The Performance Impact of Incomplete Bypassing in Processor Pipelines

Pritpal S. Ahuja Douglas W. Clark Anne Rogers

Department of Computer Science
Princeton University
35 Olden Street
Princeton, New Jersey 08544-2087
{psa, doug, amr}@cs.princeton.edu

Abstract

Pipelined processors employ hardware bypassing to eliminate certain pipeline hazards. Bypassing is logically simple but can be costly, especially in wide issue and deeply pipelined machines. In this paper bypassing is studied in detail, with an emphasis on designs in which the bypassing network is not complete. Cycle-level simulations of a model of integer and floating-point pipelines running some of the SPEC92 benchmarks show that at least half of the instructions executed used a bypassed register result from a previous instruction. Missing bypasses induce interlock stalls. The paper reports measurements of the performance inpact of a number of pipeline configurations with incomplete bypassing networks. This impact ranges from a slowdown of just a few percent for a configuration with one late bypass missing to a slowdown of almost a factor of two for the integer pipe with no bypassing at all. Two types of code alterations reduce the new interlock stalls. A simple code transformation, the interchange of operands in instructions that perform commutative operations, cuts the performance loss from interlock stalls in certain configurations between about 20 and 50 percent. The second transformation is to reschedule code within basic blocks to avoid any missing bypasses. In five individual experiments with a small number of configurations and two benchmarks, this rescheduling saved 25 to 50 percent of the interlock stalls. In certain configurations both transformations can be applied.

1 Introduction

Bypassing, or forwarding, is a simple, powerful, widely-used method for eliminating certain data hazards in pipelined processors [13]. Bypassing adds hardware datapaths and control so that an instruction's result is available as an input to subsequent instructions before it has been written to an architectural register. An old idea, bypassing was first described in Bloch's 1959 paper on the IBM Stretch [4]. Modern pipelined microprocessors use bypassing heavily in order to avoid hazard-induced stalls.

Bypassing hardware is not logically complicated, but can be costly, depending on the details of the implementation technology. Bypassing can take significant amounts of wiring area on the chip, possibly widening the pitch of the execution-unit datapaths; logic paths including bypassed data can put pressure on the cycle time, especially for single-cycle paths around an integer ALU; the required wide multiplexors or multi-driver buses consume more power for each input; and the control logic for bypassing, while not complicated, can also affect cycle time. These pressures increase when there are paths from many pipe stages to the inputs of multiple functional units. The use of deeper pipelines means more source stages for bypassed results, and the use of parallel functional units in superscalar and VLIW designs means both more sources and more destinations for bypass paths. If i instructions are simultaneously issued to identical 2-input functional units with s pipestages after the earliest answer-producing stage, then a fully bypassed design would require $2i^2s$ separate logical bypass connections. One current machine, the Digital Alpha 21064, has 45 separate bypass paths [18].

Our goals in this paper are to measure bypass use in a fully bypassed model of integer and floating-point pipelines, to investigate the performance impact of configurations in which the bypassing network is incomplete, and to evaluate some simple code transformations that can recover stalled cycles lost due to missing bypasses. Our hypothesis is that good pipeline performance does not require a complete set of bypass paths, and that code can be altered to accommodate the absence of certain paths. This hypothesis is in line with two general trends of the significant architectural and organizational advances in uniprocessors of the last fifteen years: it takes advantage of the fact that real programs do not use the full generality available to them in the hardware, and it relies on the compiler to make simpler hardware perform well.

Removing bypasses cannot increase Instruction-Level Parallelism, of course—it does just the opposite. What we are trying to do here is to *maintain* ILP while reducing hardware. This might be called an increase in ILP relative to some hardware cost.

The bypassing technique, though widely used, has been little studied. The work closest to our own is that of Cohn, *et al.* [7], which included a study of a single incompletely-bypassed configuration of the iWarp VLIW processor running floating-point kernels. They found that appropriate code scheduling could bring the performance of their model almost up to that of a fully-bypassed processor. Huang and Despain [14] propose a system for pipeline dependency classification, but touch only briefly on bypasses. Capitanio *et al.* [6] look at the related idea of partitioned register files in the context of VLIW machines. They consider "limited connectivity" among register files assigned to particular VLIW functional units, arguing that having fewer ports on a register file can lead to a faster cycle time. In their model, a special move instruction is needed to transfer data from one VLIW "cluster" to another; they do not look at pipeline bypassing, nor at data communication within one cluster. Abnous and Bagherzadeh [2] analyze partial bypassing between VLIW functional units in their 4-integer-unit VIPER processor [11]. They conclude that full inter-unit bypassing is not the highest-performance interconnection, due to its modelled cost in extra cycle time.

The next section of this paper presents our processor model and describes the simulation methods and the benchmarks. Then Section 3 reports baseline measurements of bypassing in a fully-bypassed version of the model. In Section 4 we investigate the performance of various model configurations with incomplete bypass networks, in which a missing but needed bypass introduces new stalled cycles. In Section 5 we look at two code transformations that recover many of these stalled cycles. Section 6 summarizes our results and suggests avenues for further work in this area.

2 Experimental Framework

A simple example will illustrate bypassing and help us motivate the work described here. Consider a simple single-issue RISC integer pipeline [15, 19] with five stages: IF (Instruction Fetch), ID (Instruction Decode and register read), EX (EXecute), MEM (MEMory access), and WB (Write Back result). This pipeline leads to a read-after-write hazard in this instruction sequence

```
sub r1, r2, r3          (r1 = r2 - r3)
add r4, r1, r5          (r4 = r1 + r5)
```

because the `add` instruction needs to use the result of the `sub` instruction. Since the WB stage is the last one in the pipeline, the `add` instruction would appear to need three stall cycles due to the hazard:

```
sub:  IF   RD   EX   MEM   WB
add:       IF stall stall stall RD EX MEM WB
```

Stalls like this would be devastating to pipeline performance, but an easy remedy is at hand: a hardware path can be added from the output of the ALU back to its input multiplexors (including one pipeline latch). After all, the result of the `sub` instruction is certainly valid in time to be an input to the ALU stage for the add instruction; it just has not been written to the architectural register yet. This new hardware path would eliminate all the stall cycles in this 2-instruction example.

Additional bypass paths are needed in this simple pipeline to prevent read-after-write stalls caused by reading a register two and three instructions after the one that writes it. A fully general implementation of bypassing—which is the usual practice—would require bypass paths from the EX, MEM, and WB stages back to both input ports of the ALU. Depending on the details of the architecture and implementation, additional paths might be needed for branches and for store instruction data. And of course the floating-point pipeline would need some bypasses of its own.

Models

We modelled a single-issue MIPS design with separate integer and floating-point pipes, much like the MIPS R3000/R3100 [15]. The bypassing structure in our model is shown in Figures 1 and 2.

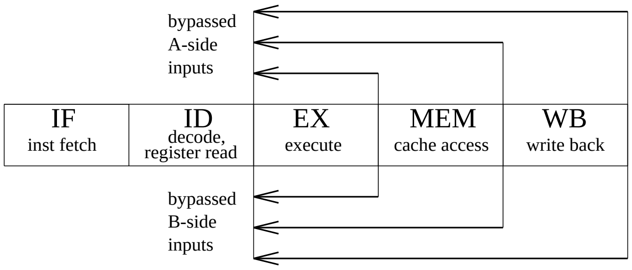

Figure 1: Integer pipeline organization, showing bypassing

Integer bypasses carry results from the final three pipe stages to the two inputs of the EX stage, which we will call the A side and B side. The use of register operands on the two sides in the integer execution unit is architecturally specified: most arithmetic instructions of course use both, but some instructions use only one or the other. Loads, arithmetic with immediates, branches using a comparison with zero, and a few others use only the A side; some shifts and a few other instructions use only the B side. A couple of jumps and a small number of special instructions need no register operand at all.

The floating-point pipeline, shown in Figure 2, presents a more complex bypassing picture. In the floating-point pipe, double-precision arithmetic results use 64-bit bypasses, but because the cache interface is modelled as 32 bits wide, a load of a double precision operand will require two 32-bit loads and potentially result in a subsequent consuming instruction using words from two stages simultaneously. Furthermore, cache data is piped forward independent of the arithmetic result, and the floating-point register file is modelled as having two write ports.

Floating-point arithmetic instructions spend a variable number of non-overlapped cycles in the FX1 stage: adds and subtracts take 2 cycles; multiplies take 4 in single-precision and 5 in double; and divides

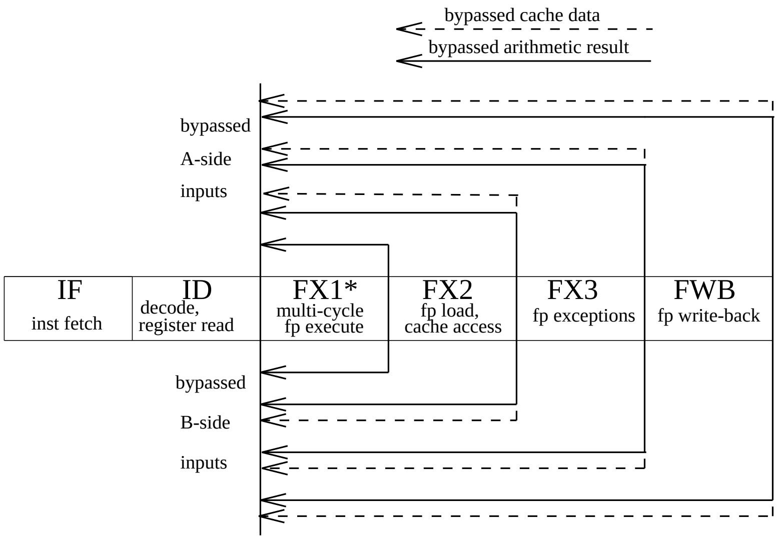

Figure 2: Floating-point pipeline organization, showing bypassing

take 12 cycles in single precision, 19 in double. All floating-point instructions use the A side of the execution unit. A small number of instructions (converts, negate, etc.) use only the A side, but most instructions use both sides. Floating-point stores get their data from the B side, but their addresses, like those of floating loads, come from the A side of the integer pipe. Because we are interested in bypass use, we will use the term "floating-point instructions" throughout this paper to include floating stores and moves.

In this paper we treat a bypass as a logical connection from a pipestage result to the input of an execution unit. Different implementations are of course possible: the Digital Alpha 21064 [18, 8], for example, has a separate set of wires for each source pipestage and wide multiplexors physically near the execution units; the Alpha 21164 [3, 5], on the other hand, has a single multi-driver bus for each of the two inputs of each functional unit, with physically distributed control.

Simulator

We performed our experiments using a modified version of CL-SPIM, a cycle-level extension to Larus's instruction-level MIPS simulator, SPIM [16]. The bypasses shown in Figures 1 and 2 were instrumented for our experiments. We modelled bypassing from the register write-back stages in both pipes as separate pairs of inputs to the multiplexors; in some designs this function could be implemented inside the register file itself.

CL-SPIM can simulate a detailed memory system including caches, but for the purposes of our experiments we chose to model a simple pair of infinite caches to reduce the time needed for simulation. Load hits had a single delay slot in both pipes.

Benchmarks

We used four of the six SPEC92 integer benchmarks and four of the fourteen floating-point benchmarks [1] for our experiments. Table 1 briefly describes them, including short descriptions of the concessions we made to the speed of the simulator. The four integer benchmarks are written in C and were compiled using the MIPS C compiler, version 2.10; the floating-point benchmarks were compiled with the MIPS FORTRAN 77 compiler. The standard reference input was used for *compress*, but the uncompress step was skipped; standard short inputs were used for *espresso* and *sc*; in *eqntott* we reduced the number of variables and equations from 12 to 9; *hydro2D* ran with the number of calculated timesteps reduced from 400 to 30; *ora* ran with 1520 iterations instead of 15200; in *swm256* we reduced the number of iterations from 1200 to 120, and also reduced the grid size from 256 × 256 to 50 × 50; finally, *tomcatv* was run unaltered.

3 Baseline measurements

In this section we report the results of measuring bypass use in our baseline processor model, the model with all bypasses present. As each instruction passes through the simulated pipelines, it gets its register operands (if any) via two input multiplexors in front of the EX or FX1 stages. The simulator keeps a count of each choice.

Integer pipe

Table 2 shows the bypass usage for all of the integer benchmarks, normalized to the number of instructions executed. Thus, for example, 18.3 percent of the instructions executed in *compress* got their A-side register operands via the pipeline's EX-stage bypass. All of the benchmarks used A-side bypassing roughly twice as often as B-side bypassing, in almost every stage. This tendency has an architectural basis: several common instructions have no register operand on the B side. All of the programs except *eqntott* (on both sides) used bypassing less as the pipeline distance in-

Table 1: Benchmarks

Benchmark	Description	Instructions	Input Details
compress	data compression	90M	std. long input, no uncompress
espresso	logical CAD	30M	std. short input
eqntott	logical CAD	30M	fewer variables, equations
sc	spreadsheet	65M	std. short input
hydro2D	Navier Stokes equation solver	15M (fp)	reduced timesteps
ora	ray tracer	7M (fp)	reduced iterations
swm256	weather prediction program	24M (fp)	reduced iterations, smaller grid
tomcatv	vectorized mesh generation program	11M (fp)	no changes

Table 2: Bypass usage in integer benchmarks (normalized to instruction count)

		source of register operand				
		bypasses			no bypasses	
		EX	MEM	WB	Reg file	none
compress	A	0.183	0.118	0.044	0.404	0.250
	B	0.109	0.077	0.025	0.378	0.411
eqntott	A	0.078	0.163	0.110	0.506	0.143
	B	0.032	0.057	0.010	0.473	0.428
espresso	A	0.194	0.113	0.065	0.370	0.257
	B	0.107	0.058	0.036	0.402	0.396
sc	A	0.144	0.112	0.032	0.473	0.239
	B	0.055	0.045	0.018	0.474	0.408

creased: EX sources were used more than MEM, which were used more than WB.

We also measured the "cross-product" of the register operand sources on the two sides: each instruction can be characterized by its choices on the two input multiplexors. Table 3 shows the average bypass behavior of the integer benchmarks on our baseline model. It records for each (A-side, B-side) pair the average fraction of instruction executions that used that pair. For example, it shows that an average of 0.3 percent of all instruction executions used the EX bypass for their A-side operands and the WB bypass for their B-side operands. An example of such an instruction is the `add` in the sequence below:

```
and  r1,r2,r3     ;puts r1 on WB bypass for add
or   r4,r5,r6
xor  r7,r8,r9     ;puts r7 on EX bypass for add
add  r10,r7,r1    ;gets r7 on A side, r1 on B side
```

Table 3 therefore describes the entire runs of the benchmarks from the bypassing point of view. The row and column totals measure the total number of uses of a particular bypass on one side, independent of the other side. Thus, for example, an average of 12.7 percent of all instructions, according to the table, bypassed their A-side inputs from the MEM stage. This matches the one-dimensional view of Table 2 in the obvious way.

The northwest corner of Table 3 describes instruction executions that needed *no* bypassing, either because they had no register operands (like `jmp` instructions), or because both of their operands came from the register file itself, without benefit of bypassing. In these programs, about half of all instructions executed are in that upper corner.

The other half needed bypassing to avoid stalling. Of these, about half (an average of 22.6 percent of the total) needed only the closest bypass, the path from the EX stage. The remainder of the instructions used various combinations of bypass paths, with modest tendencies to need closer bypasses more than farther ones, and to need A-side bypassing more than B-side. We will explore incomplete B-side bypassing in Section 4 of this paper.

Floating-point pipe

A similar set of measurements was made of the floating-point benchmarks, using the pipeline model of Figure 2. The story is a little more complicated than the integer story, due to the mixture of double and single precision and to the 32-bit interface to the cache. One double-precision instruction could potentially get its four 32-bit input words from four separate bypasses! Thus Table 4 is a good deal bigger then Table 2. It separates bypassed arithmetic results from the results of load instructions and also reports use of the high-order and low-order words separately. Single-precision operations, predominant in the *swm256* benchmark, get data only on the low sides of the FX1 input multiplexors. The three double-precision benchmarks bypass only 64-bit arithmetic results on the A side, and so have identical entries for the high and low words in the left-hand columns of Table 4.

How many floating-point instructions (including, as usual, floating stores) needed a bypass? Table 4 shows the lowest usage of B-side bypassing to be in *hydro2D*: just over half of its instructions got their B-side operands from the register file, or had none. Thus in all four programs, at least half of the instructions used some bypass.

These programs share with their integer brethren a tendency to prefer close bypasses to far ones, but only for inputs produced by arithmetic operations. The piped cache data shows no such tendency. There is, however, an understandable pattern in the cache data

Table 3: Average bypass usage in integer benchmarks (normalized to instruction count)

A-side operand	B-side operand: No Reg	Reg File	Ex	Mem	Wb	Totals
No Reg	0.039	0.155	0.013	0.012	0.003	0.222
Reg File	0.231	0.122	0.046	0.030	0.009	0.438
Ex	0.067	0.074	0	0.006	0.003	0.150
Mem	0.050	0.054	0.016	0	0.006	0.127
WB	0.024	0.026	0.001	0.011	0	0.063
Totals	0.411	0.432	0.076	0.059	0.022	1.000

Table 4: Bypass usage in floating-point benchmarks (normalized to floating-point instruction count)

		bypassed arithmetic results								bypassed cache data						no bypass		
		FX1		FX2		FX3		FWB		FX2		FX3		FWB		Reg		
		hi	lo	hi	lo	hi	lo	hi	lo	hi	lo	hi	lo	hi	lo	hi	lo	none
hydro2D	A	.131	.131	.017	.017	.013	.013	.007	.007	.047	.004	.049	.033	.045	.057	.380	.428	.281
	B	.079	.116	.025	.037	.059	.024	.019	.013	.093	.008	.038	.095	.029	.032	.387	.415	.109
ora	A	.077	.077	.029	.029	.020	.020	.000	.000	.015	.004	.003	.015	.010	.003	.495	.501	.335
	B	.290	.312	.098	.027	.004	.003	.017	.000	.067	.000	.002	.076	.059	.006	.266	.317	.052
swm256	A		.147		.027		.013		.000		.037		.025		.012		.575	.169
	B		.288		.038		.062		.025		.018		.050		.063		.431	.028
tomcatv	A	.091	.091	.018	.018	.073	.073	.009	.009	.030	.026	.013	.036	.035	.009	.421	.428	.310
	B	.153	.211	.067	.081	.059	.003	.022	.009	.071	.018	.061	.080	.045	.052	.319	.346	.046

portion of the table. A double-precision arithmetic operation would be likely to get just-loaded data from bypass sources that are adjacent in the pipeline, with the high-order word coming from one stage and the low-order one from the next. Thus in the non-*swm256* rows of the table, the value for the high-order word of cache data sources FX2 and FX3 is quite close to the value for the low-order word of sources FX3 and FWB, respectively.

The integer programs used A-side bypassing considerably more than B-side, in large part because a healthy percentage of integer instructions had no B-side operand. Here the situation is reversed, and for a similar reason: almost all of the floating-point instructions have operands on the B side, but stores in particular have no (floating) operand on the A side.

4 Performance with incomplete bypassing

When an instruction needs a path that is missing, the modelled machine must stall. It would be very convenient if the number of stalled cycles was the same as the total number of uses of the missing bypass path, as reported in Tables 2 and 4. But this is not correct. An instruction's one-cycle stall to avoid a missing bypass also shifts the bypassing relationships of the next few instructions. This shift can affect those instructions' own need for bypass stalls. Consider this integer code fragment:

```
add  r1,r2,r3      ;puts r1 on WB bypass for or
sub  r4,r5,r6      ;puts r4 on WB bypass for and
xor  r7, r8,r9
or   r10,r11,r1    ;gets r1 (B-side) from add
and  r12,r13,r4    ;gets r4 (B-side) from sub
```

If the B-side bypass from the WB stage is missing, the or instruction will stall for one cycle. This cycle will allow the sub to finish writing r4 before the and needs it. The and, an apparent user of the missing bypass, is held back for one cycle and thus escapes its stall by reading its r4 operand from the register file. Thus we have two apparent uses of the missing bypass leading to just one stalled cycle.

It can go the other way too, of course. Consider this fragment:

```
add  r1,r2,r3      ;puts r1 on WB bypass for or
sub  r4,r5,r6
xor  r7,r8,r9      ;puts r7 on MEM bypass for and
or   r10,r11,r1    ;gets r1 (B-side) from add
and  r12,r13,r7    ;gets r7 (B-side) from xor
```

If the B-side WB bypass is taken away, the or instruction will stall, forcing the and, an apparent user of the MEM-stage bypass, to stall as well, since it will now be held back for one cycle and need the WB bypass. So here one apparent use of the WB bypass has led to two stalled cycles.

These and similar effects required the re-simulation of the benchmarks on a modified model in order to

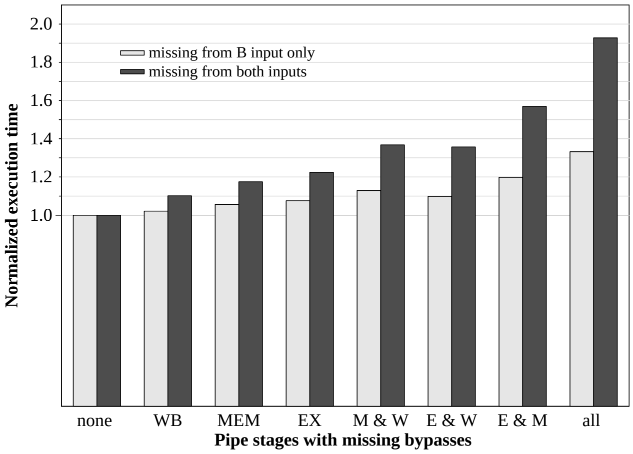

Figure 3: Incomplete bypassing in integer benchmarks

get accurate cycle counts and usage statistics. The CL-SPIM model was altered so that an instruction would wait in the ID stages of the appropriate pipeline until its register operands (if any) were available on the execution unit input multiplexors. A needed but absent bypass would stall the instruction for one cycle. If in the next cycle a different bypass was also missing, the instruction would stall again. Performance was measured as number of cycles of execution.

Integer results

There are three bypass sources in the integer pipeline and hence 8 possible combinations of present or absent bypass sources. Each source may be connected to 2 EX-stage input multiplexors, so there are therefore a total of 64 configurations of the bypassing network. We did not look at all 64. We investigated the 16 configurations in which bypass paths were removed either from the B-side multiplexor or from both multiplexors.

Figure 3 shows the execution time of the integer benchmarks on each configuration, normalized to the time for the baseline model (all three bypasses present), and averaged over all four benchmarks. Figure 3 quantifies the value of these bypasses in this particular model for these programs, under our liberal cache assumption. A more realistic cache model would add stalled cycles to the execution time of all 16 configurations and thereby flatten the bar graphs somewhat.

The measurements show that on average these programs would be slowed down by between 2 and 33 percent if various bypasses were removed from the B side (and the code not changed). The relative value of these different bypasses is roughly as predicted by Table 2, with one interesting difference. If just one path is removed from the B-side, then the most valuable path is the one from EX, then MEM, then WB. But if two B-side paths are removed, leaving just one, then the fastest configuration is not the one with just the EX path left. Figure 3 shows that the fastest one-path model is the one with the MEM bypass. This is because the other two have missing paths in *adjacent* pipe stages (EX and MEM, or MEM and WB), which can lead to 2-cycle stalls when an instruction needs a path from the earlier stage. A 2-cycle stall cannot occur in the MEM-only configuration. All three configurations with two paths missing are slower than any of the three with just one missing.

These same properties hold for the configurations with bypasses gone from both sides (the dark bars), and the slowdown for them is over twice the slowdown for B-side removal only, since A-side bypassing is more heavily used, as we observed earlier. The least valuable bypass (from the WB stage) still slows down the model by 10 percent if it is missing. If all the bypasses are missing, these programs suffer an average slowdown of just over 90 percent. This number characterizes the gross value of bypassing in this model (albeit with perfect caches).

In Section 5 we will explore some code-transformation remedies for these performance penalties.

Floating-point results

Because the floating-point pipe is considerably more complex than the integer pipe, there are many more partial bypass configurations. There are 7 sources of bypassed results (4 arithmetic and 3 cache—see Figure 2), and each may be connected to the high or low word of the 2 multiplexors of the FX1 stage inputs. Thus the design space of bypass network configurations contains $4*2^7$, or 512 separate points. We visited only certain neighborhoods in this design space. In particular, any bypass removal involved both high- and low-order words of the input multiplexor. Also, experiments with bypasses from the last three pipe stages treated the arithmetic operand and the cache data operand together.

We did not study *integer* bypassing in the floating-point experiments, although each benchmark executes many integer instructions. These instructions mean that the bargraphs of Figure 4, which illustrates incomplete bypassing in the floating-point benchmarks, are flatter than the integer ones of Figure 3. Integer execution is unaffected by floating-point bypass removal. Each bar represents normalized execution time, averaged over the four benchmarks.

Without any floating-point bypassing at all, the model suffers a slowdown of just over 50 percent. If B-side bypassing only is removed, this declines to 36 percent, and if bypasses are taken off the A side instead, it declines further, to 20 percent. All of the other configurations represented in Figure 4 cost a slowdown of under 10 percent, with the least harmful one (no bypasses from FX2 stage) adding just 2.7 percent more cycles of execution.

The methods of the next section will siginificantly reduce some of these penalties.

5 Code transformations

Commutative swaps

When the bypassing network is asymmetric with respect to the two functional unit inputs, there are opportunities for a simple architectural palliative that

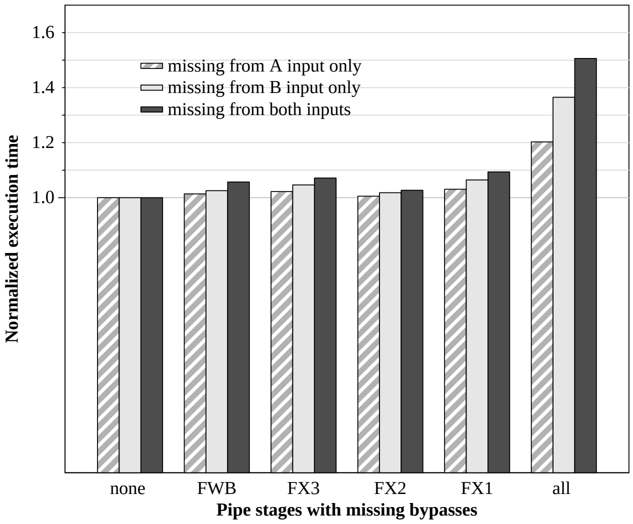

Figure 4: Incomplete bypassing in floating-point benchmarks

can improve the performance of these models significantly.

Consider this code fragment:

```
and  r1,r2,r3     ;puts r1 on B-side MEM for xor
sub  r4,r5,r6     ;puts r4 on A-side EX for xor
xor  r7,r4,r1     ;gets r4 on A side, r1 on B side
```

The `xor` instruction gets its A-side operand from the EX bypass and its B-side operand from the MEM bypass. If the B-side MEM path were missing in the hardware (and the WB path present), the `xor` would need to stall for one cycle so that its B-side operand could move forward to the WB stage, where a bypass is possible. But because the `xor` instruction performs a commutative operation on its register operands, the input operand register numbers could be swapped with no logical effect on the program. Were this done, the `xor` would no longer need to stall if the B-side MEM bypass were missing:

```
and  r1,r2,r3     ;puts r1 on A-side MEM for xor
sub  r4,r5,r6     ;puts r4 on B-side EX for xor
xor  r7,r1,r4     ;gets r1 on A side, r4 on B side
```

The potential for widespread use of this *commutative swap* operation is illustrated by Table 5, which reports average bypass use by all of the commutative instructions in the integer benchmarks. The fractions in the table are of the total number of instruction executions, not just the commutative ones. In the integer programs about 28 percent of the instructions are commutative, on average, and they are heavy users of bypassing. Furthermore, very few instructions need bypassing for both operands. (We suspected that a simulator bug might have been responsible for the zeroes on the diagonal of Table 5, and verified that there was no such bug. These programs simply never need the same bypassed result on both sides.)

Table 5: Average bypass usage for commutative instructions in integer benchmarks (normalized to instruction count)

A-side	B-side operand				
operand	Reg File	Ex	Mem	Wb	Totals
Reg File	0.080	0.032	0.014	0.006	0.132
Ex	0.060	0	0.004	0.002	0.066
Mem	0.048	0.009	0	0.003	0.059
Wb	0.017	0.001	0.007	0	0.025
Totals	0.205	0.041	0.025	0.011	0.282

If a single B-side bypass is missing from the integer pipe, the instructions represented by the corresponding column in Table 5 can be "pivoted" into the matching row of the table by applying commutative swaps. Thus the `xor` instruction in the example above would move from the (EX, MEM) bin to the (MEM, EX) bin to accommodate a missing MEM-stage bypass on the B side.

Just as before, inter-instruction effects required rerunning the model configurations with commutative swapping. Rather than alter the commutative instructions at compile-time, we modelled the swap at simulated run-time. A commutative instruction had its operand register numbers swapped as it passed through the ID pipeline stage, if the swap would prevent or reduce stalls due to missing bypasses *and* if the instruction was in the same basic block as the writer of the register whose access caused the stall. We did the experiments this way because it was easier for us than modifying the compiler, which could certainly perform this optimization in principle.

Figure 5 shows the results from the integer benchmarks for all configurations of B-side bypasses. In this figure, normalized runtimes with swapping are superimposed on the runtimes without swapping (from Figure 3). Swapping was quite effective in reducing runtimes. Stalled cycles from missing bypasses were reduced by roughly one third to one half, across all configurations. Four configurations had runtimes no more than about 5 percent slower, on average, than the fully bypassed model.

In the floating-point model we looked at a more limited set of configurations, but considered A-side inputs as well as B-side ones. In the floating-point programs, about 47 percent of the floating-point instructions are commutative, on average. Figure 6 shows the effects of swapping in configurations with bypasses missing from the FX1 stage, and also in configurations with all bypasses gone from one or the other side. Just as in the integer model, swapping is very effective at reducing the stalls introduced by the absent bypasses. Perhaps the most striking result is that with no bypassing on the A side at all, commutative swaps are able to reduce the average slowdown from 20 percent to just over 8 percent.

Instruction scheduling

Instruction scheduling, both within and between basic blocks, can further improve the performance

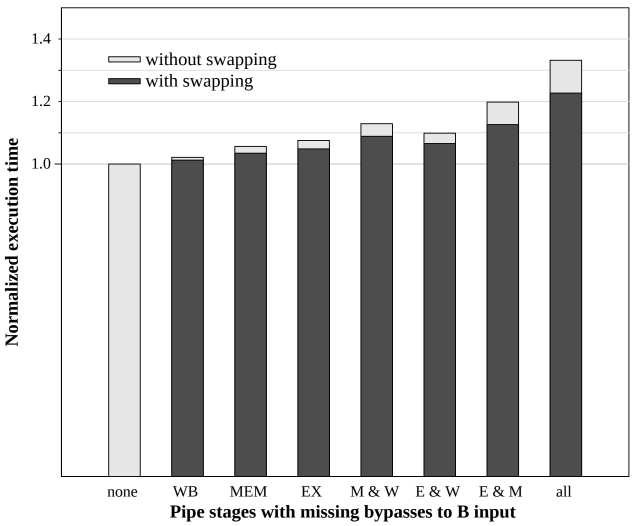

Figure 5: Asymmetric bypassing in integer benchmarks, with and without commutative swaps

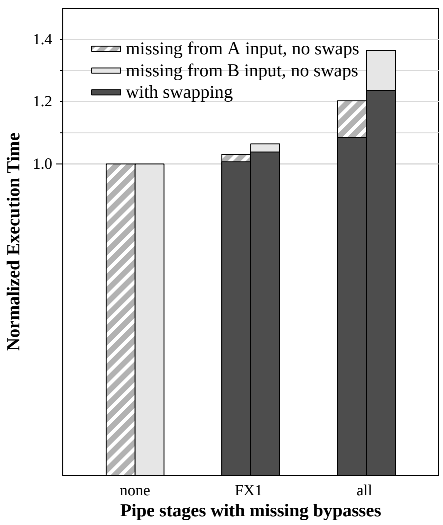

Figure 6: Asymmetric bypassing in floating-point benchmarks, with and without commutative swaps

of asymmetric bypassing. When processing a basic block, if swapping operands is not beneficial or even possible, it may be advantageous to rearrange the code itself so as to avoid using missing bypasses. If a bypass connection is missing from both sides of the execution unit, then of course commutative swaps cannot compensate at all for the missing paths. The code-rescheduling recourse remains, however.

While techniques such as trace scheduling [9, 17] would allow the compiler to look for swapping and scheduling opportunities across basic block boundaries, we will confine ourselves to simple basic-block methods. Code scheduling to avoid absent bypasses closely resembles scheduling to avoid register interlocks [12, 10]. We adapted Gibbons and Muchnick's DAG-based scheduling algorithm [10] for our purposes. Our algorithm differs from the one described by Gibbons and Muchnick in two ways. First, we examine more of the previously scheduled instructions to determine if an instruction would cause a stall. The exact number of previously scheduled instructions that need to be examined is determined by the missing bypasses, but is guaranteed to be no more than the distance from the register-read pipestage (the ID stage in both pipes of our model) to the furthest missing bypass. And second, we use an additional heuristic for choosing the next instruction to schedule from a set of candidate instructions. In particular, we must consider the possibility that an instruction that does not stall in this cycle might stall if scheduled later; if there is such an instruction it is scheduled. If this heuristic fails, we schedule a (non-stalling) candidate that has a successor that might stall for a result produced by the candidate. Sometimes, of course, a stall is unavoidable.

Lacking access to the source code for the MIPS reorganizing assembler, we could not implement our rescheduling algorithm directly. Instead, we manually optimized the assembly code for our particular benchmarks and bypass configurations, choosing for rescheduling a manageably small number of instructions responsible for a high percentage of stalls. To perform these experiments we needed to dictate explicit instruction sequences to the MIPS assembler. The MIPS assembly directives "set noreorder" and "set reorder" can be used for this purpose. A tedious complication of this approach is that fixing some instruction sequences may change the order produced by the assembler for the remaining instructions, which could change the number of instructions executed and thereby skew our results. To account for this, we compared the performance of our hand-optimized code to code that used the noreorder/reorder directives to fix in its original order each of the rescheduled instruction sequences. This forced all instruction sequences outside the directives to be the same in both versions.

We performed code scheduling experiments using the integer benchmark *compress* in three missing-bypass configurations, and the floating-point benchmark *tomcatv* in two configurations.

First, we rescheduled *compress* to run on a configuration that had no bypassing from the WB stage. Without scheduling *compress* is 8.6 percent slower on

Table 6: Code-scheduling experiments: execution times normalized to fully-bypassed configurations

benchmark	bypass configuration	without scheduling or swapping	with swapping only	with scheduling and swapping
compress	No WB	1.086	N.A.	1.036
	No B-side EX	1.109	1.063	1.033
	No B-side at all	1.486	1.287	1.226
tomcatv	No FX1	1.081	N.A.	1.060
	No B-side FX1	1.060	1.042	1.021

this configuration. Ten instructions accounted for 93 percent of the bypass stalls. Of these seven could be rescheduled to remove the stalls. Dependences between instructions prevented us from rescheduling the remaining three instructions. Reordering these seven instructions removed 62 percent of the stalls caused by eliminating the WB bypasses. The end result was that the rescheduled code was 3.6 percent slower on this configuration than the fully bypassed configuration.

Second, we experimented with a configuration that eliminated the EX bypass only on the B-side. As discussed earlier, swapping the operands of commutative instructions can substantially reduce the number of stalls caused by an asymmetric bypass network. In this experiment, we combined rescheduling and swapping. Eight instructions in *compress* accounted for 90 percent of the bypass stalls that remained after swapping. Four of these could be rescheduled to remove 48 percent of the total stalls. The resulting program was 3.3 percent slower than the fully bypassed configuration as compared with the swapped version, which was 6.3 percent slower.

The final experiment with *compress* used a configuration that eliminated bypassing from the B-side entirely. Again, we combined rescheduling and swapping to reduce the stalls introduced by the missing bypasses. (Using this configuration, *compress* with swapping was 29 percent slower than the fully bypassed version.) Eleven instructions account for 86 percent of the bypass stalls that cannot be eliminated by swapping. Of these, five could be reordered to remove 29 percent of the stalls. The final program was 23 percent slower than the fully-bypassed configuration.

We did two scheduling experiments with *tomcatv*. First, we experimented with removing the FX1 bypass from the floating point pipeline. Without rescheduling *tomcatv* is 8.1 percent slower on this configuration than on the fully bypassed configuration. Forty instructions are responsible for 98 percent of the stalls. Of these nine can be rescheduled, which reduces the number of bypass stalls by 26 percent. The resulting program is 6.0 percent slower than the original program on the fully bypassed configuration.

Second, we removed the FX1 bypass only on the B side. As in the integer experiments, we combined rescheduling with swapping. Twenty-two instructions represent 96 percent of the instructions that continue to suffer bypass stalls after swapping. Eight of these can be reordered to recover 49 percent of the remaining stalls. The rescheduled program is 2 percent slower on this configuration than the original program on the fully bypassed configuration, as compared to 4 percent slower with swapping but without rescheduling.

All of this is summarized in Table 6. The N.A. entries in the table occur because swapping is only meaningful in the asymmetric configurations.

6 Conclusion

We have shown in this paper that some incompletely bypassed pipeline configurations, especially when accompanied by simple code transformations, can perform well. Thus, while a reduction of ILP-enhancing hardware—the bypass network—can be expected to reduce ILP, we have shown how that loss can be minimized.

Eight of the SPEC92 benchmarks (four integer, four floating-point) were run in a cycle-level simulator of a model based on the MIPS R3000 design [15], with instrumented bypasses. Patterns of bypass use in the benchmarks were measured. Removing combinations of bypass paths from one or both sides of the execution unit slowed down the benchmarks (assuming perfect caches) by from just a few percent to over 90 percent, depending on the configuration. In asymmetric configurations, much of the lost performance could be regained by simply interchanging the operands in instructions that perform commutative operations. Our code-scheduling experiments showed that many of the remaining interlock stalls could be eliminated by reordering a small number of instruction sequences to avoid missing bypasses. Code scheduling was also productive for configurations in which commutative swaps were ineffective. It seems likely that more sophisticated and systematic code-scheduling methods could do even better, and could accommodate more complex configurations of bypass paths.

All of our normalized execution times come from models with perfect caches. Realistic caches would reduce the relative cost of missing bypasses. Our results are therefore an upper bound on these costs for these programs running on this model.

Further work suggested by these results includes modelling and measuring more complex designs with multiple instruction issue, and/or deeper pipelines; incorporating realistic cache behavior into the measurements; investigating additional portions of the very large design space of incomplete bypass network configurations; exploring more sophisticated code scheduling algorithms, including ones that operate across basic block boundaries; and incorporating these algorithms into the compiler or assembler.

Future designs, with wider instruction issue and deeper pipelines, will need still more bypassing. Our results suggest that it should be possible for future microprocessor designs to save hardware, or cycle time, or both, by implementing incomplete bypassing together with code transformations to minimize interlock stalls. It may also be possible to design longer pipelines, or wider instruction issue, than would seem feasible if a complete set of bypasses was required.

Acknowledgements

We are grateful to the anonymous referees for their helpful comments. This work was supported in part by the National Science Foundation under grant CCR–9423123.

References

[1] *SPEC Newsletter*. Standard Performance Evaluation Corporation, September 1994.

[2] A. Abnous and N. Bagherzadeh. Pipelining and Bypassing in a VLIW Processor. *IEEE Transactions on Parallel and Distributed Computing*, 5(6):658–664, June 1994.

[3] P. Bannon and J. Keller. Internal Architecture of Alpha 21164 Microprocessor. In *Proceedings of the IEEE COMPCON '95 Conference*, pages 79–87, 1995.

[4] E. Bloch. The Engineering Design of the Stretch Computer. In *Proc. Eastern Joint Computer Conference*, pages 48–59, 1959.

[5] W. Bowhill et al. A 300-MHz 64b Quad-Issue CMOS RISC Microprocessor. *IEEE J. of Solid-State Circuits*, 30(11):to appear, November 1995.

[6] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *The 25th Annual International Symposium on Microarchitecture*, pages 292–300, December 1992.

[7] R. Cohn, T. Gross, M. Lam, and P. S. Tseng. Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor. In *The 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, April 1989.

[8] D. Dobberpuhl et al. A 200-MHz 64-b Dual-Issue CMOS Microprocessor. *IEEE J. of Solid-State Circuits*, 27(11):1555–1564, November 1992.

[9] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[10] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, June 1986.

[11] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadeh. VIPER: A VLIW Integer Microprocessor. *IEEE Journal of Solid State Circuits*, 28(12):1377–1383, December 1993.

[12] J. L. Hennessy and T. R. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[14] I.-J. Huang and A. M. Despain. An Extended Classification of Inter-instruction Dependency and Its Application in Automatic Synthesis of Pipelined Processors. In *The 26th Annual International Symposium on Microarchitecture*, pages 236–246, December 1993.

[15] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[16] J. R. Larus. *Assemblers, Linkers, and the SPIM Simulator*. Appendix A of *Computer Organization & Design: the Hardware/Software Interface*, D. A. Patterson and J. L. Hennessy, Morgan Kaufmann, 1994.

[17] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnel, and J.C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[18] E. McLellan. The Alpha AXP Architecture and 21064 Processor. *IEEE MICRO*, 13(3):36–47, June 1993.

[19] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: the Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1994.