

The Performance Impact of Incomplete Bypassing in Out-of-Order Processors

T.J.Highley KevinSkadron

DepartmentofComputerScience
OlssonHall
UniversityofVirginia
Charlottesville,Virginia
{tjhighley,skadron}@cs.virginia.edu

Abstract

In an out-of-order processor, bypassing is used to reduce the latency between the time that results are produced and the time those results are available to instructions that need them. If bypassing was removed, the processor should not run any faster. However, since the instructions are issued out of order, resource utilization might remain high even without bypassing. Bypassing networkstake up a large amount of space on the die; it might be supposed that the impact of their removal on performance may not be very dramatic. However, our results show that for some out-of-order machines, there is significant performance degradation if the bypassing networks are removed.

1 Introduction

Bypassing is neither extremely complicated nor a new innovation. It involves the addition of extra datapaths to provide computation results to waiting instructions without waiting for those results to be written back to architectural registers. The concept has been around since Bloch first described it in 1959 [Blo59].

Ahuja, Clark, and Rogers found that the removal of some bypassing networks in a pipelined in-order processor does not necessarily significantly affect the speed of the processor, particularly if compiler modifications take into account the absence of certain bypassing networks [ACR95].

This work was inspired by those results. Our hypothesis was that the removal of bypassing networks in an out-of-order processor would also have only a small effect on the performance of the processor.

Section 2 of this paper describes the equipment and software used to perform our tests. Section 3 describes our baselineresults. Section 4 describes what we found when we removed the bypassing networks. Section 5 describes the conclusions we have drawn from our results.

2 Experimental Framework

We are using the SimpleScalar/PISA simulation tool as the basis for our work, specifically the

sim-outorder simulator from that software suite. This tool is a detailed simulation of a processor that issues instructions out of order. See [AuB97] for more details. We ran most of four experiments under Linux on a dual Pentium III 450MHz. The only exception is the m88ksim program from the SPEC95 benchmark which was run on a SPARC server 20.

While we used SimpleScalar as a basis, we made some modifications in order to more closely model a specific architecture. We selected the Alpha 21264 [KMW98] because it is an aggressive out-of-order machine. We accepted a number of simplifications in our simulation in favor of ease of implementation. For instance, we did not distinguish between the two clusters that the 21264 uses during instruction slotting. It follows naturally from that fact that neither did we take into account the one cycle of additional latency that is incurred when an instruction needs to use an operand that was generated in the other cluster. We did not closely model the set-predict algorithm that the Alpha uses during instruction fetch. The instruction window in the 21264 is 80 instructions, but we used 64 because it is a power of two. Rather than using the 21264's own instruction set, we used the portable instructions set architecture (PISA) which SimpleScalar provided. Naturally, there are other simplifications that were made due to the nature of simulations, but these are the major ones. These simplifications will affect the exact

numbers, but have little effect on the conclusions we draw from our results, since our conclusions are based on the relative sizes of those numbers.

In our simulation of the Alpha 21264, we changed the number and makeup of SimpleScalar's functional units. To model the four subclusters of the Alpha 21264, we changed SimpleScalar's `sfu_config` data structure:

```
struct res_desc fu_config[] = {
  {
    "integer-ALU",
    1,
    0,
    {
      { IntALU, 1, 1 }
    }
  },
  {
    "integer-MULT/DIV/ALU",
    1,
    0,
    {
      { IntMULT, 7, 1 },
      { IntDIV, 20, 19 },
      { IntALU, 1, 1 }
    }
  },
  {
    "memory-port/integer-ALU",
    2,
    0,
    {
      { RdPort, 1, 1 },
      { WrPort, 1, 1 },
      { IntALU, 1, 1 }
    }
  },
  ...
};
```

The "memory-port/integer-ALU" functional units represent the two lower subclusters, which each have a memory port as well as an ALU. The "integer-ALU" and "integer-MULT/DIV/ALU" functional units represent the two upper subclusters, as only one of the subclusters has a multiplier.

Some important parameters include a 64K L1 data cache, 2 way set-associative, with an identical L1 instruction cache. The L2 cache was unified, 1 megabyte, and direct mapped. Access latency to the L1 data cache was

estimated to 2 cycles. To model the set-predict algorithm that the 21264 uses on instruction fetch, we simply set the access time of the L1-Cache to one cycle. Memory latency was 100 cycles to access and 2 cycles per unit. Memory access bus width was 8 bytes. We used a 2-level branch predictor with a misprediction latency of 2 cycles. The RUU size was 64 while the LSQ size was 16.

With the bypassing in the Alpha 21264, instructions that complete execution will immediately send their result to instructions that are waiting in the issue queue via a bypass network. The next cycle, the results will be written back to the physical registers. The thrust of this paper is removal of those bypassing networks, such that the instructions in the issue queue would always have to wait an additional cycle to receive any data from instructions that completed ahead of them.

3 Baseline Experiments

We ran several of the SPEC95 benchmark programs on the simulator. The benchmarks were compiled to PISA (portable instruction set architecture). Due to the complexity of sim-out order and the size of the benchmarks, the time needed to run to completion would be quite long. Consequently, none of the benchmarks were run to completion. Additionally, the start-up phase of the programs do not use a representative mix of instructions. Therefore, we used the fast-forward option to skip over a number of instructions at the beginning of each benchmark. After the fast-forward, we collected statistics for another number of instructions. The number of instructions executed for each benchmark was held constant over all variations of four

Benchmark	Cycle to complete	Number of instructions skipped	Instructions counted during statistic gathering
go	62496084	210000000	100000000
gcc	61145540	900000000	100000000
m88ksim	22160237	950000000	50000000
compress	77560763	160000000	100000000
xlisp	58248166	900000000	100000000
jpeg	21465436	823000000	50000000
perl	26113649	195000000	50000000
TOTAL	329189875	922300000	550000000

Table 1: Baseline Experiments

experiments. Our baseliner results are represented in Table 1.

4 Performance with incomplete bypassing

We did not expect the performance to be improved with the removal of bypassing, but we hoped to find that the performance degradation would be negligible.

We ran simulations with five different variations on the incomplete bypassing. The first simulation had no bypassing. The second had bypassing for all address computations. The third simulation had bypassing for integer ALU instructions, but not for any other instructions. The fourth simulation had bypassing only on the two lower subclusters (which are also the memory ports). The fifth simulation retained the bypassing for the two lower subclusters, but gives those ports to address generation instructions if any are at the front of the queue for that cycle.

Our first step in removing the bypassing network was complete removal. By adding one to each of the instruction latencies, it was as though the instructions waiting in the issue queue could not access the results of earlier instructions until after they were written back to the register file. We ran the same benchmarks with this configuration. The results are in Table 2.

Benchmark	Cycle to complete
go	83116985
gcc	72411798
m88ksim	24376268
compress	108268436
xlisp	69428478
jpeg	25833405
perl	31833753
TOTAL	415269123

Table 2: Nobypassing

We next hypothesized that a bypassing network may be of great advantage to some instructions while it is of only slight benefit to other instructions. As a result of this theory, we decided to include a bypassing network for all address computations so that subsequent memory

accesses could start sooner. This was accomplished by checking instructions when they were issued. The latency of any address computation was reduced by one. The results from running the benchmarks with this variation are in Table 3.

Benchmark	Cycle to complete
go	76940990
gcc	68468034
m88ksim	23734288
compress	106090170
xlisp	63861125
jpeg	25071738
perl	30119478
TOTAL	394285823

Table 3: Bypassing on address calculations

We then inserted bypassing networks for all integer ALU instructions. Since the great majority of the instructions in the benchmarks fall under this category, the results here are only slightly worse than the baseliner results. The results were received from running this variation and can be found in Table 4.

Benchmark	Cycle to complete
go	62512241
gcc	61211921
m88ksim	22162765
compress	80629219
xlisp	58248873
jpeg	21695904
perl	26171264
TOTAL	332632187

Table 4: Bypassing for ALU instructions

Our next variation is specific to the Alpha 21264. We added bypassing networks for two of the architecture's four subclusters, specifically the lower two subclusters. This was accomplished by simply adjusting the instruction latencies. Our results from this execution are in Table 5.

Benchmark	Cyclestocomplete
go	81213608
gcc	70623500
m88ksim	24137678
compress	107902630
xlisp	67711738
jpeg	25041608
perl	30942180
TOTAL	407572942

Table5: Bypassing on lower subclusters

Our final variation was to again have bypassing on the lower two subclusters, but to give those clusters to address computations whenever address computations would be in the set of instructions being issued during a cycle when the functional unit is available. This was accomplished by giving priority to both address computations and the lower two subclusters. That way, if there is a functional unit available in a lower subcluster during the same cycle that an address computation is going to issue, they will be matched together since they are both at the top of their respective lists. These results are in Table 6.

Benchmark	Cyclestocomplete
go	72012419
gcc	71692631
m88ksim	30250633
compress	83776078
xlisp	70542128
jpeg	26022810
perl	34291630
TOTAL	388588329

Table6: Bypassing on lower subclusters (address calculations favored)

5 Conclusion

In Table 7 we summarize the results of four experiments.

Description	Totalcycles	Percentchange
Baseline	329189875	n/a
Nobypassing networks	415269123	26.15%
Bypassing only for address calculations	394285823	19.77%
Bypassing for all ALU instructions	332632187	1.05%
Bypassing on two subclusters	407572942	23.81%
Bypassing on two subclusters with preference to address calculations	388588329	18.04%

Table7: Summary of results

These numbers are not what we had hoped, but they are illustrative. We find that the bypassing networks in out-of-order execution can have a large impact on the execution times of programs. The reasons why are readily apparent; they are the same reasons why bypassing networks have been popular all along. It is obvious that bypassing networks will improve the speed of computations, even in an out-of-order machine. What is not obvious is the extent to which they will improve performance, which is what we have investigated. Our tests have demonstrated that for some out-of-order processors, bypassing networks provide a dramatic increase in performance.

It should be pointed out that the specific work of Ahuja, Clark and Rogers would likely still apply to out-of-order machines. That is, the removal of only *one* side of the bypassing network (with associated compiler modifications) would produce only minor drops in performance in an out-of-order machine, just as it has only a minor impact on the speed of an in-order machine. It is recommended that this design be verified.

It is also recommended that the impact of incomplete bypassing be investigated with respect to other out-of-order processor designs. For instance, a larger instruction window (RUU) may provide enough in-flight instructions to keep the functional units busy and maintain a high throughput, thus eliminating the need for the bypassing networks. It has been shown that this is not the case for a 64 instruction window on our simulation of the Alpha 21264, but our results do not preclude the possibility of other designs working well without bypassing. This should be examined.

References

[ACR95] Ahuja P.S., Clark D.W., Rogers A.,
The performance impact of incomplete bypassing
in processor pipelines. *Proc. of the 28th Annual
International Symposium on Microarchitecture,
IEEE Comput. Soc. Press.* pp.36-45, 1995.

[AuB97] Austin T.M., Burger D.C., The
Simple Scalar Tool Set, Version 2.0, *Computer
Architecture News*, 25(3), pp.13-25, June, 1997.

[Blo59] Bloch E., The Engineering Design of the
Stretch Computer, *Proc. Easter Joint Computer
Conference*, pp.48-59, 1959.

[KMW98] Kessler R.E., McLellan E.J., Webb,
D.A., The Alpha 21264 Microprocessor
Architecture, *Proc. International Conference on
Computer Design, VLSI in Computers and
Processors, IEEE Comput. Soc.*, pp.90-95,
1998.