Formal verification of explicitly parallel microprocessors[⋆]

Byron Cook, John Launchbury, John Matthews, and Dick Kieburtz
{byron,jl,johnm,dick}@cse.ogi.edu

Version: Fri Mar 5 19:27:54 PST 1999

Abstract. An emerging trend in microprocessor design is to move complexity from a machine's microarchitecture into its instruction-set architecture. This trend will allow compilers to express inter-instruction dependency information that current superscalar out-of-order machines, such as the Pentium III, derive while performing computation. This trend will change the nature of microprocessor verification: The microarchitectural models will become simpler; but their specifications will become more subtle.
This paper explores the implications that this trend will have on microprocessor verification. We develop an explicitly parallel instruction-set architecture motivated by Intel's IA-64 and discuss possibilities for microarchitectural implementations. We then explore correctness criteria for relating microarchitectures to explicitly parallel instruction sets.

1 Introduction

Historically, each generation of microprocessors has been more aggressive than the previous generation in its search and exploitation of instruction-level parallelism [23]. For example, Intel's Pentium III (which is based on the P6 microarchitecture [6, 12]) maintains a graph of 40 instructions, from which it analyzes inter-instruction dependencies and dynamically schedules instructions into execution units.

There is a cost to this sophistication. Complex superscalar out-of-order microarchitectures lead to larger, hotter microprocessors that consume more power [8]. They are difficult to design and debug, and typically have long critical paths, which inhibit faster clock speeds [5]. Some microarchitects feel that the returns are diminishing from their continued investment into the run-time discovery of instruction-level parallelism [25].

A new trend is developing. Intel [13, 14], Hewlett-Packard [13, 19], Compaq [30], Tera [2], Elbrus [9] and others are all extending or designing new instruction-set architectures with constructs for explicit parallelism. These features include predication [1], speculative load instructions [17], and annotations that describe the dependencies between instructions [28].

⋆ This research is supported in part by Intel, the National Science Foundation, the Defense Advanced Research Projects Agency, and Air Force Material Command.

What will these new instruction-sets look like? How will we verify microarchitectures against them? These are the questions that we hope to address. In this paper, we construct a formal semantics for an instruction-set architecture based on publicly available information regarding Intel's new IA-64 [10]. We then develop a clustered microarchitectural design, and discuss its correctness criteria.

2 OA-64: an explicitly parallel instruction set

This section introduces and motivates the emerging style of architecture design through the Oregon Architecture (OA-64) — an explicitly parallel instruction set. OA-64 extends a traditional instruction set in three ways:

Predication allows for the conditional execution of instructions.
Speculative loads are instructions that can be issued before the value they produce is needed without risk of raising an exception.
Parallelism annotations describe the dependencies between instructions.

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

– An OA-64 program is a finite sequence of *packets*, where each packet consists of three instructions. OA-64 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 `check_s` instruction declares that instructions with thread identifiers that are not equal to `0` can be executed in any order with respect to the `check_s` instruction.
– Packets can be annotated with a fence directive `(FENCE)`, which directs the machine to retire all in-flight instructions before executing the following packet.
– Instructions are predicated on boolean-valued registers. For example, the `check_s` instruction will only be executed if the value of `p5` is true in the current register-file state.

2.1 Calculating regions

Thread identifiers and fences are annotations to express concurrency information about instructions. One useful presentation of this concurrency information is a directed graph whose nodes are sets of threads (which are finite instruction sequences) that occur between fence directives. We call each set of threads a *region*. The general idea is that 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.

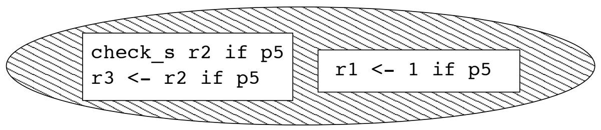
```
101:      check_s r2 if p5 in 0
          r3 ← r2 if p5 in 0
(FENCE)   r1 ← 1 if p5 in 1

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

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 in 3
          pc ← 33 if p3 in 4
(FENCE)   nop
```

Fig. 1. OA-64 implementation of factorial function.

In this section we derive the meaning of the code in Fig. 1 by calculating its regions. We assume that packet 100 issues a fence, and that before entering this code, the machine has loaded a value into register r2 with the speculative load instruction (load_s).

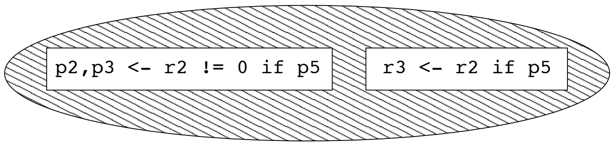
In packet 101, the check_s instruction declares that the machine is about to use the value stored into r2. It is at this point that the machine should raise any exceptions that might have been encountered while speculatively loading data into r2. The first packet also initializes the values of registers r1 and r3. Because r3 depends on the value of r2, the check_s instruction must be executed before writing to r3 — this is expressed by placing the same thread-identifier (0) in 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. Because both packets 100 and 101 issues fence directives, packet 101 forms its own region:



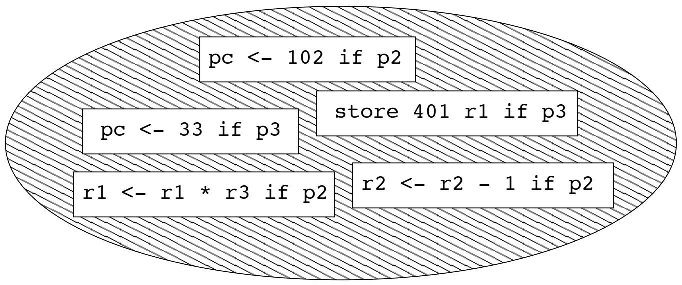
where boxes represent threads. Instructions within a thread must be executed in order. Threads, however, can be executed in any interleaving-order with other threads. Packet 101 forms a region — therefore the machine is required to synchronize the state before executing the next packet.

Because packet 102 is also fenced, it also forms its own region:



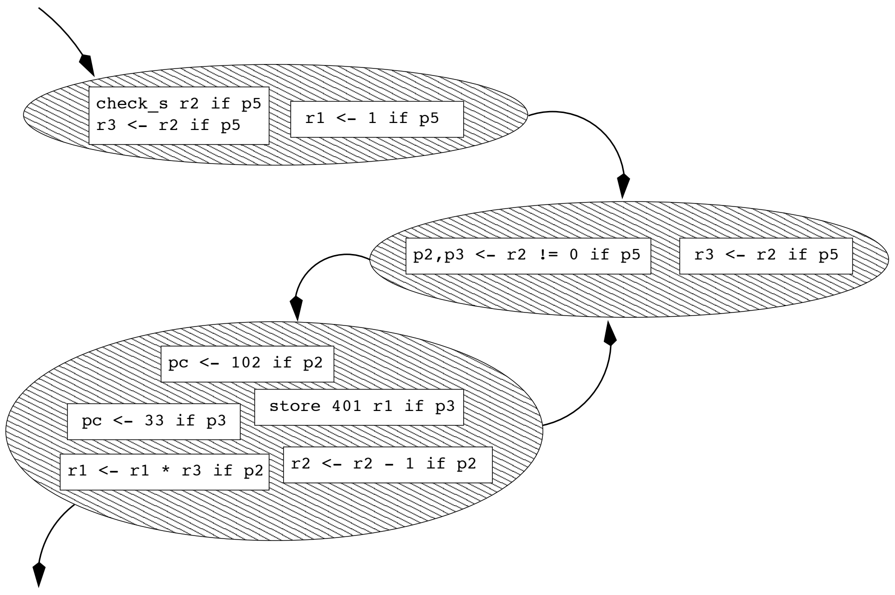
The comparison instruction sets the predicate register `p2` to true if `r2` is not equal to 0. The value of `p3` is set to the negation of `p2`.

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. 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:



At first glance, issuing speculative loads and calculating regions may appear strange. However this is precisely the sort of control calculation an out-of-order machine performs while executing a traditional program [25]. For example

- An out-of-order execution core allows instructions following a memory load to execute before retiring the load. The Pentium III temporarily stores completed successors of a load into a content-addressable array until the load is retired, and flushes the array if the load raises an exception.

  The OA-64 code fragment in Fig. 1 uses a `check_s` instruction that checks to see if the previously issued speculative load succeeded before executing the instructions that depend on it.
- A traditional encoding of the factorial function would use a conditional branch in the place of the predicate calculation. A machine with branch speculation might predict that the branch is not taken and issue the instructions in the third packet before calculating the condition. In this case the branch target buffer is acting as the predicate register file.

  The OA-64 program calculates a predicate, issues instructions from both sides of the would-be branch, and in the end only commits the instructions that satisfy the predicate.
- In a traditional instruction set the encoding of the factorial function would leave much of the instruction-level parallelism implicit. The scheduling logic within an out-of-order machine might analyze the register references and discover that the subtract and multiply instructions are not dependent and can be scheduled out-of-order.

  In OA-64, the compiler (or programmer) declares the dependencies between instructions. If the compiler expresses that the subtract and multiply instructions are not dependent, the machine may retire them out-of-order.

3 Semantics of OA-64

In this section we describe a formalization of OA-64 that facilitates the mathematical verification of microarchitectural implementations. The meaning of OA-64 is defined by a set of restrictions on the source program, an initial state, and a transition relation that describes how instructions effect the state.

3.1 Source code restrictions

The following restrictions are placed on OA-64 programs:

- a multiple packet region must always execute at least one branch instruction;
- a branch instruction can only jump to a packet that immediately follows a packet with a fence directive, or to the first packet in the program;
- a program must be a finite sequence of packets;

3.2 Initial state and transition relation

We view OA-64 as a two-level language — the bottom level, or *base-level*, is a vanilla RISC instruction set with support for speculative loads; the top level, or *concurrency-level*, handles predication, thread identifier annotations and the fence directives. The concurrency-level language is used to express dependencies between base-level instructions.

The semantics of OA-64 highlight this perspective by defining a traditional base-level transition relation and a concurrency-level transition relation. The base-level relation $\triangleright$ is defined over instructions and pairs of base-level architectural states — called *base-states* — which represent the state of the register file and memory (the program counter is modeled as the special register `pc` in the register file). The expression:

$$\Delta, w \triangleright \Gamma$$

asserts that instruction w in state Δ can execute and result in state Γ in $\triangleright$. This relation is simply the familiar instruction-set style of relation used in many papers, i.e. $\Delta, (\mathtt{x} \leftarrow \mathtt{y} + \mathtt{z}) \triangleright \Delta[x \mapsto \Delta(y) + \Delta(z)]$

The concurrency-level transition relation $\blacktriangleright$ is defined in Fig. 2 over pairs of concurrency-level architectural states, called *concurrency-states*, which have the form:

$$(P, \Delta, \Sigma)$$

where P is a finite sequence of packets representing the OA-64 program, and Δ is an base-level state. Σ is the state of the region, which is a finite set of finite instruction sequences. Given an OA-64 program, P, the machine's initial concurrency-state is the triple:

$$(P, \mathbf{init}, \emptyset)$$

where **init** is an initialized base-state, and $\emptyset$ is the empty region.

In the initial concurrency-state, or when the machine has completely executed a region, the concurrency-state of the machine will be in the following form

$$(P, \Delta, \emptyset)$$

In this case, the rule **next** (in Fig. 2) states that the machine should use the value of `pc` in the current base-state (Δ) to fetch the next region. The function **region**, when given an OA-64 program and an index, returns the region pointed to by the index. Also, the base-state is updated by incrementing the program counter.

If the region in the current concurrency-state is not empty, then it will have the form

$$(P, \Delta, \langle \ldots, (w \ \mathtt{if}\ p) : ws, \ldots \rangle)$$

where $(w \ \mathtt{if}\ p)$ is the first instruction of an arbitrarily chosen thread[1]. If, in the base-state Δ, the value of p is false then the instruction w is thrown away (rule

[1] We use : as a constructor of lists. In the expression $x : xs$, x is the first element in the list and xs represents the remaining elements

skip in Fig. 2). Otherwise, if p is true in the base-state, then a new base-state $\varGamma$ is related to $\varDelta$ and w by $\triangleright$ (rule **execute**).

$$\textbf{(next)} \qquad (P, \varDelta, \emptyset) \ \blacktriangleright\ (P, \varDelta[\texttt{pc} \mapsto \texttt{pc} + 1], \textbf{region}(P, \varDelta(\texttt{pc})))$$

$$\textbf{(skip)} \qquad (P, \varDelta, \langle \ldots, (w \texttt{ if } p) : ws, \ldots \rangle) \ \blacktriangleright\ (P, \varDelta, \langle \ldots, ws, \ldots \rangle) \qquad \text{if } \neg\varDelta(p)$$

$$\textbf{(execute)} \qquad \frac{\varDelta, w \ \triangleright\ \varGamma}{(P, \varDelta, \langle \ldots, (w \texttt{ if } p) : ws, \ldots \rangle) \ \blacktriangleright\ (P, \varGamma, \langle \ldots, ws, \ldots \rangle)} \qquad \text{if } \varDelta(p)$$

Fig. 2. Concurrency-level semantics of OA-64

4 Columbia: An OA-64 microarchitecture

The advantage of OA-64 is that the microarchitecture can dedicate more of its resources to computation, and less to scheduling. This section presents an outline of a possible microarchitecture for OA-64.

The picture in Fig. 3 is of Columbia, a clustered OA-64 microarchitecture. The machine's execution core is composed of three independent execution pipelines, or clusters. At each cycle Columbia fetches a packet from the **ICache** and feeds it 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.

Fetched instructions travel from the **ICache** to the **Route** unit, where they are routed to one of three pipelined execution clusters based on their thread-identifier (modulus 3). The execution clusters act as traditional in order pipelined execution cores, except that they share a communal register file (**RF**) and data cache (**MCache**). At each clock cycle the clusters calculate how many instructions they can accept on the next cycle. The minimum of these values is sent to the control logic (because all of the instructions in a packet might be routed to one execution cluster). The control logic is also signaled when all of the clusters are in a flushed state.

The fetch logic uses the register file's program counter value. The **Valid** circuit 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).

Notice that, in contrast to the large amounts of interconnected state found in superscalar out-of-order models, Columbia's state is smaller and mostly local (i.e. local buffers within execution clusters). This is good news for everyone: The reduced state will be simpler for algorithmic formal verification; and the reduced interaction between components will be good for deduction.

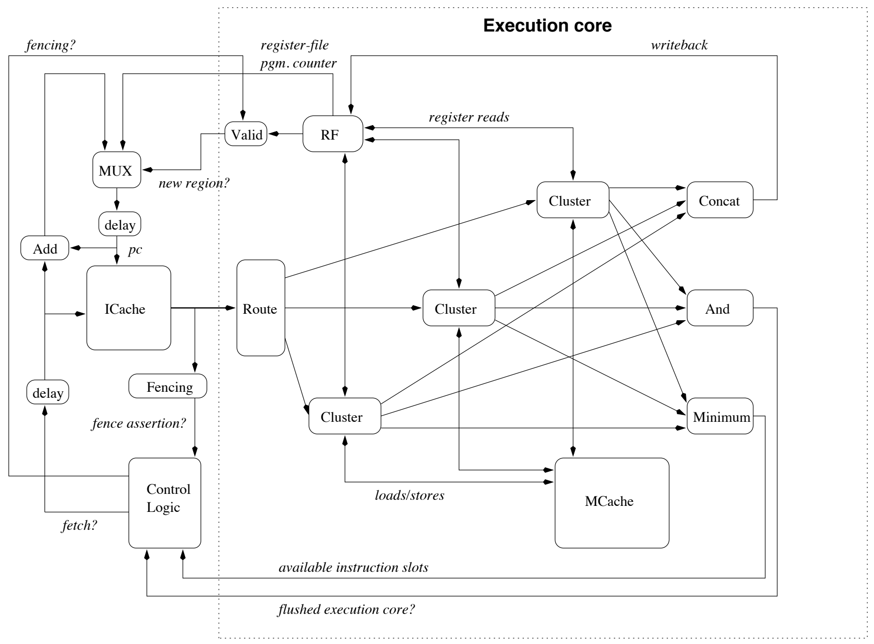


Fig. 3. Columbia microarchitecture — pictured here with three pipeline clusters

5 Verification

Explicitly parallel machines aim to exploit much of the same instruction-level parallelism that superscalar out-of-order machines use — with a twist. They use less hardware, but are more difficult to program. It is therefore natural that the verification of explicitly parallel microarchitectures will be similar to the verification of superscalar out-of-order machines — with a twist. They will be easier to prove correct, but the correctness criteria are more difficult to define.

Assume that, for a given microarchitectural model, ξ_n is a projection representing the machine's region state at time n, and δ_n is the base-state within the microarchitecture. In the case of Columbia, ξ is the contents of the pipelines (and their buffers) and δ equals the contents of the register file and memory cache.

The criteria that we advocate for OA-64 are, for a given program (P), the concurrency-state formed with ξ and δ should infinitely often enter into a reachable concurrency-state defined by the closure of the instruction-set semantics (safety)

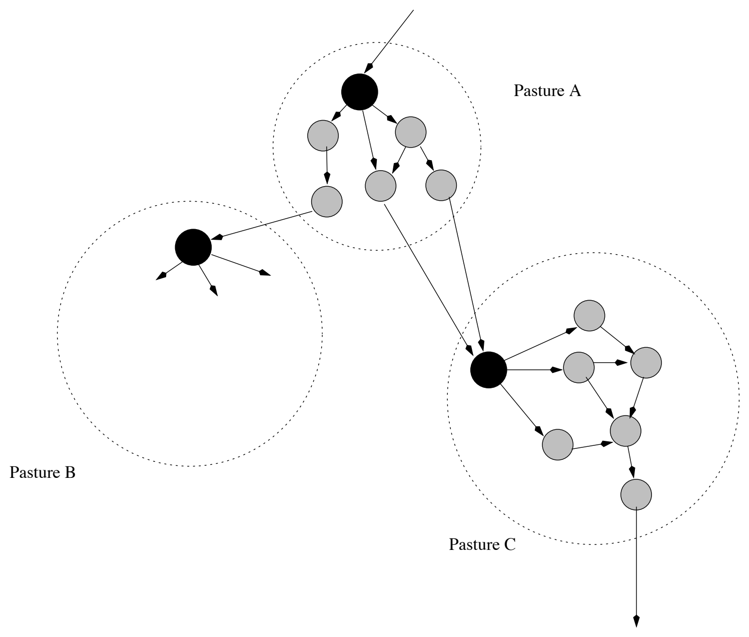
$$\forall n.\exists n'.\ n \leq n' \ \wedge\ (P, \mathbf{zero}, \emptyset) \overset{*}{\blacktriangleright} (P, \delta_{n'}, \xi_{n'})$$

and that the machine infinitely often makes progress in the computation (liveness)

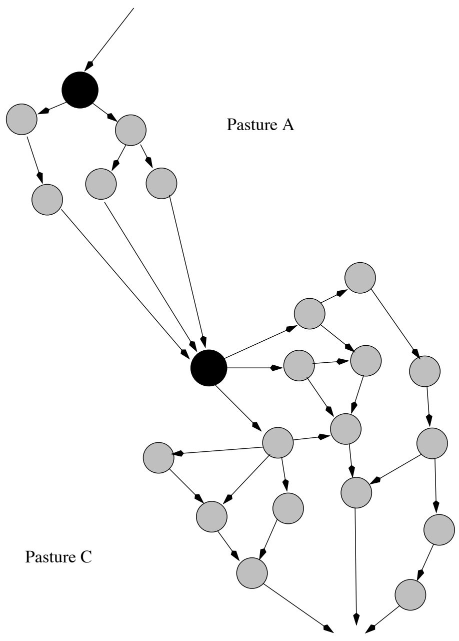
$$\forall n.\ (P, \mathbf{zero}, \emptyset) \overset{*}{\blacktriangleright} (P, \delta_n, \xi_n) \quad \Rightarrow \quad \exists n'.\ n < n' \ \wedge\ (P, \delta_n, \xi_n) \overset{+}{\blacktriangleright} (P, \delta_{n'}, \xi_{n'})$$

The key here is regions, which declare the existence of *synchronization points* — concurrency-states along the path of execution in which threads have access to the results of computation from previously executed threads. In $\blacktriangleright$, every concurrency-states resulting from a **next** transition is a synchronization point. The formulation of OA-64, coupled with the constraints on the input program, guarantee that regions are always finite. Therefore OA-64 guarantees that the transition **next** will be applied infinitely often.

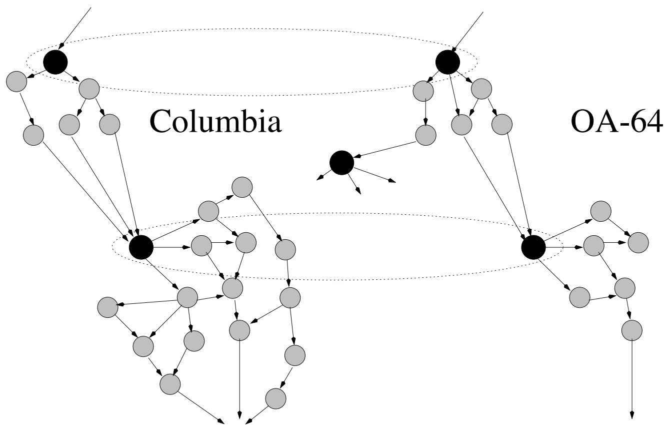
Suppose that, for a given program, the concurrency-state transition graph (based on the region element of the concurrency-state) has the following form



where the black circles are the synchronization points. Also, suppose that the microarchitectural transition graph (based on the value of the microarchitectural thread state ξ) has the form



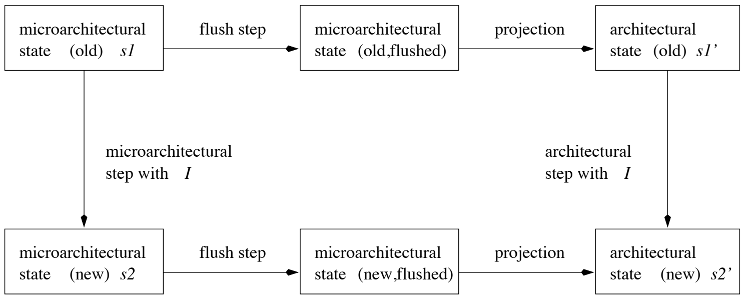
where the black circles represent microarchitectural synchronization points. Between synchronization points the microarchitecture might make more or fewer transitions than the instruction-set architecture. However, when viewing synchronization points, the microarchitecture's transitions are contained by the architecture. The verification problem is then to demonstrate that, when the microarchitecture has reached a synchronization point, the state of the register file and the region that it is executing relates to a reachable concurrency-state in OA-64.



5.1 Adapting pipeline flushing methods

When paired with an inductive proof over the infinite path of regions, the pipeline flushing method [4] for pipeline verification can be adapted to imply the proposed safety property.

In Burch and Dill's formulation, one must prove the commuting square for all possible instructions I:


microarchitectural state (old) s1
flush step
microarchitectural state (old,flushed)
projection
architectural state (old) s1'
microarchitectural step with I
architectural step with I
microarchitectural state (new) s2
flush step
microarchitectural state (new,flushed)
projection
architectural state (new) s2'

In the setting of explicitly parallel architectures, we propose letting I range over regions instead of instructions. That is, assume that the microarchitecture begins to execute a region in synchronization point $s1$, and that $s2$ is the next synchronization point resulting from the execution starting at $s1$. Let $s1'$ be the result of flushing and projecting out the architectural state from $s1$, and $s2'$ be the analogous calculation from $s2$. Does there exist a path in $\blacktriangleright$ from $s1'$ to $s2'$?.

A drawback to this formulation is that I no longer has a clear bound (ie. 16 bit instructions). Instead, I is bounded by the size of regions — which is not satisfactory for model checking. In our verification, we made deductive arguments based on the fact that some finite number of cycles after fetching a packet with a fence declaration, Columbia transitions into a synchronization point. We used a symmetry-reduction styled argument to show that, if the microarchitecture fetched an entire region before executing (given sufficient buffering), then that is the same as concurrently executing and fetching that region. The more abstract transition relation calculated from this symmetry argument was then compared to $\blacktriangleright$. The final step was to show that, when the machine has entered into a synchronization point, that it correctly transitions to the next region. This final step was proved using Symbolic Trajectory Evaluation [15]

A useful characteristic of Columbia-like microarchitectural models is that the number and arrangement of clusters doesn't affect the correctness of a microarchitectural design. This is because the transition relation $\blacktriangleright$ allows for any order of evaluation when many threads are trying to write to a shared location in the object state.

No matter how many clusters the execution core employs, so long as the clusters behave analogously to $\triangleright$, the correctness of the execution core outlined in Fig. 3 can be abstractly characterized by the following assertion (certain preconditions have been omitted):

$$(S, \delta_n, (\mathbf{schedule}(\texttt{fetched}_n, \xi_n))) \overset{*}{\blacktriangleright}_{\{\mathbf{execute},\mathbf{skip}\}} (S, \delta_{n+1}, (\xi_{n+1}))$$

where $\overset{*}{\blacktriangleright}_{\{\mathbf{execute},\mathbf{skip}\}}$ is the closure of the relation $\blacktriangleright$ using only the rules **execute** and **skip**, and **schedule** distributes a packet into a partial region.

Note to reviewers: We're waiving our hands a bit in this section. The statements made in this section are based on a pencil-and-paper proof. We are building a proof in Isabelle which should be done before a camera-ready version of this paper would be due.

6 Related work

The work in this paper is closely aligned in approach with the existing research on the verification of superscalar out-of-order machines [3, 7, 11, 24, 26, 27], all of which use refinement based techniques or flushing (which can be cast as an instance of refinement). In most of these papers, extra information about the dependencies, which OA-64 makes explicit, has been added to the models. For example, Damm and Pnueli construct a non-deterministic data-flow machine that computes the same result as the instruction-set architecture and is refined by a Tomasulo-like transition system. Of course their machines can only execute finite instruction streams that do not contain branches; but their abstract data-flow machine is similar to OA-64.

The instruction set of the Java virtual machine includes facilities for threaded execution. Unfortunatly, the formalizations of the Java virtual machine have, to date, concentrated mainly on type-safety ([22], for example) or have assumed a single-threaded semantics (such as [29]).

Techniques from formal verification have been used to automate the test generation for a dual-issue DLX microprocessor [16] which can be viewed as a simple explicitly parallel machine. The Stanford Validity Checker has been used to show properties of this same processor [18]. However, that paper focuses primarily on the quantifier-free logic of equality with uninterpreted functions and does not go into detail about the properties verified.

7 Future work

The upcoming explicitly parallel instruction-set architectures will take many forms; OA-64 is only one conservative possibility. For example, the real instruction sets might allow sychronization between individual threads; or they might allow branch instructions to take immediate effect on the machine's program counter. Meanwhile, real explicitly parallel microarchitectures will use branch prediction, or even out-of-order clusters to improve performance. The work presented here is conservative in its specification and model. We hope to verify more sophisticated microarchitectures against more realistic instruction sets.

The use of layered transition relations ($\blacktriangleright$ and $\triangleright$) has been invaluable to the understanding and verification of explicitly parallel machines. We hope to generalize this notion, with separate levels for each instruction-set feature: concurrency, predication, speculation, etc. We may find that a particular piece of a microarchitecture implements a single-level of an instruction-set's semantics; which might allow us to treat the other semantic layers much more abstractly — perhaps as uninterpreted functions.

Letting the I range over entire regions in Section 5.1, while theoretically interesting, makes algorithmic verification difficult. We hope to find other finer-grained approaches (perhaps still based on flushing) that imply correctness.

McMillan's use of symmetry [21] might prove to be useful in the setting of multiple symmetric execution clusters. It should be possible that a small set of cluster configurations could represent all possible cluster configurations. McMillan applied this technique to reduce the number of reservation station and execution unit pairs in his model of Tomosulo's algorithm. He represented all configurations with two reservation station / execution unit pairs — one to represent the active pair, and the other to represent all other pairs.

From ▶, it might be interesting to develop a reference model and verify more sophisticated OA-64 models against it using the algebraic approach proposed by Matthews and Launchbury [20]. This will involve developing (perhaps non-finite state) circuits that model the characteristics of the instruction set such as predicated execution, speculative execution, etc, and then using algebraic laws to transform the microarchitectural model into a reference machine.

8 Acknowledgments

For their contributions and comments, we would like to thank Mark Aagaard, Todd Austin, and John O'Leary of Intel Corporation; Tim Leonard and Abdelillah Mokkedem of Compaq/Digital Corporation; Mandayam Srivas of SRI; and Tito Autrey, Nancy Day, Sava Krstić, Jeff Lewis, Thomas Nordin, and Mark Shields of OGI.

References

1. Allen, J., Kennedy, K., Porterfield, C., and Warren, J. Conversion of control dependence to data dependence. In *The 10th ACM Symposium on Principles of Programming Languages* (Jan. 1983).
2. Bokhari, S., and Mavriplis, D. The Tera multithreaded architecture and unstructured meshes. Tech. Rep. NASA/CR-1998-208953, NASA/ICASE, 1998.
3. Burch, J. Techniques for verifying superscalar microprocessors. In *33rd annual Design Automation Conference* (Las Vegas, Nevada, June 1996).
4. Burch, J., and Dill, D. Automatic verification of pipelined microprocessor control. In *6th International Conference of Computer Aided Verification* (Stanford, California, June 1994).
5. Case, B. IA-64's static approach is controversial. *Microprocessor Report 11*, 16 (1997).
6. Cook, B., Launchbury, J., and Matthews, J. Specifying superscalar microprocessors with Hawk. In *Workshop on Formal Techniques for Hardware* (Maarstrand, Sweden, June 1998).

7. Damm, W., and Pnueli, A. Verifying out-of-order executions. In *Conference on Correct Hardware Design and Verification Methods* (Montreal, Canada, 1997).
8. Diefendorff, K. Microarchitecture in the ditch. *Microprocessor Report 12*, 17 (1998).
9. Diefendorff, K. The Russians are coming. *Microprocessor Report 13*, 2 (1999).
10. Dulong, C. The IA-64 architecture at work. *IEEE Computer 31*, 7 (1998).
11. Fox, A. C., and Harman, N. A. Algebraic models of superscalar microprocessor implementations: A case study. In *Prospects for Hardware Foundations*. Springer-Verlog, 1998.
12. Gwennap, L. Intel's P6 uses decoupled superscalar design. *Microprocessor Report 9*, 2 (1995).
13. Gwennap, L. Intel, HP make EPIC disclosure. *Microprocessor Report 11*, 14 (1997).
14. Gwennap, L. Intel outlines high-end roadmap. *Microprocessor Report 12*, 14 (1998).
15. Hazelhurst, S., and Seger, C.-J. H. Symbolic trajectory evaluation. In *Formal Hardware Verification*. Springer-Verlog, 1997.
16. Ho, R. C., Yang, C. H., Horowitz, M. A., and Dill, D. Architecture validation for processors.
17. Johnson, D. Techniques for mitigating memory latency in the the PA-8500 processor. In *Hot Chips 10* (Palo Alto, Aug. 1998).
18. Jones, R. B., Dill, D. L., and Burch, J. R. Efficient validity checking for processor verification. In *Proceedings of the 1995 International Conference on Computer-Aided Design* (San Jose, 1995).
19. Kathail, V., Schlansker, M., and Rau, B. R. HPL PlayDoh architecture specification: Version 1.0. Tech. Rep. HPL-93-80, Hewlett Packard Laboratories, 1993.
20. Matthews, J., and Launchbury, J. Elementary microarchitecture algebra. In *International Conference on Computer-Aided Verification* (Trento, Italy, July 1999).
21. McMillan, K. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *International Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998).
22. Qian, Z. A formal specification of a large subset of Java(tm) virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java(tm)*. Springer-Verlog, 1998.
23. Rau, B. R., and Fisher, J. A. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing 7*, 1 (1993).
24. Sawada, J., and Hunt, W. Processor verification with precise exceptions and speculative execution. In *International Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998).
25. Schlansker, M., Rau, B. R., , Mahlke, S., Kathail, V., Johnson, R., Anik, S., and Abraham, S. G. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Tech. Rep. HPL-96-120, Hewlett Packard Laboratories, 1996.
26. Shen, X., and Arvind. Design and verification of speculative processors. In *Workshop on Formal Techniques for Hardware* (Maarstrand, Sweden, June 1998).
27. Skakkebaek, J., Jones, R., and Dill, D. Formal verification of out-of-order execution using incremental flushing. In *International Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998).

28. Song, P. Demystifying EPIC and IA-64. *Microprocessor Report 12*, 1 (1998).
29. Stephenson, K. Towards an algebraic specification of the Java virtual machine. In *Prospects for Hardware Foundations*. Springer-Verlog, 1998.
30. Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., and Stamm, R. L. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture* (Philadelphia, PA, May 1996).