid pg)} \quad \textit{if}\ \ \mathsf{prog[ia] =\ r{:=}Load(r_1)} \\ \longrightarrow & \mathsf{Sys(m,\ Proc(ia{+}1,\ rf[r{:=}m[a]],\ prog)\mid pg)} \quad \textit{where}\ \ \mathsf{a = rf[r_1]} \end{array}$$

$\mathcal{MP}_\mathcal{B}$-Store Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ Proc(ia,\ rf,\ prog)\mid pg)} \quad \textit{if}\ \ \mathsf{prog[ia] =\ Store(r_1, r_2)} \\ \longrightarrow & \mathsf{Sys(m[a{:=}rf[r_2]],\ Proc(ia{+}1,\ rf,\ prog)\mid pg)} \quad \textit{where}\ \ \mathsf{a = rf[r_1]} \end{array}$$

In $\mathcal{MP}_\mathcal{S}$, each processor is a $\mathcal{P}_\mathcal{S}$ processor. While all the processor rules remain unchanged, the memory access rules are changed as following.

$\mathcal{MP}_\mathcal{S}$-Load Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ Proc(ia,\ rf,\ ITB(ia_1,\ t{:=}Load(a),\ wf,\ sf) \oplus itbs,\ btb,\ prog)\mid pg)} \\ \longrightarrow & \mathsf{Sys(m,\ Proc(ia,\ rf,\ ITB(ia_1,\ t{:=}m[a],\ wf,\ sf) \oplus itbs,\ btb,\ prog)\mid pg)} \end{array}$$

$\mathcal{MP}_\mathcal{S}$-Store Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ Proc(ia,\ rf,\ ITB(ia_1,\ Store(a,v),\ wf,\ sf) \oplus itbs,\ btb,\ prog)\mid pg)} \\ \longrightarrow & \mathsf{Sys(m[a{:=}v],\ Proc(ia,\ rf,\ itbs,\ btb,\ prog)\mid pg)} \end{array}$$

It can be proved that $\mathcal{MP}_\mathcal{B}$ and $\mathcal{MP}_\mathcal{S}$ can simulate each other in multiprocessor systems. To do this, we define a lifting function MITBL (multiprocessor ITBL) by adding an empty ITBs and a BTB to each processor. We define a projection function MITBK (multiprocessor ITBK) by deleting the ITBs and the BTB from each processor, and setting the program counter appropriately for each processor. The simulation theorems are as following:

Theorem 7 Let s_1 and s_2 be terms in $\mathcal{MP}_\mathcal{B}$. If $s_1 \twoheadrightarrow s_2$ in $\mathcal{MP}_\mathcal{B}$, then $\mathsf{MITBL}(s_1) \twoheadrightarrow \mathsf{MITBL}(s_2)$ in $\mathcal{MP}_\mathcal{S}$.

Theorem 8 Let s_1 and s_2 be terms in $\mathcal{MP}_\mathcal{S}$. If $s_1 \twoheadrightarrow s_2$ in $\mathcal{MP}_\mathcal{S}$, then $\mathsf{MITBK}(s_1) \twoheadrightarrow \mathsf{MITBK}(s_2)$ in $\mathcal{MP}_\mathcal{B}$.

The proofs of the above theorems are similar to the ones shown in the previous section.

It is worth noting that we cannot prove that $\mathcal{MP}_\mathcal{B}$ can simulate $\mathcal{MP}_\mathcal{S}$ by using a multiprocessor version of the ITBF function. The potential memory access race implies that $\mathcal{R}_{\mathcal{ITBF}}$ is not confluent in multiprocessor systems. In other words, non-determinism can happen if two processors intend to access the same memory location at the same time, and at least one of them is a Store operation.

8 An Aggressive Implementation of Memory Operations

Unnecessary constraint imposed on memory accesses can dramatically degrade the system performance. Memory access instructions can be implemented more aggressively while still preserving the semantics for single processor systems. Various optimization techniques such as write buffers and non-blocking loads can be used to reduce or hide memory access latencies. However, these techniques are often aimed at performance optimization for sequential programs. Different memory access implementations may behave very differently in multiprocessor systems, and the difference can be very subtle.

The following rules suggest an aggressive implementation of memory operations in which memory accesses can be performed in arbitrary order, provided the data dependences imposed by the program order are not violated. The load rule allows a Load instruction to read the memory if there is no outstanding Store instruction in front of it in the ITBs that may write in the same memory location. The store rule allows a Store instruction to write the memory if it is not on a speculative path, and there is no other outstanding Load or Store instruction in front of it in the ITBs that may read or write the memory location. (Tag t' represents an unresolved addresses).

$\mathcal{P_S}$-Load Rule

$\mathsf{Sys(m,\ Proc(ia,\ rf,\ itbs_1 \oplus ITB(ia_1,\ t{:=}Load(a),\ wf,\ sf) \oplus itbs_2,\ btb,\ prog))}$
$\quad\quad$ *if* $\mathsf{Store(a,\text{-}),\ Store(t',\text{-}) \notin itbs_1}$
$\longrightarrow\ \mathsf{Sys(m,\ Proc(ia,\ rf,\ itbs_1 \oplus ITB(ia_1,\ t{:=}m[a],\ wf,\ sf) \oplus itbs_2,\ btb,\ prog))}$

$\mathcal{P_S}$-Store Rule

$\mathsf{Sys(m,\ Proc(ia,\ rf,\ itbs_1 \oplus ITB(ia_1,\ Store(a,v),\ wf,\ sf)\ \oplus itbs_2,\ btb,\ prog))}$
$\quad\quad$ *if* $\mathsf{Jz,\ Load(a),\ Load(t'),\ Store(a,\text{-}),\ Store(t',\text{-}) \notin itbs_1}$
$\longrightarrow\ \mathsf{Sys(m[a:=v],\ \ Proc(ia,\ rf,\ itbs_1 \oplus itbs_2,\ btb)\ prog)}$

Intuitively, the predicate of the load rule maintains the data-dependence (read-after-write), while the predicate of the store rule ensures that the anti-dependence (write-after-read) and the output dependence (write-after-write) cannot be violated. The correctness of these rules in a uniprocessor setting can be proved using a projection function that flushes the ITBs. It is interesting to note that simply aborting instructions in ITBs does not give a mapping function from $\mathcal{P_S}$ to $\mathcal{P_B}$.

9 Effect of Aggressive Memory Operations on Multiprocessors

The memory access rules discussed in the previous section can produce very different results for parallel programs in multiprocessor systems. For example, consider the following program. (Instructions are listed in the program order. Assume initially on both processors, registers $\mathsf{r_1}$ and $\mathsf{r_2}$ contain addresses $\mathsf{a_1}$ and $\mathsf{a_2}$, respectively):

processor 1	processor 2
$\mathsf{r}_3 := \mathsf{Loadc}(1);$	$\mathsf{r}_3 := \mathsf{Loadc}(2);$
$\mathsf{Store}(\mathsf{r}_1, \mathsf{r}_3);$	$\mathsf{Store}(\mathsf{r}_2, \mathsf{r}_3);$
$\mathsf{Store}(\mathsf{r}_2, \mathsf{r}_3);$	$\mathsf{Store}(\mathsf{r}_1, \mathsf{r}_3);$

When the program execution terminates on both processors, it is possible that memory location a_1 has value 1, while memory location a_2 has value 2. However, this cannot happen with the memory access rules defined in the $\mathcal{MP}_\mathcal{B}$ model.

Even more bizarre behavior can be observed since Load operations on the same location can be performed in an order different from the program order. In the program given below, assume initially on both processors, register r_1 contains address a:

processor 1	processor 2
$\mathsf{r}_2 := \mathsf{Loadc}(1);$	$\mathsf{r}_2 := \mathsf{Loadc}(2);$
$\mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2);$	$\mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2)$
$\mathsf{r}_3 := \mathsf{Load}(\mathsf{r}_1);$	$\mathsf{r}_3 := \mathsf{Load}(\mathsf{r}_1);$
$\mathsf{r}_4 := \mathsf{Load}(\mathsf{r}_1);$	$\mathsf{r}_4 := \mathsf{Load}(\mathsf{r}_1);$
$\mathsf{r}_5 := \mathsf{Load}(\mathsf{r}_1);$	$\mathsf{r}_5 := \mathsf{Load}(\mathsf{r}_1);$

When the execution terminates on both processors, it is possible that in processor 1, registers r_3, r_4 and r_5 hold values 1, 2 and 1, respectively, while in processor 2, registers r_3, r_4 and r_5 hold values 2, 1 and 2, respectively. The Store operations for memory location a are observed in different orders, and a later Load instruction can read an older value. (If this result is not desired, we can append the predicate of the load rule with an extra condition "$\mathsf{Load}(\mathsf{a})$, $\mathsf{Load}(\mathsf{t}') \notin \mathsf{itbs}_1$").

It is common to encounter design alternatives in the memory interface design where the implication of a choice on the behavior of programs is not completely clear. For example, what is the consequence of adding a short-circuiting rule that allows a Load operation to read from the ITBs if there is an outstanding Store in front of it in the ITBs which is to write in the same location?

```
$\mathcal{P}_\mathcal{S}$-Load-Short-Circuiting Rule
     Proc(ia, rf,
          $\mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{Store}(\mathsf{a},\mathsf{v}), \mathsf{wf}_1, \mathsf{sf}_1) \oplus \mathsf{itbs}_2 \oplus \mathsf{ITB}(\mathsf{ia}_2, \mathsf{t} := \mathsf{Load}(\mathsf{a}), \mathsf{wf}_2, \mathsf{sf}_2) \oplus \mathsf{itbs}_3$,
          btb, prog)
               if  $\mathsf{Load}(\mathsf{a}), \mathsf{Load}(\mathsf{t}'), \mathsf{Store}(\mathsf{a},\text{-}), \mathsf{Store}(\mathsf{t}',\text{-}) \notin \mathsf{itbs}_2$
$\longrightarrow$  Proc(ia, rf,
          $\mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{Store}(\mathsf{a},\mathsf{v}), \mathsf{wf}_1, \mathsf{sf}_1) \oplus \mathsf{itbs}_2 \oplus \mathsf{ITB}(\mathsf{ia}_2, \mathsf{t} := \mathsf{v}, \mathsf{wf}_1, \mathsf{sf}_1) \oplus \mathsf{itbs}_3$,
          btb, prog)
```

We conjecture that this newly added rule does not affect the observable behavior of the processor even in the multiprocessor setting. Precise modeling of memory access instructions allows us to carefully examine issues like this and their impact on memory consistency models.

Aggressive memory access implementations usually require extra instructions that act as "memory fences" in order to be able to realize reasonable memory models such as sequential consistency.

10 Conclusions and Work-In-Progress

It is worth emphasizing that the proof technique presented in this paper is quite general and the definition of the mapping (lifting and projection) functions is straightforward. In [8] we defined sequential consistency based on the $\mathcal{P}_\mathcal{B}$ model, and designed a family of cache coherence protocols for a distributed shared-memory system with hierarchical caches. The correctness of the cache coherence protocols was proved by showing that the TRS's for the protocols and the memory model can simulate each other. Our experience shows that the technique not only makes protocol verification more systematic, but also helps us in designing adaptive protocols by successive refinement. We are now exploring the processor-memory interface that may lead to more aggressive implementations of memory access and synchronization instructions in multiprocessor systems.

We are also exploring hardware synthesis from the type of TRS's presented in this paper. The preliminary result based on hand compilation of TRS rules into synthesizable Verilog looks promising. The goal is to produce an architecture description language and a compiler that will dramatically reduce the design effort to implement complex systems.

The use of formal techniques in designing systems partially depends upon the tools available to support the technique. We have just begun the investigation of appropriate tools to support our technique so that the tedious case analysis can be performed by machine. It should be possible to build a model checker type of tool to explore all the reductions of a given term. Model checkers like Murphi [2] verify assertions by exploring a finite state graph. When a problem can be expressed without using too many states, such tools have proven very useful as debuggers for engineers in verifying properties of their designs.

Many of our systems can be expressed using other formal techniques such as I/O automata [6]. Techniques based on general theorem proving systems, such as HOL, let the user express more general assertions but require more help from the user in actually doing the proofs. Like TRS's, assertions in none of these formalisms can be automated fully due to the infinite number of states. Nevertheless, useful commercial tools are available to verify that an implementation satisfies its specifications.

Acknowledgement Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Ft Huachuca contract DABT63-95-C-0150.

References

[1] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *International Conference on Computer-Aided Verification*, June 1994.

[2] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1996.

[4] J. W. Klop. Term Rewriting System. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[5] J. Levitt and K. Olukotun. A Scalable Formal Verification Methodology for Pipelined Microprocessors. In *33nd ACM IEEE Design Automation Conference*, June 1996.

$$\begin{tikzcd}
s_1 \arrow[r, "\beta"] \arrow[d, "\alpha"] & s_3 \\
s_2 \arrow[ur, "\beta"] &
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
s_1 \arrow[rr, "\beta"] \arrow[d, "\alpha"] & & s_3 \arrow[d, "\alpha"] \\
s_2 \arrow[r, two heads, "\text{value-forward}"] & s_2' \arrow[r, "\beta"] & s_4
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
s_1 \arrow[r, "\beta"] \arrow[d, "\alpha"] & s_3 \arrow[d, "\alpha"] \\
s_2 \arrow[r, "\beta"] & s_4
\end{tikzcd}$$

(c)

Figure 4: Simulate Instruction Issue Rules

[6] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[7] X. Shen and Arvind. Processor Models. CSG Memo 400, Laboratory for Computer Science, MIT, June 1997.

[8] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. CSG Memo 398, Laboratory for Computer Science, MIT, June 1997.

[9] X. Shen and Arvind. Modeling and Verification of ISA Implementations. In *Proceedings of the Australasian Computer Architecture Conference, Perth, Australia,* Feb. 1998.

[10] P. J. Windley. Formal Modeling and Verification of Microprocessors. *IEEE Transactions on Computers,* 44(1), Jan. 1995.

Appendix: Simulation of $\mathcal{P}_{\mathcal{S}}$ by $\mathcal{P}_{\mathcal{B}}$ Using ITBF

Different mapping functions can be employed in the simulation proof. In this section, we prove by induction on rewriting steps that $\mathcal{P}_{\mathcal{B}}$ can simulate $\mathcal{P}_{\mathcal{S}}$ with respect to ITBF defined in Section 6. The simulation theorem is as following:

Theorem 9 Let s_1 and s_2 be system terms in $\mathcal{P}_{\mathcal{S}}$. If $s_1 \twoheadrightarrow s_2$ in $\mathcal{P}_{\mathcal{S}}$, then $\mathsf{ITBF}(s_1) \twoheadrightarrow \mathsf{ITBF}(s_2)$ in $\mathcal{P}_{\mathcal{B}}$.

Proof: It is trivial to show that ITBF maps the initial $\mathcal{P}_{\mathcal{S}}$ term to the initial $\mathcal{P}_{\mathcal{B}}$ term. Assume $s_1 \longrightarrow s_2$ in $\mathcal{P}_{\mathcal{S}}$ by applying rule α. There are two cases on α:

- $\alpha \in \mathcal{R}_{\mathcal{ITBF}}$. Needless to say, $\mathsf{ITBF}(s_1)$ and $\mathsf{ITBF}(s_2)$ are identical.
- $\alpha \notin \mathcal{R}_{\mathcal{ITBF}}$ (i.e. α is an instruction-issue rule). In this case, we can show that either $\mathsf{ITBF}(s_1)$ and $\mathsf{ITBF}(s_2)$ are identical, or $\mathsf{ITBF}(s_1)$ can be rewritten to $\mathsf{ITBF}(s_2)$ by applying an appropriate $\mathcal{P}_{\mathcal{B}}$ rule.

  Suppose $s_1 \longrightarrow s_3$ by applying some $\mathcal{R}_{\mathcal{ITBF}}$ rule, say β. There are two cases on β:

  – β is a misprediction-recover rule (i.e. the $\mathcal{P}_{\mathcal{S}}$-*Jz-Jump-WrongSpec* or $\mathcal{P}_{\mathcal{S}}$-*Jz-NoJump-WrongSpec* rule). It is trivial to show that $s_2 \longrightarrow s_3$ by applying β, since the instruction is issued on a wrong speculative path, and the issuing will be nullified by β (see Figure 4a).

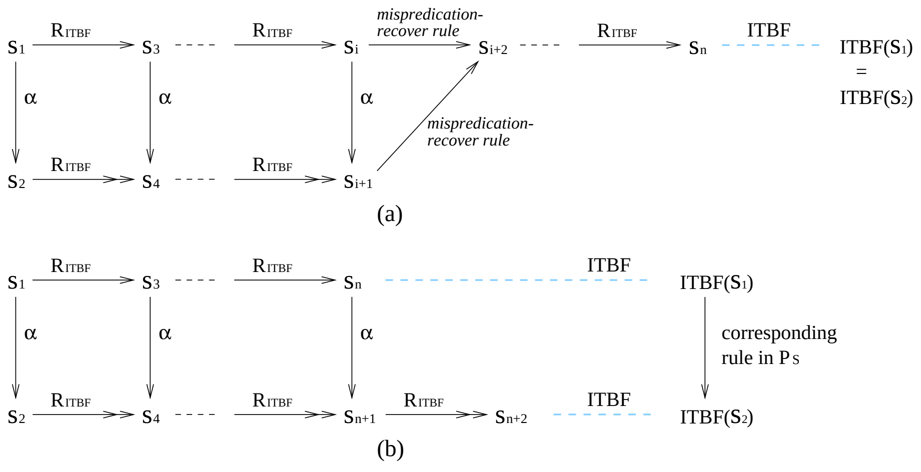


Figure 5: Simulate Instruction Issue Rules

- β is not a misprediction-recover rule. It can be seen by inspecting the $\mathcal{R}_{\mathcal{ITBF}}$ rules that α can also be applied to s_3. Assume $s_3 \longrightarrow s_4$ by applying α.

  If α is the $\mathcal{P}_{\mathcal{S}}$-*Value-Commit* rule, and the register to which the value is committed is referenced as an operand register in the instruction issued by α, then $s_2 \twoheadrightarrow s_4$ by first applying the *ValueForward* rule one or two times, and then applying β (see Figure 4b). Otherwise $s_2 \longrightarrow s_4$ by applying β (see Figure 4c).

Let s_n be the normal form of s_1 with respect to $\mathcal{R}_{\mathcal{ITBF}}$. There are two cases:

- if the rewriting from s_1 to s_n invokes a misprediction-recover rule, then there exist terms s_i, s_{i+1} and s_{i+2} such that $s_i \longrightarrow s_{i+1}$ by applying α, $s_i \longrightarrow s_{i+2}$ and $s_{i+1} \longrightarrow s_{i+2}$ by applying the misprediction-recover rule. This implies that s_1 and s_2 have the same normal form with respect to $\mathcal{R}_{\mathcal{ITBF}}$. In other words, $\mathsf{ITBF}(s_1)$ and $\mathsf{ITBF}(s_2)$ are identical (see Figure 5a).
- if the rewriting from s_1 to s_n does not invoke any misprediction-recover rule, then by induction α can be applied to s_n to yield s_{n+1} such that $s_2 \twoheadrightarrow s_{n+1}$ by applying just $\mathcal{R}_{\mathcal{ITBF}}$ rules.

  Furthermore, suppose s_{n+2} is the normal form of s_{n+1} with respect to $\mathcal{R}_{\mathcal{ITBF}}$. Since s_n and s_{n+2} both have empty instruction template buffers, it can be easily shown that $\mathsf{ITBF}(s_n) \longrightarrow \mathsf{ITBF}(s_{n+2})$ by applying the corresponding $\mathcal{P}_{\mathcal{B}}$ rule (see Figure 5b).

  The table below gives the correspondence between the $\mathcal{P}_{\mathcal{S}}$ instruction-issue rules and the $\mathcal{P}_{\mathcal{B}}$ rules.

$\mathcal{P}_\mathcal{S}$ instruction-issue rule	corresponding $\mathcal{P}_\mathcal{B}$ rule
$\mathcal{P}_\mathcal{S}$*-Loadc-Issue rule*	*Loadc rule*
$\mathcal{P}_\mathcal{S}$*-Loadpc-Issue rule*	*Loadpc rule*
$\mathcal{P}_\mathcal{S}$*-Op-Issue rule*	*Op rule*
$\mathcal{P}_\mathcal{S}$*-Jz-Issue rule*	*Jz-Jump / Jz-NoJump rule*
$\mathcal{P}_\mathcal{S}$*-Load-Issue rule*	*Load rule*
$\mathcal{P}_\mathcal{S}$*-Store-Issue rule*	*Store rule*

This completes the proof that if $s_1 \twoheadrightarrow s_2$ in $\mathcal{P}_\mathcal{S}$, then $\mathsf{ITBF}(s_1) \twoheadrightarrow \mathsf{ITBF}(s_2)$ in $\mathcal{P}_\mathcal{B}$. By induction, if $s_1 \twoheadrightarrow s_2$ in $\mathcal{P}_\mathcal{S}$, then $\mathsf{ITBK}(s_1) \twoheadrightarrow \mathsf{ITBK}(s_2)$ in $\mathcal{P}_\mathcal{B}$. $\Box$

Some technical details are omitted, and the complete proof can be found in [7].