Single FU Bypass Networks for High Clock Rate Superscalar Processors

Aneesh Aggarwal

Department of Electrical and Computer Engineering
Binghamton University, Binghamton, NY 13902.
aneesh@binghamton.edu

Abstract. Microprocessors depend heavily on broadcast-based bypass networks to eliminate pipeline hazards arising due to data dependencies. However, increasing clock speeds make broadcasting slower and difficult to implement, especially for wide issue and deeply pipelined processors. The problem is exacerbated by shrinking feature size, as wire delays become more important than the gate delays.
In this paper, we propose Single FU bypass networks for high clock rate superscalar processors where, instead of a fully connected broadcast-based bypass network, results from an FU are forwarded only to itself. The new bypass network is based on the observations that an instruction's result is mostly used by just one other instruction and that the operands of many instructions come from a single other instruction. The new bypass network results in a significant reduction in the data forwarding latency, while incurring only a small impact (about 2% for most of the SPEC2K benchmarks) on the Instructions Per Cycle (IPC) count. Single FU networks are also much more scalable than broadcast-based networks, as future microprocessors become wider and more deeply pipelined.

1 Introduction

The bypass network lies in the most critical loop in pipelined processors that enables data dependent instructions to execute in consecutive cycles [5]. Prior studies [4][9][10] have shown that an increase of a single cycle in this critical loop reduces the instruction throughput dramatically. Most modern processors use a broadcast-based bypass network, where a result produced by a functional unit (FU) is made available to all the other FUs. With a broadcast-based network, bypassing can take significant wiring area on the chip [2], especially for wide-issue and deeply pipelined processors. In addition, studies [2][6] have shown that the wire complexity with a broadcast-based network grows proportional to the square of the issue width and the pipeline depth, leading to significant data-forwarding delays. This problem is further exacerbated with the growing wire delays in the sub-micron technology era [1][11]. In addition, large number of bypass paths increase the fan-out delay at the source and the fan-in delay at the destination, by increasing both the capacitive load within the network and the multiplexor complexity at each destination [8]. In fact, bypass network latency is expected to set the cycle time of future micro-processors [6][8]. The overall impact of the broadcast-based bypass network complexity is that multiple cycles

may be required to forward the values [8]. With a multi-cycle bypass network, dependent instructions are not able to execute in consecutive cycles, resulting in a decrease in the instruction throughput.

In this paper, we propose a Single FU bypass network for high clock rate superscalar processors. In this bypass network, instead of a fully connected broadcast-based forwarding, an FU's output is only forwarded to its own inputs. Single FU bypass network facilitates low latency and energy-efficient data-forwarding because of a reduction in the fan-in at the inputs of FUs, a reduction in the fan-out at the outputs of FUs, and a reduction in the lengths and the number of bypass paths. We found that with a Single FU bypass network, the IPC is only about 2% less than that of a broadcast-based bypass network.

The rest of the paper is organized as follows. Section 2 discusses the motivation behind Single FU bypass networks. Section 3 presents the Single FU bypass network. Section 4 presents the results of our experiments and also discusses Single Input Single FU bypass network. Section 5 concludes the paper.

2 Motivation and Background

2.1 Impact of Multi-cycle Forwarding

Te measure the impact of increased forwarding latency on IPC, we use the parameters given in Table 1 (on page 8) for a superscalar pipeline shown in Figure 1. The forwarding latency is increased from 0 cycles to 2 cycles. Dependent instructions can execute in consecutive cycles only with a 0-cycle forwarding latency. Figure 2 shows the IPC (along the Y-axis), with varying forwarding latencies, for many of the SPEC2000 Integer and Floating Point benchmarks.

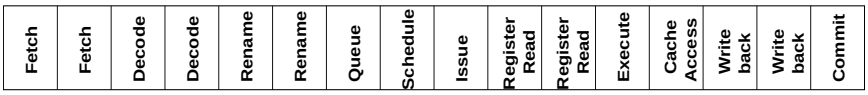

Fig. 1. Base Pipeline

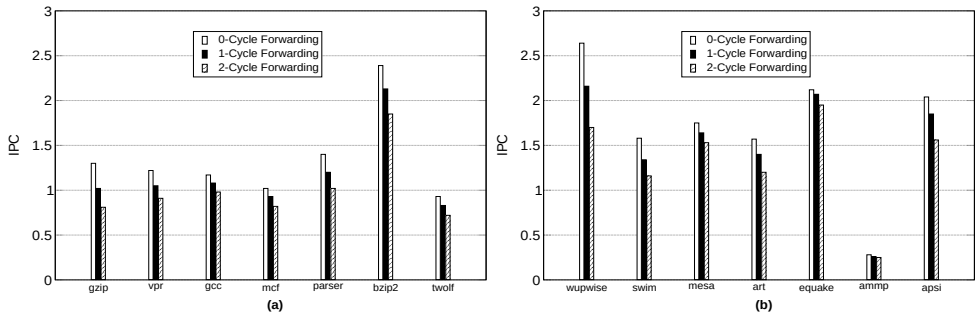

Fig. 2. Impact of Multi-cycle Bypass Networks (a)Integer; (b)FP Benchmarks

As can be seen in Figure 2, IPC reduces significantly as the forwarding latency is increased. For instance, compared to a 0-cycle forwarding latency, IPC reduces by about 15% for a 1-cycle forwarding latency for both integer and FP benchmarks. The impact of increased forwarding latency is relatively higher for

higher IPC benchmarks. The IPC impact is very similar for all the integer benchmarks (except for `gcc`, which has a relatively lower impact). The IPC impact for the FP benchmarks is much more varied.

2.2 Data Dependence Characteristics

To study the typical data dependence characteristics in the programs, we measure the type of instruction producing a value and the type and number of instructions using that value. We define 6 types of instructions, based on the type of functional unit used by the instructions: IALU (simple integer instructions using an ALU), IMULT (complex integer instructions using a multiplier), LOAD (load instructions), STORE (store instructions), FPALU (simple floating point instructions using an ALU), and FPMULT (complex floating point instructions using a multiplier). Figure 3 presents these measurements (for a typical RISC ISA) in the form of a stacked bar graph. For each integer benchmark, there are 2 sets of stacked bars, where each stacked bar represents the type of instruction producing the register value. Similarly, for each FP benchmark, there are 4 sets of stacked bars. The value on top of each stacked bar represents the percentage of results (out of the total) produced by instructions of that particular type and used by other instructions (depicted by the stacks). The total of all the stacked bars is less than 100% because some results are not used at all and some results are produced by instructions of other types.

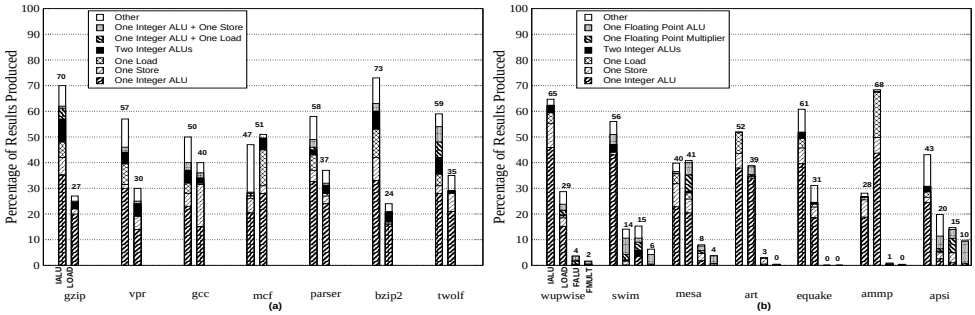

Fig. 3. Result Usage Characteristics for SPEC2K (a) Int and (b) FP Benchmarks

As can be seen in Figure 3, most of the values (about 70%) produced are only used by just one instruction. Figure 3 also explains the different reduction in IPC observed for different benchmarks, in Figure 2. In general, for benchmarks where more results are used by just one other instruction and fewer results are used by store instructions, increased forwarding latency has a larger impact on IPC. This is because, when a result is used by more instructions, after the forwarding latency delay more instructions get ready at the same time, resulting in less effective average forwarding latency per instruction. The effect is exactly opposite in case a result is mostly used by just one instruction. Similarly, if the number of results used by store instructions is higher, the impact of increased forwarding latency is lower, because a delayed store instruction does not have a significant impact. This is the reason for high IPC impact for `wupwise`, `swim`, `art`, and `apsi`, and low IPC impact for `gcc`. `Ammp` also has a high percentage

of instructions with just one consumer, but its extremely low IPC results in a low IPC impact in its case. Other factors affecting the IPC impact of higher bypass latency are the percentage of results (out of the total produced) that are actually used, branch prediction accuracy, and load miss rate.

Next we look at the data dependence characteristics of programs from the perspective of the consumer. Figure 4 presents these statistics[1] in a manner similar to Figure 3, except that each stack now represents operand-producing instructions rather than result-using instructions. For instance, for `gzip`, about 68% of the instructions (out of the total executed) are IALU instructions, and about 30% of instructions (out of the total executed) are IALU instructions whose operands are produced by just one other IALU instruction. Figure 4 shows that a significant percentage (about 70%) of the total instructions executed have their operands produced either by just one other instruction or by no instructions. The LOAD instructions that do not have any producer instructions for their operands are the ones which use the same register operand and this register operand is produced before the start of the collection of the statistics (we collect the statistics after skipping the first 500 million instructions). The IALU instructions that do not have any producer instructions for their operands are mostly the ones which load an immediate value, or the ones that use register r0 to set a register to an immediate value.

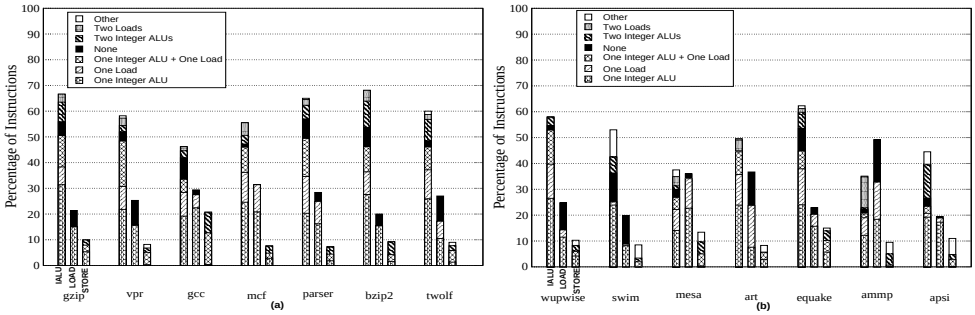

Fig. 4. Operand Characteristics for SPEC2K (a) Int and (b) FP Benchmarks

Overall, for most of the instructions, their results are used by just one other instruction and their operands are produced by just one other instruction.

2.3 Related Work

Bypassing is an old idea and was first described in 1959 by Bloch [3]. Since then, the issue width of the processors and the number of functional units in the processors have increased considerably. Unfortunately, not enough work has been done for efficient data forwarding. Here, we classify the proposed efficient bypass networks into 2 broad categories: *limited bypassing*, and *partitioned bypassing*.

In *limited bypassing*, certain paths are missing from the bypass network. Ahuja et al [2] study bypass networks where the results from the FUs are forwarded to only one of the inputs of all the FUs. They propose simple code

[1] We only show Integer instructions even for the FP benchmarks because of the low percentage of FP instructions in the instruction ranges simulated for the benchmarks.

transformations to avoid the stalls generated due to missing bypasses. For efficient bypassing, the Pentium 4 processor [7], limits the number of bypass inputs into each FU as well as the number of bypass outputs from each FU.

Partitioned bypassing is used in clustered processor architectures. In these architectures, there is typically a broadcast-based bypassing within a cluster, and either broadcast-based inter-cluster bypassing [16][21][15][13][6], or point-to-point inter-cluster bypassing [14][17][18][19][20]. Intra-cluster bypassing is fast because of reduced bypass network complexity, whereas inter-cluster bypassing may take additional cycles because of longer wires and/or multiple hops.

3 Single FU (SFU) Bypass Network

Based on the observations in Section 2, we propose *Single FU (SFU)* bypass networks, where the results produced in a FU are forwarded only to itself, thus reducing the bypass network latency and facilitating high clock rates.

3.1 Basic Idea

Figure 5(a) shows a Pentium 4 [7] style reduced-complexity broadcast-based bypass network for the integer units. A similar bypass network could be implemented for the floating point units. The multi-stage bypass network in Figure 5(a) is responsible for forwarding the correct values. Figure 5(b) shows one configuration of the SFU bypass network for the same set of integer FUs. In this configuration, the output of an ALU is immediately forwarded to only its own inputs. However, the values loaded are typically required in IALU instructions. Hence, instead of forwarding the output of the load unit to itself, it is forwarded to one of the ALUs. In addition, load units typically read the base address from the register file or are, in some cases, forwarded the value from the ALUs. Hence, the output of one of the ALUs is also forwarded to the load unit. Without loss of generality, in Figure 5(b), the output of ALU2 is also forwarded to the load unit and that of load unit is forwarded to ALU2. The rest of the bypass network remains the same. In this configuration, the results from an FU are immediately available to the ones it is directly connected to, and are available to all the FUs after an additional cycle. We call this bypass network as *Limited SFU (LSFU)* bypass network. Figure 5(c) illustrates another configuration of the SFU bypass network. In this configuration, the multiplier and store units are completely isolated, and the results from an FU are only available to the one it is directly connected to; the rest read from the register file. We call this bypass network as *Extreme SFU (ESFU)* bypass network. For all the configurations, a similar bypass network can be assumed for the FP units.

3.2 FU Assignment

The performance of the SFU bypass network relies heavily on the ability to assign instructions to the FUs where their operands are available through the bypass network, for which we propose a post-schedule FU assignment scheme. In this scheme, the FUs assigned to the instructions by the select logic are selectively discarded[2]. Once an instruction is scheduled for execution, it is assigned an FU

[2] However, the conventional select logic is still needed to select the right instructions (based on the priority scheme used, which could be the "oldest" first) to be scheduled.

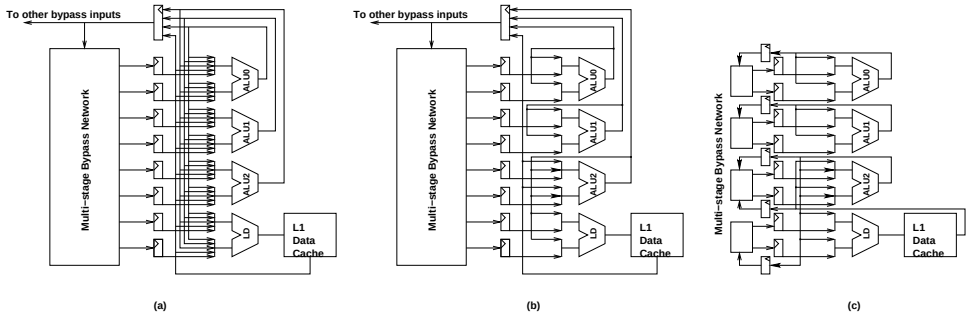

Fig. 5. (a) Conventional Bypass Network; (b) Limited SFU; (c) Extreme SFU

based on where its operands will be available. Figure 6 shows the pipeline for this FU assignment scheme. All the scheduled instructions access an *FU table* (in the *FU assign/arbiter* stage), for each valid operand, to get an FU assigned to them. From the *FU table*, the following information is obtained for each valid operand: (i) whether it is available from the bypass network, and if it is, then in which FU, or (ii) whether it is available from the register file. Based on the information obtained regarding an instruction's operands, an FU is assigned for the instruction as follows:

- If an instruction has only one valid operand with an FU where the operand is available from the bypass network, and the other operand is either not present or is available from the register file or is available in any FU from the bypass network (for the LSFU network), then it is assigned the valid FU.
- If an instruction has multiple operands with valid FUs or an operand cannot be obtained from the bypass network, then that instruction is marked "unscheduled" and it remains in the issue queue.
- For an instruction with no register operands or with all operands available from the register file or in all the FUs from the bypass network (for the LSFU network), the FU assigned by the select logic is used.

Once the FU for an instruction is decided, the *FU table* is updated.

Fig. 6. Post-schedule FU Assignment Pipeline

In this scheme, an issue that needs to be addressed is what happens if multiple scheduled instructions are assigned the same FU? This issue can be resolved by using *FU arbitration*. All the scheduled instructions send a request for the assigned FU. FU arbiters grant the requests of the scheduled instructions, based on priorities which could be the same as that used by the scheduler. In case an instruction cannot acquire the assigned FU, it is "unscheduled" and is again scheduled in the following cycles. Figure 7 shows the *FU assign/arbiter* stage. The register operands are used to index into the *FU table* and the *valid bit table* (indicating whether the *FU table* entry is valid). Based on the valid bits and the FU mappings, an FU is assigned to the instruction. The instruction then sends

a request to that particular FU's arbiter, and updates the *FU table*. The *valid bits* are reset when the register values are available in all the FUs.

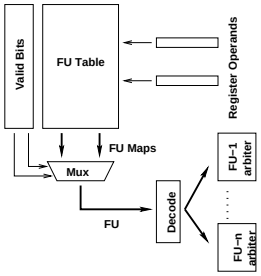

Fig. 7. Schematic FU assign/arbiter stage

To perform FU assignment and arbitration in a single cycle, they have to be fast. *FU arbitration* is similar to the select logic, but of significantly lower complexity because of the limited requests that can be generated. The calculations in [6] suggest that the latency of *FU arbitration* is about 60% less than that of the select logic, for the parameters in Table 1. For a faster *FU table* access, we stack together FUs for multiple registers in a single *FU table* entry. The higher end bits of a register tag index into the table and the lower end bits select the FU for the register. With this design, the access latency of FU table is about 90% less than that of the register file used, in Table 1. With such small FU assignment and arbitration latencies, we assume a single *FU assign/arbiter* stage.

To execute the dependent instructions in consecutive cycles, a scheduled instruction immediately wakes up the dependent instructions. In this case, "unscheduling" a scheduled instruction may lead to the consumer instruction getting executed before the producer instruction. This situation is avoided by keeping a bit-vector (of size equal to the number of physical registers) to indicate whether instruction producing a particular register has been dispatched to the FUs. The instructions check this bit-vector in parallel to FU arbitration. If the producer has not been dispatched to the FU, the consumer is also "unscheduled".

4 Results

4.1 Experimental Setup

The processor parameters used in our experiments are given in Table 1. We use a modified Simplescalar simulator [22] for our experiments. For benchmarks, we use benchmarks from the 7 Integer (gzip, vpr, gcc, mcf, parser, bzip2, and twolf) and 7 FP (wupwise, art, swim, ammp, equake, apsi, and mesa) benchmarks from the SPEC2K benchmark suite compiled with the options provided with the suite. Measurements were performed for 500M instructions, after skipping the first 500M. Latency calculations are performed for a 0.18μm feature size.

In our experiments, we experiment with 4 different bypass networks — fully connected 0-cycle latency bypass network (*FUL0*), fully connected 1-cycle latency bypass network (*FUL1*), *LSFU*, and *ESFU*. The forwarding latencies for the *LSFU* and *ESFU* bypass networks is 0 cycles for direct connections.

Parameter	Value	Parameter	Value
Fetch/Decode Width	8 instructions	*Instr. Window*	128 instructions
Phy. Register File	128 Int/ 128 FP	*Int. FUs*	3 ALU, 1 Mul/Div, 2 ld/st
Issue/Commit Width	6 instructions	*FP FUs*	3 ALU, 1 Mul/Div
Branch Predictor	bi-modal 4K entries	*BTB Size*	2048 entries, 2-way assoc.
L1 - I-cache	32K, direct-map, 2 cycle latency	*L1 - D-cache*	32K, 4-way assoc., 2 cycle latency
Memory Latency	40 cycles first chunk 1 cycles/inter-chunk	*L2 - cache*	unified 512K, 8-way assoc., 6 cycles

Table 1. Default Parameters for the Experimental Evaluation

4.2 IPC Results and Analysis

We use the calculations in [6] to compare the forwarding latencies of *FUL0* and *SFU* bypass networks (in Figure 5), and found that the forwarding latency of *SFU* is about 70% less than that of *FUL0*. Figure 8 gives the IPCs for the SPEC 2K INT and FP benchmarks. As seen in Figure 8, the IPCs from *LSFU* and *FUL0* are very close to each other, but significantly higher than that from *FUL1*. However, *ESFU* with minimal bypass hardware incurs more IPC impact than *LSFU* because more instructions get delayed due to extremely limited bypassing. However, for many of the benchmarks, *ESFU* is still quite close to or even higher than *FUL1*. *LSFU* performs better than *FUL1*, because all instructions incur a 1 cycle delay in *FUL1*, whereas in *LSFU*, most of the instructions do not incur any delays and only a few instructions suffer a delay of 2 or more cycles because of getting re-scheduled. In case of *ESFU* network, on the other hand, if an instruction's operands are not available from the bypass network, then the instruction has to wait till they are available from the register file.

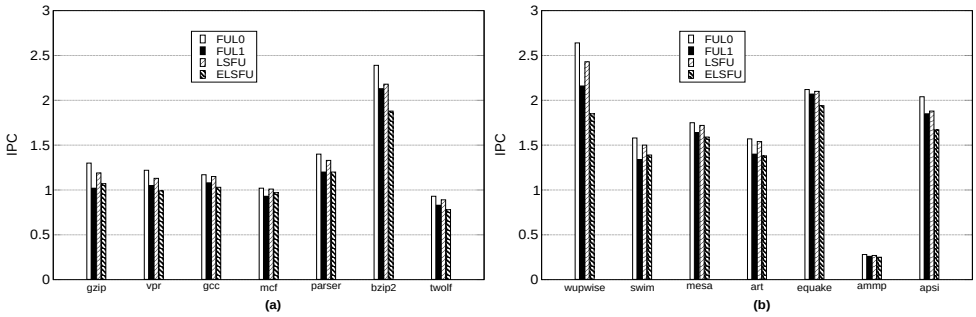

Fig. 8. IPCs for SPEC 2000 (a) INT and (b) FP Benchmarks

4.3 Lower Priority to Branch Instructions

One of the main reasons for the performance impact with *SFU* is the delayed execution of some of the instructions. To recover the performance loss, we investigate a technique that gives lower priority to branch instructions. Only the mispredicted branches affect performance. Branch instructions, on the other hand, compete with other instructions for the valuable forwarding paths. For instance, if a branch instruction and a result-producing instruction are both dependent on the same instruction, then both the instructions will be assigned the FU used by the producer instruction. In this case, if the branch instruction gets the FU, the

result-producing instruction is delayed, and performance will be hit. However, if the branch instruction is delayed, then the performance is hit only if the branch instruction is mispredicted. To avoid this, branch instructions are given a lower priority during *FU arbitration*. For this, each instruction is assigned a bit (called the "type bit"), which indicates whether the instruction is a branch instruction or not, and in case of a collision for the same FU, lower priority branch instruction gets "un-scheduled". Figure 9 shows the IPC results with this technique. Figure 9 shows a significant improvement in IPC for many benchmarks, bringing the IPC of *LSFU* almost equal to that of *FUL0*.

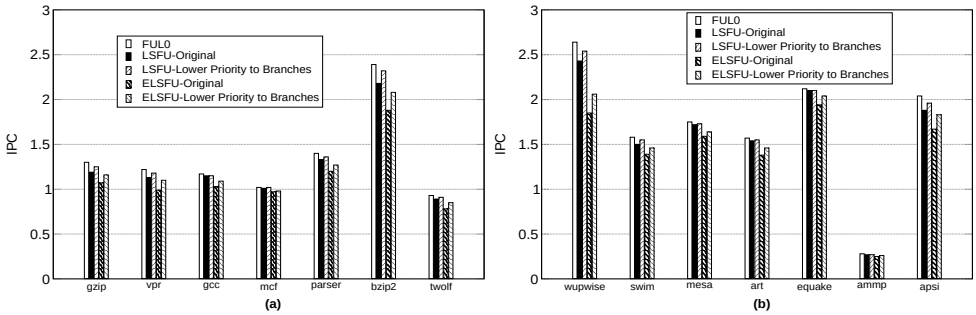

Fig. 9. IPCs for SPEC 2000 (a) INT and (b) FP Benchmarks

4.4 Single Input Single FU Bypass Network

The bypass network complexity can be further reduced by half by forwarding the values to only one of the inputs of the FUs [2]. Since, in SFU, only the instructions that have at most one operand forwarded from the bypass network are scheduled, having a single input forwarding is a natural extension of this design. This would require a switch in the operand locations in the instructions, so that the correct operand is bypassed the correct value. Since the values are forwarded to the inputs of the same FU in the SFU design, the switch can be performed once the operand that will be forwarded is known. No additional performance loss is observed for a Single Input SFU bypass network, because of the criterion used for scheduling instructions in *SFU*. However, with single input SFU bypass network, the forwarding latency is about 85% less than that with *FUL0*. This kind of data forwarding has one of the minimum bypass network hardware, apart from the case when there is no data forwarding.

5 Conclusions

Microprocessors usually use broadcast-based bypass networks to execute dependent instructions in consecutive cycles. However, for a wide-issue and deeply pipelined processor, broadcasting the results can take multiple cycles, especially with the wire delays increasing in the sub-micron technology era, reducing performance significantly. In this paper, we observed that the results of most of instructions are used by just one other instruction and the operands of many instructions come from a single other instruction. Based on this observation, we proposed a Single FU bypass network, where the results of an FU are only

bypassed to its own inputs, thus reducing the bypass network complexity significantly, and facilitating fast forwarding. Our studies showed that the forwarding latency can be reduced by more than 70%, while incurring a small IPC impact of about 2% for most of the benchmarks. Single FU bypass networks are also much more scalable than the broadcast-based bypass networks, as the future microprocessors become more wide and more deeply pipelined.

References

1. V. Agarwal, M. S. Hrishikesh, S. W. Keckler and D. Burger, "lock rate versus IPC: the end of the road for conventional microarchitectures," *Proc. ISCA-27*, 2000.
2. P. Ahuja, D. Clark, and A. Rogers, "The performance impact of incomplete bypassing in processor pipelines," *Proc. Micro-28*, 1995.
3. E. Bloch, "The Engineering Design of the Stretch Computer," *Proc. Eastern Joint Computer Conference*, 1959.
4. M. Brown, J. Stark and Y. Patt, "Select-free Instruction Scheduling Logic," *Proc. Micro-34*, 2001.
5. J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers, 2002.
6. S. Palacharla, N. P. Jouppi and J. E. Smith, "Complexity-Effective Superscalar Processors," *Proc. ISCA*, 1997.
7. G. Hinton, et al, "A 0.18-um CMOS IA-32 Processor With a 4-GHz Integer Execution Unit," *IEEE Journal of Solid-State Circuits*, Vol. 36, No. 11, Nov. 2001.
8. K. Sankaralingam, V. Singh, S. Keckler and D. Burger, "Routed Inter-ALU Networks for ILP Scalability and Performance," *Proc. ICCD*, 2003.
9. E. Sprangle and D. Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines," *Proc. ISCA-29*, 2002.
10. J. Stark, M. Brown and Y. Patt, "On Pipelining Dynamic Instruction Scheduling Logic," *Proc. Micro-33*, 2000.
11. The National Technology Roadmap for Semiconductors, Semiconductor Industry Association, 2001.
12. E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors," *Proc. 30th International Symposium on Microarchitecture*, 1997.
13. D. Leibholz and R. Razdan, "The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor," *Proc. Compcon*, pp. 28-36, 1997.
14. K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," *Proc. Micro-30*, 1997.
15. R. Canal, J. M. Parcerisa, and A. Gonz?lez, "Dynamic Cluster Assignment Mechanisms," *Proc. HPCA-6*, 2000.
16. A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors," *Proc. Micro-33*, 2000.
17. J. M. Parcerisa, J. Sahuquillo, A. Gonzalez and J. Duato, "Efficient Interconnects for Clustered Microarchitectures," *Proc. PACT-11*, 2002.
18. R. Nagarajan, et al, "A design space evaluation of grid processor architectures," *Proc. Micro-34*, 2001.
19. E. Waingold, et al, "Baring it all to software: RAW machines," *IEEE Computer*, 30(9):86-93, September 1997.
20. M. Fillo, et al, "The M-Machine Multicomputer," *Proc. Micro-28*, 1995.
21. A. Aggarwal and M. Franklin, "Instruction Replication: Reducing Delays due to Inter-Communication Latency," *Proc. PACT*, 2003.
22. D. Burger and T. Austin, "The Simplescalar Tool Set," *Technical Report*, Computer Sciences Department, University of Wisconsin, June 1997.