Detecting Pipeline Structural Hazards Quickly

Todd A. Proebsting*
University of Arizona

Christopher W. Fraser†
AT&T Bell Laboratories

1 Introduction

Most modern CPUs pipeline instructions. When two instructions need the same machine resource—like a bus, register or functional unit—at the same time, they suffer a *structural hazard*, which stalls the pipeline or corrupts a result. Compilers order or *schedule* instructions to cut structural hazards. A fundamental step in scheduling is detecting if a series of instructions suffers a structural hazard.

This paper describes a method for detecting structural hazards 5–80 times faster than its predecessors, which generally have simulated the pipeline at compile time. It accepts a compact specification of the pipeline and creates a finite-state automaton that can detect structural hazards in one table lookup per instruction.

The automaton maintains an integer *state* that encodes all potential structural hazards for all instructions in the pipe. It accepts an instruction type and a state and either reports a hazard or produces the state that folds in the new instruction and advances the pipeline by one cycle. The automaton can be implemented with a two-dimensional array.

An implementation of the method below generates practical automata quickly. For example, a 33mhz MIPS R3000 generates a 6175-state automaton for the MIPS R3000/R3010[8] in five seconds. This architecture has only 14 distinct instruction classes with respect to creating structural hazards, so the automaton's table takes only 14×6175 two-byte entries. Contrast this with the theoretical upper bound of $2^{22 \times 37}$ states for the MIPS R3000/R3010.

Such automata will speed up current instruction scheduling heuristics and allow compilers to try more schedules. Profilers that count cycles—like pixie[13] and qp[2]—could use it too.

We use an algorithm described in 1975[5], but the prior literature describes neither implementations nor measurements, perhaps because the defining paper attacked a variant problem that needed much larger 3D tables. We show that smaller 2D tables suffice for typical schedulers, we prove that the automata are minimal, and we describe an implementation and experiments.

2 Background

Some schedulers detect structural hazards by recording the instructions in the pipe and when each was issued[9]. When the next instruction is proposed, the scheduler compares each instruction in the pipe with the proposed instruction and objects if there is a hazard. The comparison code is machine-specific. If the architecture is complex, so is the code.

*Address: Todd A. Proebsting, Department of Computer Science, University of Arizona, Tucson, AZ 85721. Internet: todd@cs.arizona.edu

†Address: Christopher W. Fraser, AT&T Bell Laboratories, 600 Mountain Avenue 2C-464, Murray Hill, NJ 07974-0636. Internet: cwf@research.att.com

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

POPL 94- 1/94, Portland Oregon,USA

© 1994 ACM 0-89791-636-0/94/001..$3.50

Instruction	Resources needed						
	cycle 0	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
`mov.s`	EX						
`add.s`	U	S+A	A+R	R+S			
`mul.s`	U	E+M	M	M	N	N+A	R

Figure 1: R4000 Subset

Resource Vector	Resources needed 0	1	2	3	4	5	6
Empty state	∅	∅	∅	∅	∅	∅	∅
`add.s`	U	S+A	A+R	R+S	∅	∅	∅
Combined vectors	U	S+A	A+R	R+S	∅	∅	∅
Shifted state	S+A	A+R	R+S	∅	∅	∅	∅

Figure 2: Issuing `add.s` into an empty pipe.

Resource Vector	Resources needed 0	1	2	3	4	5	6
Previous state	S+A	A+R	R+S	∅	∅	∅	∅
`add.s`	U	S+A	A+R	R+S	∅	∅	∅
Combined vectors	S+A+U	**hazard** on A	**hazard** on R	R+S	∅	∅	∅

Figure 3: Issuing subsequent `add.s` into pipe.

Other structural-hazard detectors use *reservation tables* [5] or *resource vectors* [3, 4] instead. The vector is indexed by a cycle number, and each element records the resources needed during that cycle. There is one resource vector for each distinct class of instructions and another that composes the vectors of all instructions already in the pipe. To schedule an instruction, the scheduler compares its resource vector with the composite vector for the instructions already in the pipe. If both vectors show a need for the same resource at the same time, a structural hazard is reported. Otherwise, the resources from the instruction's vector are incorporated into the composite. Then the composite vector is shifted one cycle forward to simulate issuing the new instruction.

Figure 1 describes three instructions from the MIPS R4000 floating point unit (FPU). For example, `add.s` requires exclusive access to the `U` (for "unpack") stage in cycle 0, the `S` (for "shift") and `A` (for "adder") stages in cycle 1, `A` and `R` (for "round") in cycle 2, and `R` and `S` in cycle 3. Tables 8-7 and 8-8 in Reference [8] elaborate.

To illustrate resource vectors, we issue two `add.s` instructions into an empty pipe. The first causes no structural hazard. Combining the initial empty vector with `add.s`'s and shifting it one cycle forward yields the next state's resource vector. Figure 2 demonstrates this combination. Later instructions must respect these reservations.

The second `add.s` (Figure 3) causes a structural hazard when issued in this state. The first `add.s` needs `A` in its third cycle, and after the first `add.s` issues, this reservation appears in the composite vector's second cycle. The second `add.s` needs `A` (and `S`) in *its* second cycle, which causes a hazard. There's another hazard on `R` one cycle later.

3 The "DSTP" Automaton

In 1975, Davidson, Shar, Thomas, and Patel (here abbreviated "DSTP") [5] proposed to use resource vectors to compute an automaton. The automaton would accept integers representing an instruction class i, a cycle count c, and a pipeline state s. If the pipeline is in state s, and if c cycles later, i could be issued without structural hazard, the automaton would produce the integer that encodes the new pipeline state; otherwise, it would report a hazard. The automaton could be represented as a single 3D table.

The DSTP automaton has been underused. The literature includes few citations[12, 1, 7, 6, 10]—mostly surveys—and describes no implementations or measurements.

One explanation is table size. For example, we implemented the method and found that it generates 6175 states for the MIPS R3000. This machine has 14 instruction classes and instructions that take up to 37 cycles, so the 3D table would have $6175 \times 14 \times 37$ or over 3.1 million two-byte entries. The table's not sparse, so sparse matrix encodings wouldn't help.

DSTP's application needed the whole table, but many schedulers do not. (DSTP's application required the 3D table in order to find the *minimum average latency* of a given sequence of instructions and the cycles that contribute to that latency—a special-purpose scheduling objective.) A typical scheduler might run down a prioritized list of ready instructions and schedule the first one for which the automaton reports no hazard for the current pipeline state. If all ready instructions suffer a hazard, then the scheduler might issue a nop. The scheduler can thus get by with only the 2D slice of the 3D table for which c is 1. This observation drops the R3000 table to 6175×14 or 86,450 two-byte entries. The simplified DSTP automaton is practical for current pipelined microprocessors.

DSTP's table constructor, adapted to 2D tables, works as follows. It represents each state S internally as a 2D table of Boolean values. $S[I,t]$ is 1 if and only if instruction I suffers a structural hazard if issued t cycles after the machine enters state S.

States are computed using *collision matrices*, which record when instructions collide. The collision matrices are computed into a 3D table, M, of Boolean values. $M[I_A, I_B, t]$ is 1 if and only if issuing I_B t cycles after issuing I_A causes a structural hazard. The slice $M[I_A]$ is I_A's collision matrix. Collision matrices are computed using resource vectors:

$$M[I_A, I_B, t] = \begin{cases} 1 & \text{if } \exists k \text{ such that} \\ & \quad I_A[k+t] \cap I_B[k] \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

where $I[t]$ denotes the set of resources that instruction I uses at time t. The sample machine above yields these collision matrices:

I_A	I_B	t					
		1	2	3	4	5	6
add.s	add.s	1	1	0	0	0	0
	mov.s	0	0	0	0	0	0
	mul.s	0	0	0	0	0	0
mov.s	add.s	0	0	0	0	0	0
	mov.s	0	0	0	0	0	0
	mul.s	0	0	0	0	0	0
mul.s	add.s	0	0	1	1	0	0
	mov.s	0	0	0	0	0	0
	mul.s	1	1	0	0	0	0

For example, the leftmost 1 in the top row indicates that one add.s instruction must not follow another by exactly one cycle. The adjacent 1 indicates that the two add.s's also collide if separated by exactly two cycles. Note that mov.s collides with nothing.

If a machine is in state S, and $S[I,1]$ is 0, then instruction I may be issued immediately, with a transition to state S'. To compute the matrix for S', S is shifted one cycle forward (to reflect the passage of the issuing cycle) and combined with I's collision matrix, $M[I]$ (to reflect the subsequent potential hazards introduced by I):

$$S'[J,t] = S[J,t+1] \vee M[I,J,t]$$

(The original DSTP algorithm allowed an instruction to be issued after an arbitrary number of cycles, k. Under this model, for every k such that $S[I,k] = 0$, substitute $t+k$ for $t+1$ in

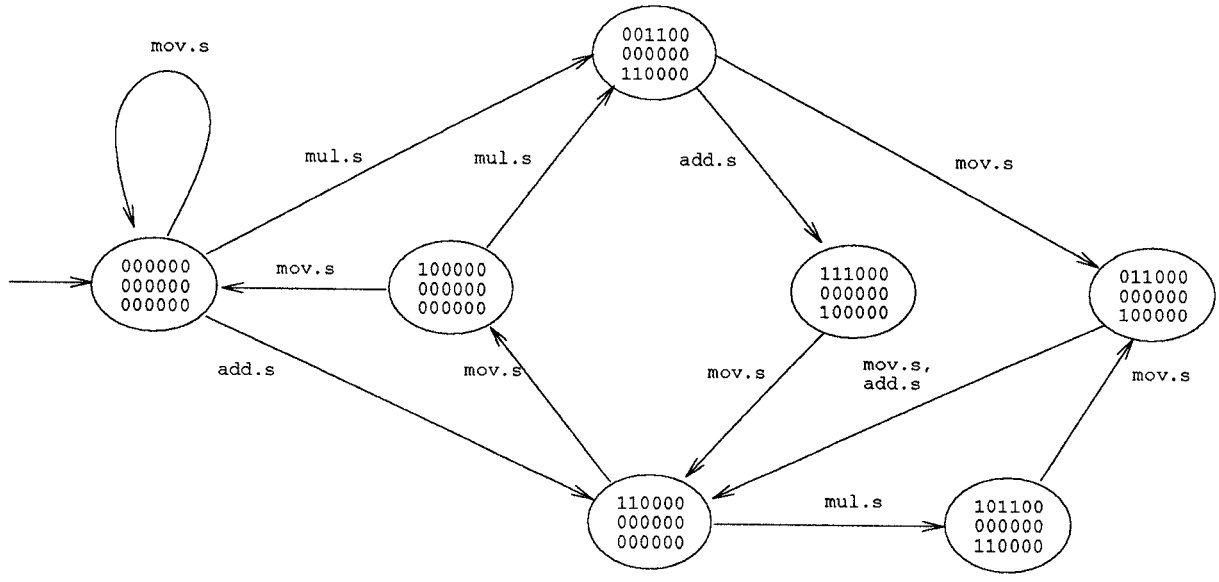

Figure 4: Finite Automaton

the equation and create S'_k. Transitions for this automaton were labelled with both I and k.)

The automaton is built as follows. Beginning with the empty start state (a matrix of all 0's), all instructions are issued, producing new states. For each state, S, an instruction I is issued if $S[I,1] = 0$. The combining rule above forms a target state, S'. If S' has not been created before, it is added to the automaton, as is a transition from S to S' on I. Otherwise, the previously created identical matrix (state) is the target of the transition. This process terminates when all possible instructions have been issued from all states. A state lacks a transition on a particular instruction (because $S[I,1] = 1$) when a structural hazard exists.

This algorithm yields the automaton in Figure 4 for the sample machine. The empty start state is on the left. The transition from that state on `add.s`, for example, shifts the start state's bits one position left and then ors in `add.s`'s collision matrix. Similarly, the transition from the target state on `mul.s` shifts that state's bits one position left and then ors in `mul.s`'s collision matrix.

The algorithm above creates a minimal finite state automaton for machines with a nop, which is an instruction that participates in no structural hazards. To prove this, it is sufficient to show that some sequence of input instructions distinguishes every pair of different states. A nop instruction has a collision matrix of all 0's (that is, $M[\text{nop}, I, t] = 0, \forall I, t)$ and all other instructions' collision matrices are all 0 with respect to issuing a nop (that is, $M[I, \text{nop}, t] = 0, \forall I, t)$. The DSTP algorithm guarantees that each state represents a unique matrix of 1's and 0's. Assume, without loss of generality, that S_x and S_y differ because $S_x[I,t] = 1$ and $S_y[I,t] = 0$ for some I and t. If $t-1$ nop's are issued to each state, the new states, S'_x and S'_y, will have the properties that $S'_x[I,1] = 1$ and $S'_y[I,1] = 0$ because the combining rule above simply shifts each state by one cycle and combines it with the nop's empty collision matrix. $S'_x[I,1] = 1$, so S'_x has no transition on I. S'_y *does* have a transition on I, so $t-1$ nops followed by instruction I distinguishes S_x and S_y. Thus any two distinct states created by the DSTP algorithm are distinguishable, and the automaton must be minimal.

An alternative automata generator has been described recently[11]. It generates minimal automata with heurisitics and a postpass minimizer. Our adapted DSTP algorithm generates minimal automata directly and appears to do so significantly faster than this alternative.

4 Fewer Instruction Classes

The size of the automaton's transition table is reduced primarily by reducing the number of states, but reducing the number of instruction classes helps too. Many instructions share resource vectors. For example, addition instructions generally have the same resource vectors as subtraction instructions. Combining like instructions into classes reduces the over one hundred R3000/R3010 instructions modelled to 20 classes.

We also combine instruction classes that are identical with respect to the generated collision matrices. Two instructions, A and B, are combined if and only if

$$M[A,I,t] = M[B,I,t] \text{ and}$$
$$M[I,A,t] = M[I,B,t] \; \forall I,t.$$

Instructions with distinct resource vectors can generate the same collision matrices, because the resource vector identifies the resources that cause the conflict, but the collision matrix records only that there is a conflict. This optimization reduces the 20 R3000/R3010 instruction classes above to 14.

5 An Implementation

An automaton compiler has been implemented as part of a larger system, RPSS (*Retargetable Pipeline Scheduling Substrate*), which is under development. RPSS accepts a compact, formal specification of a pipelined machine and emits the machine-specific part of an instruction scheduler. It produces routines for detecting control and data hazards as well as an automaton to detect structural hazards.

RPSS attacks more than just structural hazards, so its specifications include material beyond the scope of this paper. Eliminating this extra material leaves just a list of resource vectors, one per instruction class. Blanks separate vector elements, and +'s separate resources used during a single cycle. The suffix ^n flags stages that are repeated for n consecutive cycles. Multiple instructions may share the same description. For example,

```
U S+A A+R R+S.                          add.s
U A R D^27 D+A D+R D+A D+R A R.          div.d
```

describe the R4000 instruction classes that `add.s` and `div.d` exemplify. Text accompanying Figure 1 in Section 2 described some of the resources named above, and Tables 8-7 and 8-8 in Reference [8] describe them all.

Appendices A and B contain specifications for the R4000 FPU and the R3000/R3010; they present only the material relevant to structural hazards. RPSS cannot model variable-length instructions (e.g., the R4000's `sqrt.d`), so the specification must choose a single pipeline description—typically minimum, maximum or average instruction length—and automaton clients must accept approximate answers for those few instructions whose length cannot be predicted. The R4000 spec in Appendix B uses the maximum length `sqrt.d` to maximize generated states.

To create a structural hazard detection automaton, a 500-line Icon program preprocesses the specification into initialized C data structures describing the pipeline. Bit vectors encode the resources used in each cycle. One-dimensional arrays of these bit vectors encode resource vectors.

The automaton compiler results from linking this data module with 300 lines of C that construct an automaton. A 33mhz R3000 constructed automata for the R3000/R3010 and for the R4000 FPU in under five seconds each. The table below describes the automata sizes. The theoretical maxima are huge. For a machine with r resources, there are 2^r possible values for each element of each resource vector; if its longest pipeline has n stages, then there could be as many as $2^{r \times n}$ possible resource vectors or states.

R4000 FPU		R3000/R3010		variant
classes	states	classes	states	
15	$2^{9\times112}$	20	$2^{22\times37}$	theoretical maximum
15	2665	14	6175	actual

```
int tick(unsigned *new, unsigned *old, unsigned *instruction) {
  int i;

  new[MAXSTAGES-2] = instruction[MAXSTAGES-1];  /* optimized first iteration */
  for (i = MAXSTAGES - 2; i > 0; i--) {
    unsigned int a = old[i], b = instruction[i];
    if (a & b)          /* if overlap ... */
      return 0;         /* ... then report hazard */
    new[i-1] = a | b;  /* otherwise combine vectors and shift */
  }
  if (old[0] & instruction[0])  /* optimized last iteration */
    return 0;
  return 1;
}
```

Figure 5: Optimized Code to Interpret Resource Vectors

The automata are much faster than their predecessors. A typical implementation of fixed-length resource vectors accepts three resource vectors: one for the instruction, another for the current composite, and an empty one for the new composite. An optimized implementation is given in Figure 5. For the MIPS R3000/R3010 pipeline, which requires 37-stage vectors, this code takes 329 cycles to detect a structural hazard on a MIPS R3000, assuming no cache misses. A more sophisticated implementation might implement variable-length resource vectors, but even it can't average better than a fixed-length implementation of 5-stage vectors, which takes 50 cycles.

Analogous code using our automaton replaces the resource vectors with integers and the function with an expression:

```
newstate = dfa[oldstate][instruction];
```

It takes 9 cycles, and schedulers that test several instructions before picking the best can amortize the cost of one subscript calculation with

```
/* done once */
temp = dfa[oldstate];
...
/* done repeatedly */
newstate = temp[instruction];
```

and average as few as 4 cycles per test.

References

[1] Jean-Loup Baer. *Computer Systems Architecture.* Computer Science Press, 1980.

[2] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[3] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 26(6):229–240, June 1991.

[4] David Gordon Bradlee. *Retargetable Instruction Scheduling for Pipeline Processors.* PhD thesis, University of Washington, 1991. Technical report 91-08-07, Department of Computer Science and Engineering.

[5] Edward S. Davidson, Leonard E. Shar, A. Thampy Thomas, and Janak H. Patel. Effective control for pipelined computers. In *Spring COMPCON75 Digest of Pa-*

pers, pages 181–184. IEEE Computer Society, February 1975.

[6] Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985.

[7] Kai Hwang, Shun-Piao Su, and Lionel M. Ni. Vector computer architecture and processing techniques. In *Advances in Computers*, volume 20, pages 115–197. 1981.

[8] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

[9] James Larus. Assemblers, linkers, and spim. In David Patterson and John Hennessy, *Computer Organization & Design: The Hardware/Software Interface,* Morgan Kaufman, 1993.

[10] H. F. Li and R. Jayakumar. Systolic structures: A notion and characterization. *Journal of Parallel and Distributed Computing*, pages 373–397, September 1986.

[11] Thomas Müller. Employing finite automata for resource scheduling. In *The 26th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1993. To appear.

[12] C. V. Ramamoorthy and H. F. Li. Pipeline architecture. *ACM Computing Surveys*, 9(1):61–102, March 1977.

[13] MIPS Computer Systems. *RISCompiler Language Programmers Guide.* MIPS Computer Systems, 1988.

A R3000/R3010 Description

```
pc+epc+ir RD ALU DIV^15 ALU^3 MEM WB FWB+f_d. div.d
pc+epc+ir RD ALU DIV^8 ALU^3 MEM WB FWB+f_d.  div.s
pc+epc+ir RD ALU MEM WB FWB+f_d.              abs.d
pc+epc+ir RD ALU MUL^2 ALU MEM WB FWB+f_d.    mul.s
pc+epc+ir RD ALU MUL^3 ALU MEM WB FWB+f_d.    mul.d
pc+epc+ir RD ALU^2 MEM WB FWB+f_d.            add.d
pc+epc+ir RD ALU^3 MEM WB FWB+f_d.            cvt.d.w
pc+epc+ir RD C+ALU MEM WB FWB.                c.eq.d
pc+epc+ir rd+pc+epc alu mem r_d+wb.   jal
pc+epc+ir rd+pc+epc alu mem wb.       j
pc+epc+ir rd+pc+epc alu r_31+mem wb.  bgtzal
pc+epc+ir rd alu C+S+MEM WB FWB.      ctc1
pc+epc+ir rd alu MEM WB FWB+f_d.      lwc1
pc+epc+ir rd alu lo+mem wb.           mtlo
pc+epc+ir rd alu mem+hi wb.           mthi
pc+epc+ir rd alu mem r_d+wb.          add
pc+epc+ir rd alu mem wb.              sb
pc+epc+ir rd alu r_t+mem wb.          lwl
pc+epc+ir rd lo+alu+hi lo+hi^11.      mult
pc+epc+ir rd lo+alu+hi lo+hi^34.      div
```

B R4000 FPU Description

```
EX.                                   mov.s
U S+A A+R R+S.                        add.s
U A R.                                c.cond.s
U S.                                  neg.s
U A R S A R.                          cvt.s.w
U S A R S.                            cvt.d.w
U A R S S A R.                        cvt.s.l
U A R S.                              cvt.d.l
U S A R.                              cvt.s.d
U E+M M M N N+A R.                    mul.s
U E+M M M M N N+A R.                  mul.d
U S+A S+R S D^13 D+A D+R D+A D+R A R. div.s
U A R D^27 D+A D+R D+A D+R A R.       div.d
U E A+R^50 A R.                       sqrt.s
U E A+R^108 A R.                      sqrt.d
```