Formal Verification of Explicitly Parallel Microprocessors

Byron Cook, John Launchbury, John Matthews, and Dick Kieburtz

Oregon Graduate Institute
{byron,jl,johnm,dick}@cse.ogi.edu

Abstract. The trend in microprocessor design is to extend instruction-set architectures with features—such as parallelism annotations, predication, speculative memory access, or multimedia instructions—that allow the compiler or programmer to express more instruction-level parallelism than the microarchitecture is willing to derive. In this paper we show how these instruction-set extensions can be put to use when formally verifying the correctness of a microarchitectural model. Inspired by Intel's IA-64, we develop an explicitly parallel instruction-set architecture and a clustered microarchitectural model. We then describe how to formally verify that the model implements the instruction set. The contribution of this paper is a specification and verification method that facilitates the decomposition of microarchitectural correctness proofs using instruction-set extensions.

1 Introduction

Simple instruction sets were once the fashion in microprocessor design because they gave a microarchitect more freedom to exploit instruction-level parallelism [29]. However, as microarchitectural optimizations have become more complex, their effect on microprocessor performance has begun to diminish [5,7,9,31]. The trouble is that sophisticated implementations of impoverished instruction sets come at a cost. Superscalar out-of-order microarchitectures [22], for example, lead to larger and hotter microprocessors. They are difficult to design and debug, and typically have long critical paths, which inhibit fast clock speeds.

This explains why microprocessor designers are now adding constructs to instruction sets that allow the explicit declaration of instruction level parallelism. Here are a few examples:

Parallelism annotations declare which instructions within a program can be executed out of order. This feature appears in Intel's IA-64 [5,8,12,15,24,34], and indications are that Compaq [37] and others are exploring similar approaches.

Predication [2] expresses conditional execution using data dependence rather than branch instructions. IA-64 and ARM [20] are examples of predicated instruction-sets.

L. Pierre and T. Kropf (Eds.): CHARME'99, LNCS 1703, pp. 23–36, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Speculative loads [8] behave like traditional loads—however exceptions are raised only when and only if the data they fetch is used. IA-64 and PA-RISC [21] both support speculative loading mechanisms.

Multimedia instructions denote where optimizations specific to multimedia computation can be applied. MMX [3], AltiVec [13], and 3DNow [14,32] are examples of multimedia specific instruction-set extensions.

What do these new instruction-sets look like? How will we verify microarchitectural designs against them? These are the questions that we hope to address. In this paper we develop a modeling and verification method for extended instruction-sets and microarchitectures that facilitates the decomposition of microarchitectural correctness proofs. Using this method we construct a formal specification for an instruction-set architecture like Intel's IA-64; and a microarchitectural design that draws influence from Compaq's 21264 [11] and Intel's Merced [34]. We demonstrate how to decompose the microarchitectural correctness proof using the extensions from the specification. We then survey related work, and propose several directions for future research.

2 OA-64: An Explicitly Parallel Instruction Set

In this section we develop a specification of an explicitly parallel instruction-set, called the *Oregon Architecture* (OA-64), which is based on Intel's IA-64. OA-64 extends a traditional reduced instruction set with parallelism annotations and predication.

To see how these extensions fit into OA-64, look at Fig. 1 which contains an OA-64 implementation of the factorial function:

- An OA-64 program is a finite sequence of *packets*, where each packet consists of three instructions. Programs are addressed at the packet-level. That is, instructions are fetched in packets, and branches can jump only to the beginning of a packet.
- Instructions are annotated with *thread identifiers*. For example, the `0` in the `load` instruction declares that instructions with thread identifiers that are not equal to `0` can be executed in any order with respect to the `load` instruction.
- Packets can be annotated with the directive `FENCE`, which directs the machine to calculate all in-flight instructions before executing the following packet.
- Instructions are predicated on boolean-valued predicate registers. For example, the `load` instruction will only be executed if the value of `p5` is true in the current predicate register-file state.

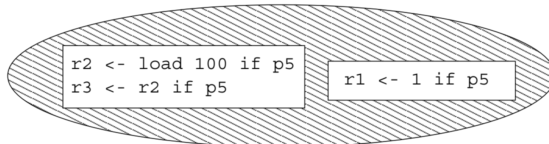
One way to view the parallelism annotations (thread identifiers and fences) is with directed graphs whose nodes are sets of threads that occur between fence directives. We call the sets of threads *regions*. The idea is that an OA-64 machine will execute one region at a time. In this manner, all values computed in previously executed regions are available to all threads in the current region.

```
101:  r2 ← load 100 if p5 in 0
      r3 ← r2 if p5 in 0
      r1 ← 1 if p5 in 1
FENCE

102:  p2,p3 ← r2 != 0 if p5 in 0
      r3 ← r2 if p5 in 1
      nop
FENCE

103:  r1 ← r1 * r3 if p2 in 0
      r2 ← r2 - 1 if p2 in 1
      pc ← 102 if p2 in 2


104:  store 401 r1 if p3 if 3
      pc ← 33 if p3 in 4
      nop
FENCE
```

Fig. 1. OA-64 factorial function.

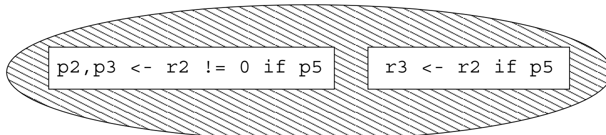
For example, the first packet loads data into `r2` and initializes the values of registers `r1` and `r3`. Because `r3` depends on the value of `r2`, the `load` instruction must be executed before writing to `r3` — this is expressed by placing the same thread-identifier (`0`) on the two instructions. The calculation of `r1`, however, can be executed in any order with respect to the `0` thread.

The fence directive in packet `101` instructs the machine to retire the active threads before executing the following packet. Assuming that packet `100` also issues a fence directive, packet `101` forms its own region:



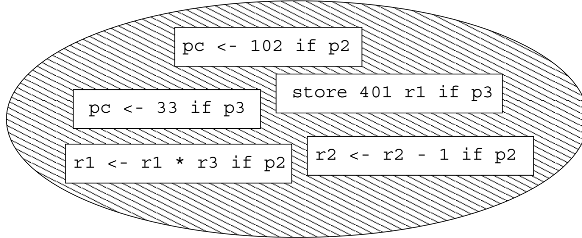
where boxes represent threads. Instructions within a thread must be executed in order. Threads, however, can be executed in any interleaving-order with other threads. Because packet `101` is a region the machine is required to synchronize the state before executing the next packet.

Because packet `102` also issues a fence, it forms its own region:



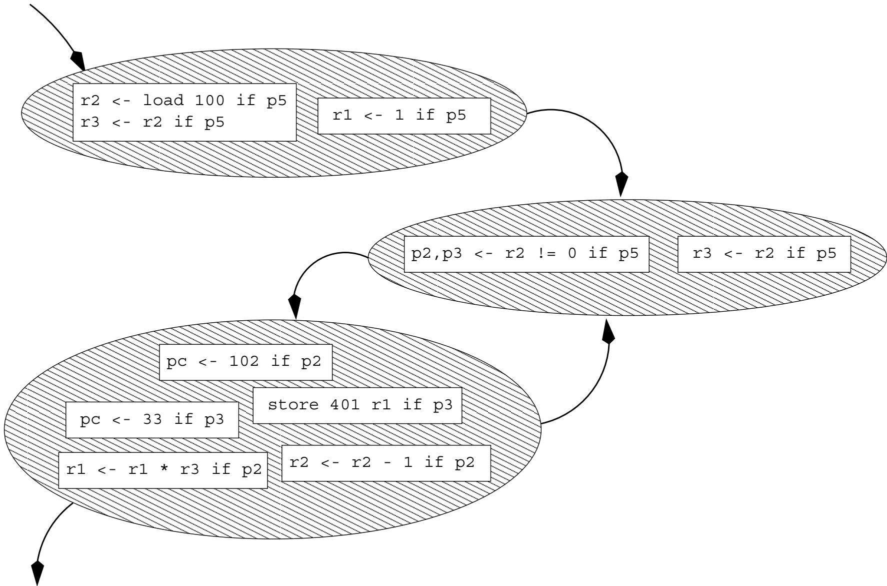
The comparison instruction sets the predicate register `p2` to true if `r2` is not equal to 0. The value of `p3` is set to the negation of `p2`.

Because packet `103` is not fenced, but packet `104` is, the next region is formed from packets `103` and `104`:



This region contains 5 singleton threads. Note that, if both `p2` and `p3` were true, two threads would write to the program counter (`pc`) in an arbitrary order. However, because `p2` and `p3` are the negation of one another, for a given run of the region only one thread will write to `pc`.

Assignments to the program counter within a region are visible to the machine's fetch mechanism only after a fence directive has been issued. That is, assignments to `pc` tell the machine where to fetch from after executing the next fence. Therefore, a trace of an OA-64 program can be viewed as an infinite path through the finite directed graph formed by regions and their successors:



2.1 An OA-64 Specification

As is standard practice, we specify OA-64 as a state machine. However, rather than a single monolithic machine, OA-64's specification is the composition of

functions from machines to machines:

$$\texttt{oa64}_p = \texttt{fnt}_p \circ \texttt{cls} \circ \texttt{prd} \circ \texttt{risc}$$

We call these functions *state machine transformers*.

Each transformer specifies an extension such as instruction caching, parallelism or predication. The transformer `risc`, for example, ignores the incoming state machine and constructs a classic reduced instruction-set. The instructions defined by `risc` are then predicated by `prd`. The parallelism transformer `cls` then adds a notion of threads, thread identifiers, and synchronization. The final transformer `fnt` builds a state machine with program memory, a program counter, and instruction fetching.

Transformers accept and return state machines with the type constructor `SM` in their type. $\texttt{SM}\ \Sigma\ \Gamma\ \Delta$ represents the set of state machines with state Σ, input-type Γ and observation-type Δ (Σ, Γ, and Δ are polymorphic type variables). Given a state machine $m :: \texttt{SM}\ \Sigma\ \Gamma\ \Delta$, three projections are defined:

- $\texttt{initial}_m :: \Sigma$ is the initial state.
- $\texttt{observe}_m :: \Sigma \rightarrow \Delta$ is an observation function.
- $\texttt{next}_m :: \Gamma \rightarrow \Sigma \rightarrow \{\Sigma\}$ is a next-state relation indexed by Γ.

For example, the transformer `prd`, which has type:

$$\begin{aligned}\texttt{prd} :: \texttt{BUBBLE} \in \Gamma \wedge \texttt{BIND}\ \Psi\ \Phi \in \Gamma \Rightarrow \texttt{SM}\ \Sigma\ \Gamma\ (\texttt{Maybe}\ (\Psi \rightarrow \Phi)) \rightarrow \\ \texttt{SM}\ (\texttt{Prd_St}\ \Psi\ \Sigma)\ (\texttt{Prd_Instr}\ \Psi\ \Gamma)\ (\texttt{Maybe}\ (\Psi \rightarrow \Phi))\end{aligned}$$

expects a state machine with an environment observation-type and returns a new state machine with the same observation-type but slightly richer input- and state-types. The type constructor `Maybe` adds a "bottom" element to a type:

$$\texttt{Maybe}\ (\Psi \rightarrow \Phi) = \texttt{Nothing} \mid \texttt{Just}\ (\Psi \rightarrow \Psi)$$

That is, an environment of type $\texttt{Maybe}\ (\Psi \rightarrow \Phi)$ is either available ($\texttt{Just}\ f$ where the value f has type $\Psi \rightarrow \Phi$) or unavailable (`Nothing`). The type constructor `Prd_Instr`, when given an instruction type Γ and register-type Ψ, returns a type that represents tagged expressions that can be either register to predicate-register moves ($\texttt{TO}\ \Psi\ \Psi$), predicate-register to register moves ($\texttt{FROM}\ \Psi\ \Psi$), or instructions predicated by a predicate register ($\texttt{IF}\ \Gamma\ \Psi$):

$$\texttt{Prd_Instr}\ \Psi\ \Gamma = \texttt{TO}\ \Psi\ \Psi \mid \texttt{FROM}\ \Psi\ \Psi \mid \texttt{IF}\ \Gamma\ \Psi \mid \ldots$$

The type constructor `Prd_St`, when given a state-type and register-type, returns the state-type paired with the type that represents a predicate register file:

$$\texttt{Prd_St}\ \Psi\ \Sigma = (\Sigma, \texttt{Env}\ \Psi\ \texttt{Bool})$$

The function `prd`, found in Fig. 2, takes a state machine and uses its components to build a new predicated state machine.

$$
\begin{array}{l}
\texttt{prd}\ m = \\
\quad \{\ \texttt{observe} = \lambda(s,p).\ \texttt{observe}_m\ s \\
\quad ,\ \texttt{initial} = (\texttt{initial}_m, \texttt{emptyEnv True}) \\
\quad ,\ \texttt{next} = \lambda i.\ \lambda(s,e).\ \textbf{if}\ \texttt{stall}_m\ s\ \textbf{then}\ (\texttt{next}_m\ \texttt{BUBBLE}\ s, e) \\
\qquad \textbf{else case}\ i\ \textbf{of} \\
\qquad\quad \texttt{BIND}\ r\ w \rightarrow (\texttt{next}_m\ (\texttt{BIND}\ r\ w)\ s, e) \\
\qquad \mid \texttt{BUBBLE}\ r\ w \rightarrow (\texttt{next}_m\ \texttt{BUBBLE}\ s, e) \\
\qquad \mid \texttt{TO}\ r\ r' \rightarrow (s, \texttt{updateEnv}\ e\ (r, \texttt{read}_m\ s\ r' \neq 0)) \\
\qquad \mid \texttt{FROM}\ r\ r' \rightarrow \textbf{if}\ \texttt{readEnv}\ e\ r'\ \textbf{then}\ (\texttt{next}_m\ (\texttt{BIND}\ r\ 1)\ s, e) \\
\qquad\qquad\qquad\qquad \textbf{else}\ (\texttt{next}_m\ (\texttt{BIND}\ r\ 0)\ s, e) \\
\qquad \mid \texttt{IF}\ i\ r \rightarrow \textbf{if}\ \texttt{readEnv}\ e\ r\ \textbf{then}\ (\texttt{next}_m\ i\ s, e)\ \textbf{else}\ (s, e) \\
\quad \}
\end{array}
$$

Fig. 2. Predication state machine transformer

As is the case for `prd`, we have found that most transformers expect that state machines make visible an environment and a stalling-bit. Let $m :: \texttt{SM}\ \Sigma\ \Gamma$ $(\texttt{Maybe}\ (\Psi \rightarrow \Phi))$, we can define the following functions on the states of m:

$$
\begin{array}{ll}
\texttt{read}_m & :: \Sigma \rightarrow \Psi \rightarrow \Phi \\
\texttt{read}_m & = \lambda s.\ \textbf{case}\ \texttt{observe}_m\ s\ \textbf{of}\ \texttt{Just}\ e \rightarrow e \mid \texttt{Nothing} \rightarrow \textbf{undefined}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{stall}_m & :: \Sigma \rightarrow \texttt{Bool} \\
\texttt{stall}_m & = \lambda s.\ \textbf{case}\ \texttt{observe}_m\ s\ \textbf{of}\ \texttt{Just}\ e \rightarrow \textbf{False} \mid \texttt{Nothing} \rightarrow \textbf{True}
\end{array}
$$

The partial function $\texttt{read}_m$ when given a state of type Σ and a register reference of type Ψ returns a value of type Φ.

Transformers usually expect that $\texttt{BIND}\ \Psi\ \Phi$ is an instruction in the instruction-type that allows for the update of the environment. That is,

$$\forall s, r, w.\ \neg\texttt{stall}_m\ s \Rightarrow \texttt{read}_m(\texttt{next}_m\ s\ (\texttt{BIND}\ r\ w))\ r = w$$

We also expect that a `BIND` cannot bring a machine out of a stalling state:

$$\forall s, r, w.\ \texttt{stall}_m\ s \Rightarrow \texttt{stall}_m(\texttt{next}_m\ s\ (\texttt{BIND}\ r\ w))$$

3 Columbia: An OA-64 Microarchitecture

This section describes the OA-64 microarchitecture—named Columbia—pictured in Fig. 3. Columbia employs three independent execution clusters (buffered execution pipelines); though, in principle, it could use many more clusters. Columbia fetches packets from the instruction cache (**ICache**) and feeds them to the clusters. In the case that a packet contains a fence directive, the machine stops fetching instructions until all of the clusters have been flushed. The **Route** unit schedules fetched instructions into execution clusters based on their

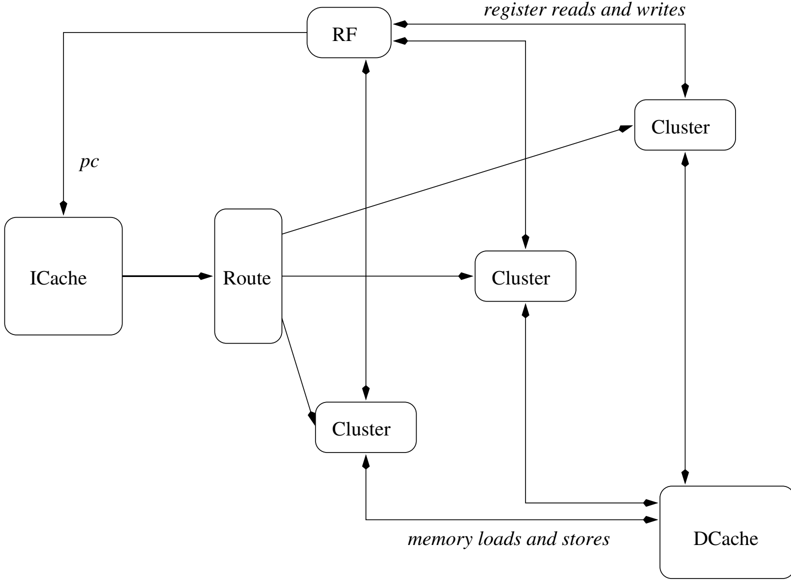


Fig. 3. Columbia data paths— pictured with three clusters

thread-identifier (modulus 3). The fetch logic uses the register-file's program counter value. It determines, based on whether or not the machine is still servicing a fence directive, if the program counter should be used (i.e. the machine has finished processing a region).

Like OA-64, we can construct Columbia's formal model as the composition of transformers:

$$\texttt{columbia}_p = \texttt{fnt}_p \circ \texttt{cls}' \circ \texttt{prd}' \circ \texttt{pipeline}$$

The transformer `pipeline` constructs a pipeline like that found in Hennessy & Patterson [16]. Then $\texttt{prd}'$ adds a predicate register-file and appropriately locks the pipeline in the case of an instruction stream like:

$$\texttt{r2} \leftarrow \cdots ; \quad \texttt{p2} \leftarrow \texttt{r2}$$

The transformer $\texttt{cls}'$ maintains three parallel buffered and pipelines. That is, rather than finding one instruction to issue at each cycle, $\texttt{cls}'$ maintains an instruction buffer for each pipeline from which it issues up to three instructions in parallel every cycle.

4 Decomposing Correctness with Transformers

In this section we introduce *transformer decomposition*—which allows us to verify machines by point-wise verification of their transformers. For example, does

`pipeline` implement `risc`?, does `cls'` implement `cls`?, etc. We assume that the correctness criteria, abstractly denoted as $\sqsubseteq$, is a reflexive and transitive relation over `SM` (this could be trace containment [1], simulation [27], correspondence [38], etc). We assume the existence of a family of transformers `F` that represent the microarchitectural model, and transformers `G` that represent the architectural model. We also assume that $\mathtt{F}_1 \ldots \mathtt{F}_\mathtt{K}$ and $\mathtt{G}_1 \ldots \mathtt{G}_\mathtt{K}$ are all defined up to some `K`. In the case of OA-64 and Columbia, $\mathtt{K} = 4$, $\mathtt{F}_1 = \mathtt{pipeline}$, $\mathtt{G}_1 = \mathtt{risc}$, etc.

If a model and its specification are the composition of state machine transformers (i.e. $\mathtt{K} > 1$), we can exploit this structure to decompose the proof of $\sqsubseteq$. From `F` and `G`, we construct two indexed families of state machines `M` and `N`:

$$\mathtt{M}_j = (\mathtt{F}_j \circ \ldots \circ \mathtt{F}_1)\ \mathtt{unit}$$
$$\mathtt{N}_j = (\mathtt{G}_j \circ \ldots \circ \mathtt{G}_1)\ \mathtt{unit}$$

Proposition 1.

$$\mathtt{F}_j \sqsubseteq \mathtt{G}_j \wedge \ldots \wedge \mathtt{F}_1 \sqsubseteq \mathtt{G}_1 \Rightarrow \mathtt{M}_j \sqsubseteq \mathtt{N}_j$$

where $\sqsubseteq$ *is extended point-wise over functions:*

$$f \sqsubseteq g \equiv \forall x, y.\ x \sqsubseteq y \Rightarrow f\ x \sqsubseteq g\ y$$

Proof. By induction on j. If $j = 0$ then, by the reflexivity of $\sqsubseteq$, $\mathtt{unit} \sqsubseteq \mathtt{unit}$. If $j > 0$ then, by the inductive hypothesis, $\mathtt{M}_{j-1} \sqsubseteq \mathtt{N}_{j-1}$. By assumption, $\mathtt{F}_j \sqsubseteq \mathtt{G}_j$, meaning that:

$$\mathtt{M}_{j-1} \sqsubseteq \mathtt{N}_{j-1} \Rightarrow \mathtt{M}_j \sqsubseteq \mathtt{N}_j$$

□

4.1 Using Uninterpreted Next-State Relations

When proving that $\sqsubseteq$ holds for two state machines at level $j \in \{1 \ldots \mathtt{K}\}$, we can use uninterpreted functions to represent the composition of levels 1 to $j-1$. One caveat: the implementation transformer, $\mathtt{F}_j$, must be monotonic.

Proposition 2. $\forall j \in \{1 \ldots \mathtt{K}\}$ *where* $\mathtt{F}_j$ *is monotonic with respect to* $\sqsubseteq$,

$$(\mathtt{M}_{j-1} \sqsubseteq \mathtt{N}_{j-1}) \wedge (\mathtt{F}_j\ \mathtt{N}_{j-1} \sqsubseteq \mathtt{G}_j\ \mathtt{N}_{j-1}) \ \Rightarrow\ (\mathtt{F}_j\ \mathtt{M}_{j-1} \sqsubseteq \mathtt{G}_j\ \mathtt{N}_{j-1})$$

Proof. By the monotonicity of $\mathtt{F}_j$, and by the assumption that $\mathtt{M}_{j-1} \sqsubseteq \mathtt{N}_{j-1}$,

$$\mathtt{F}_j\ \mathtt{M}_{j-1} \sqsubseteq \mathtt{F}_j\ \mathtt{N}_{j-1}$$

By assumption,

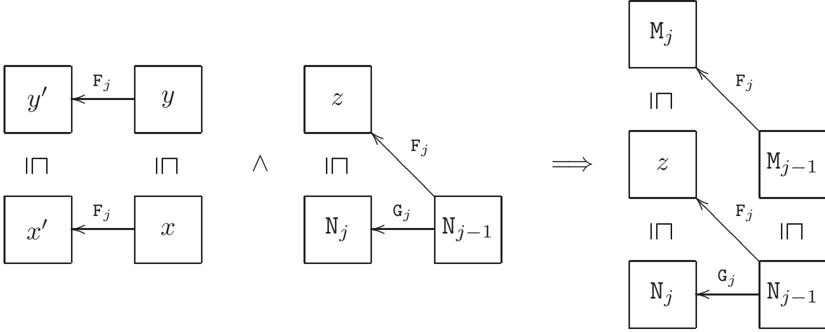
$$\mathtt{F}_j\ \mathtt{N}_{j-1} \sqsubseteq \mathtt{G}_j\ \mathtt{N}_{j-1}$$

Therefore, by the transitivity of $\sqsubseteq$,

$$\mathtt{F}_j\ \mathtt{M}_{j-1} \sqsubseteq \mathtt{G}_j\ \mathtt{N}_{j-1}$$

□

Notice the subtle difference (of one letter) between $\mathtt{F}_j\ \mathtt{N}_{j-1} \sqsubseteq \mathtt{G}_j\ \mathtt{N}_{j-1}$ and $\mathtt{F}_j\ \mathtt{M}_{j-1} \sqsubseteq \mathtt{G}_j\ \mathtt{N}_{j-1}$. By Proposition 1 we need to show that $\mathtt{F}_j\ \mathtt{M}_{j-1} \sqsubseteq \mathtt{G}_j\ \mathtt{N}_{j-1}$ for each j in the set $\{1 \ldots \mathtt{K}\}$. During the proof for a particular j, if you can use the same state machine $\mathtt{N}_{j-1}$ as arguments to both $\mathtt{F}_j$ and $\mathtt{G}_j$, then the proof is easier to construct. Why? Because the underlying next-state relation is shared by the two machines $\mathtt{F}_j\ \mathtt{N}_{j-1}$ and $\mathtt{G}_j\ \mathtt{N}_{j-1}$. In pictures, Proposition 2 states that for all machines x, x', y, y', and z:



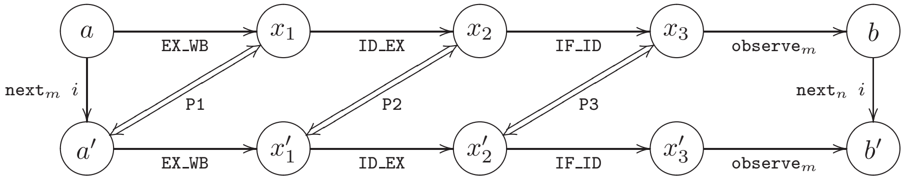
5 Related Work

5.1 Completion Functions

The approach to correspondence checking proposed by Hosabettu et al. [19] replaces Burch & Dill's flushing abstraction [4] with a composition of *completion functions*—each of which contains knowledge about how a pipeline stage behaves. For example, the abstraction function for a DLX pipeline [16] might be

$$\mathtt{observe}_m \circ \mathtt{IF_ID} \circ \mathtt{ID_EX} \circ \mathtt{EX_WB}$$

which results in the commuting diagram:



P1, P2 and P3 assert properties about the effect that pipeline stages have on the implementation state. For example:

$$\mathtt{P1}\ x\ y \equiv \mathtt{observe}_m(\mathtt{next}_m\ i\ x) = \mathtt{observe}_m(\mathtt{EX_WB}\ y)$$

Therefore, the proof is decomposed into the obligations P1, P1 $\Rightarrow$ P2, and P2 $\Rightarrow$ P3. Transformers are quite similar to completion functions: whereas a completion function modifies the behavior of a next-state relation, a transformer modifies behavior *and* changes the structure of the next-state relation's state.

5.2 Compositional Reasoning

Transformer decomposition involves reasoning about the composition of "higher-order" state machines—which, to our knowledge, is new to the literature on processor verification. However, reasoning about the *parallel composition* of state machines is not new [26]. For example, the Pentium III's microarchitecture (called P6) [10] could hypothetically be modeled as three concurrent processes (for some assembly-language program p):

$$\texttt{p6}_p = \texttt{inst_fetch}_p \parallel \texttt{reorder_buffer} \parallel \texttt{res_station}$$

that communicate through shared state. If run in isolation, `res_station`'s inputs would be non-deterministicly assigned. When run in conjunction with other processes, `res_station`'s inputs would be assigned by `inst_fetch` and `reorder_buffer`.

Suppose that `p6`'s specification could also be developed as: the composition of parallel processes:

$$\texttt{x86}_p = \texttt{F} \parallel \texttt{R} \parallel \texttt{E}$$

Showing that $\texttt{inst_fetch} \sqsubseteq \texttt{F}$, $\texttt{reorder_buffer} \sqsubseteq \texttt{R}$, and $\texttt{res_station} \sqsubseteq \texttt{E}$ would be sufficient to show that $\texttt{p6} \sqsubseteq \texttt{x86}$, however its difficult to prove properties about processes in isolation. The assume-guarantee principle [35] provides a more useful form of decomposition:

$$(q_1 \parallel p_2) \sqsubseteq p_1 \ \wedge \ (p_1 \parallel q_2) \sqsubseteq p_2 \Longrightarrow (q_1 \parallel q_2) \sqsubseteq (p_1 \parallel p_2)$$

Assuming that the surrounding context meets its specification, if each process in the model is proved to meet the process' specification then the model meets the specification. When the assume-guarantee principle is applied to $\texttt{p6} \sqsubseteq \texttt{x86}$, the proof is decomposed into cases such as:

$$(\texttt{inst_fetch}_p \parallel \texttt{R} \parallel \texttt{E}) \sqsubseteq \texttt{F}$$

Like transformer decomposition, the assume-guarantee principle is valid if $\sqsubseteq$ denotes trace containment or simulation [17].

5.3 Intermediate Models

In a proof that `cls` contains the traces of $\texttt{cls}'$ we constructed an an intermediate model which simplified the construction of a simulation relation. This is a common approach in microarchitecture verification. For example, when relating microarchitectural models to their specifications, it is often helpful to carry around more state than necessary using *auxiliary models*. Sawada & Hunt [30] built a microarchitectural model that includes a *trace table* to record the behavior of speculative execution, internal exceptions and external interrupts. The model does not use the trace table to make decisions—only to record the past. Sawada & Hunt then established microarchitectural correctness by demonstrating properties about the trace table.

Abstract models are sometimes used to reduce the complexity of microarchitectural models. That is, an abstract model is constructed from the microarchitecture with a simpler and more general next-state relation. The concrete next-state relation is then proved to meet the specification of the abstract one, and it is then shown that the intermediate abstraction implements the specification. Damm & Pnueli [6] and Skakkebaek et al. [33] both use this technique.

5.4 Other Extended Instruction-Sets

The instruction set of the Java virtual machine [25] includes facilities for multi-threaded execution. However, to date, the formalizations of the Java virtual machine have concentrated on type-safety ([28], for example) or have assumed a single-threaded semantics (such as [36]).

Jones et al. [23] and Ho et al. [18] have both based case studies on a dual-issue VLIW microprocessor—called *the protocol processor*—which is found in the Stanford FLASH multiprocessor. The published papers, which focus on validity checking and test-case generation, do not provide many details on the formal specification of the extended instruction-set.

6 Discussion

In this paper we have demonstrated that instruction-set extensions, when modeled as state-machine transformers, can be used to decompose the proof of a correctness relation on extended state machines. In the future we hope to pursue research in the following directions:

Architectural relevance: Beyond parallel and predicated instructions, what can transformers specify? We should explore the boundaries by developing transformers that specify other architectural phenomena such as multimedia extensions, operating system support instructions, speculative loads, rotating register-files, etc.

Microarchitectural relevance: The microarchitectural transformers that we have thus far developed are not very exciting. We should demonstrate that transformers, and the theory that facilitates their decomposition, can model microarchitecturally interesting optimizations—such as branch predication, or trace caches. In cases where optimizations break the “transformer barrier”, we should develop techniques for constructing and verifying intermediate-level models with large `K`-values against microarchitectural models with lower `K`.

Correctness relations: It is tempting to believe that properties (such as reflexivity or transitivity) are maintained when lifted point-wise on functions. Unfortunately this is not so. We should develop techniques to characterize sets of transformers over which the lifted relations have useful properties.
For example, it can probably be proved that any transformer that is completely polymorphic in its state-type is monotonic with respect to simulation. Simulation lifted to the set of transformers with polymorphic state-types is then a reflexive order.

Connections to existing techniques: We should demonstrate that other popular techniques—such as symbolic model checking, symmetry reduction, abstraction, or compositional reasoning—can be used on decomposition obligations. For example, can we use symbolic model checking and uninterpreted functions to build refinement mappings between $\texttt{prd}'$ and `prd`? As in McMillan's correctness proof of a Tomasulo-based model [26], can we use a symmetry argument to show that any number of clusters in $\texttt{cls}'$ correctly implement `cls`? Rather than composition with `fnt`, is parallel composition the right way to specify the front-end of the machine? For example:

$$\texttt{oa64}_p\ m = \texttt{inst_fetch}_p \parallel (\texttt{cls} \circ \texttt{prd} \circ \texttt{risc})\ m$$

Automation and structuring techniques: We should explore the existence of new automation and structuring techniques for transformer verification. For example, for certain fixed correctness relations, such as correspondence, we might develop special techniques that help automate or structure verification.

Liveness properties: We should investigate how the addition of liveness properties would effect transformer decomposition.

Acknowledgments

Mark Aagaard, Todd Austin, Nancy Day, Sava Krstić, Bee Lavender, Tim Leonard, Abdelillah Mokkedem, John O'Leary, Mark Shields, and the anonymous reviewers : thank you for your comments, suggestions and support.

Funding for this research was provided by Intel Corporation, the U.S. National Security Agency, and the U.S. Air Force Material Command (F19628-93-C-0069). John Matthews is supported by a fellowship from the U.S. National Science Foundation.

References

1. Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 2(82):253–284, 1991. 30
2. J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *The 10th ACM Symposium on Principles of Programming Languages*, January 1983. 23
3. David Bistry, Carole Delong, Mickey Gutman, Michael Julier, Michael Keith, Lawrence M. Mennemeir, Millind Mittal, Alex D. Peleg, and Uri Weiser. *The Complete Guide to MMX Technology*. McGraw-Hill, 1997. 24
4. Jerry Burch and David Dill. Automatic verification of pipelined microprocessor control. In *6th International Conference of Computer Aided Verification*, Stanford, California, June 1994. 31
5. Brian Case. IA-64's static approach is controversial. *Microprocessor Report*, 11(16), 1997. 23

6. Werner Damm and Amir Pnueli. Verifying out-of-order executions. In *Conference on Correct Hardware Design and Verification Methods*, Montreal, Canada, 1997. 33
7. Keith Deifendorff. WinChip 4 thumbs nose at ILP. *Microprocessor Report*, 12(16), 1998. 23
8. Carole Delong. The IA-64 architecture at work. *IEEE Computer*, 31(7), 1998. 23, 24
9. Kieth Diefendorff. Microarchitecture in the ditch. *Microprocessor Report*, 12(17), 1998. 23
10. Linley Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2), 1995. 32
11. Linley Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 14(10), 1996. 24
12. Linley Gwennap. Intel, HP make EPIC disclosure. *Microprocessor Report*, 11(14), 1997. 23
13. Linley Gwennap. AltiVec vectorizes PowerPC. *Microprocessor Report*, 12(6), 1998. 24
14. Linley Gwennap. AMD deploys K6-2 with 3DNow. *Microprocessor Report*, 12(7), 1998. 24
15. Linley Gwennap. Intel outlines high-end roadmap. *Microprocessor Report*, 12(14), 1998. 23
16. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1995. 29, 31
17. Thomas A. Henzinger, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. An assume-guarantee rule for checking simulation. In *Formal Methods in Computer-Aided Design*, Palo Alto, California, 1998. 32
18. Richard C. Ho, C. Han Yang, Mark A. Horowitz, and David Dill. Architecture validation for processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995. 33
19. Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In *International Conference on Computer-Aided Verification*, Vancouver, Canada, July 1998. 31
20. Dave Jaggar. *Advanced RISC Machines Architectural Reference Manual*. Prentice Hall, 1997. 23
21. David Johnson. Techniques for mitigating memory latency in the the PA-8500 processor. In *Hot Chips 10*, Palo Alto, August 1998. 24
22. Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991. 23
23. Robert B. Jones, David L. Dill, and Jerry R. Burch. Efficient validity checking for processor verification. In *Proceedings of the 1995 International Conference on Computer-Aided Design*, San Jose, 1995. 33
24. Vinod Kathail, Michael Schlansker, and B. Ramakrishna Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett Packard Laboratories, 1993. 23
25. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997. 33
26. Ken McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *International Conference on Computer-Aided Verification*, Vancouver, Canada, July 1998. 32, 34
27. R. Milner. An algebraic definition of simulation between programs. In *Proceedings of 2nd International Joint Conference on Artificial Intelligence*, The British Computer Society, 1971. 30

28. Zhenyu Qian. A formal specification of a large subset of Java virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java*. Springer-Verlog, 1998. 33
29. B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1), 1993. 23
30. Jun Sawada and Warren Hunt. Processor verification with precise exceptions and speculative execution. In *International Conference on Computer-Aided Verification*, Vancouver, Canada, July 1998. 32
31. Michael Schlansker, B. Ramakrishna Rau, , Scott Mahlke, Vinod Kathail, Richard Johnson, Sdum Anik, and Santosh G. Abraham. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Technical Report HPL-96-120, Hewlett Packard Laboratories, 1996. 23
32. Bruce Shriver and Bennett Smith. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. IEEE Computer Society Press, 1998. 24
33. Jens Skakkebaek, Robert Jones, and David Dill. Formal verification of out-of-order execution using incremental flushing. In *International Conference on Computer-Aided Verification*, Vancouver, Canada, July 1998. 33
34. Peter Song. Demystifying EPIC and IA-64. *Microprocessor Report*, 12(1), 1998. 23, 24
35. E.W. Stark. A proof technique for rely/guarantee properties. In *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, August 1985. 32
36. Karen Stephenson. Towards an algebraic specification of the Java virtual machine. In *Prospects for Hardware Foundations*. Springer-Verlog, 1998. 33
37. Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA, May 1996. 23
38. Miroslav N. Velev and Randal E. Bryant. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In *Formal Methods in Computer-Aided Design*, Palo Alto, California, 1998. 30