Performance Evaluation of the PowerPC 620 Microarchitecture

Christopher P. Nelson

Department of Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh, PA 15213

Abstract

The PowerPC 620 superscalar microprocessor is the most recent and performance leading member of the PowerPC family, which is being jointly developed by IBM and Motorola. The 64-bit 620 represents the most aggressive microarchitecture for superscalar processors to date. It employs a two-level branch prediction scheme, dynamic renaming for all the register files, distributed multi-entry reservation stations, true out-of-order execution by six pipelined execution units, and a completion buffer for ensuring precise interrupts.

This paper presents an instruction-level performance evaluation of the PowerPC 620 microarchitecture. A performance simulator for the 620 is developed using the VMW (Visualization-based Microarchitecture Workbench) retargetable framework. The VMW-based simulator accurately models the 620 microarchitecture down to the machine cycle level. Extensive trace-driven simulation is performed using the SPEC92 benchmarks. The experimental results indicate that the 620 is a well balanced design and achieves a maximum IPC rating of 1.94 on one of the benchmarks. Detailed quantitative analyses of the effectiveness of all the key microarchitecture features are presented. A brief philosophical comparison with the Alpha AXP 21164 is also included.

Keywords: Superscalar processors, Out-of-order execution, Performance evaluation, Instruction-level parallelism.

Performance Evaluation of the PowerPC 620 Microarchitecture

Abstract

The PowerPC 620 superscalar microprocessor is the most recent and performance leading member of the PowerPC family, which is being jointly developed by IBM and Motorola. The 64-bit 620 represents the most aggressive microarchitecture for superscalar processors to date. It employs a two-level branch prediction scheme, dynamic renaming for all the register files, distributed multi-entry reservation stations, true out-of-order execution by six pipelined execution units, and a completion buffer for ensuring precise interrupts.

This paper presents an instruction-level performance evaluation of the PowerPC 620 microarchitecture. A performance simulator for the 620 is developed using the VMW (Visualization-based Microarchitecture Workbench) retargetable framework. The VMW-based simulator accurately models the 620 microarchitecture down to the machine cycle level. Extensive trace-driven simulation is performed using the SPEC92 benchmarks. The experimental results indicate that the 620 is a well balanced design and achieves a maximum IPC rating of 1.94 on one of the benchmarks. Detailed quantitative analyses of the effectiveness of all the key microarchitecture features are presented. A brief philosophical comparison with the Alpha AXP 21164 is also included.

1 Introduction

The latest announcement by the IBM-Motorola-Apple alliance is the PowerPC 620, the first 64-bit member and performance leader of the PowerPC family. While the latest Alpha AXP 21164 [8] (@300MHz) from DEC may edge out the 620 (@150MHz) as the industry performance leader, the 620 employs the most aggressive microarchitecture and achieves the highest level of instruction-level parallelism of any microprocessor currently on the market. The 620 is the first 64-bit superscalar microprocessor to employ true out-of-order execution, aggressive branch prediction, distributed multi-entry reservation stations, dynamic renaming for all register files, six pipelined execution units, and a completion buffer to ensure precise interrupts. Most of these features have not been previously implemented in a single-chip microprocessor. Their actual effectiveness is of great interest to both academic researchers as well as industry designers. This paper presents an instruction-level, or machine-cycle level, performance evaluation of the PowerPC 620 microarchitecture.

Section 2 presents an overview of the PowerPC architecture and the details of the 620 microarchitecture. Section 3 describes the software tools and the framework used in our experimental evaluation. Sections 4 through 7 present the experimental results that characterize the effectiveness of the 620 microarchitecture in instruction fetching, instruction dispatching, instruction execution, and instruction completion, respectively. Section 8 contains some concluding remarks and a brief philosophical comparison between the PowerPC 620 and the Alpha AXP 21164.

2 The PowerPC 620

2.1 The PowerPC Architecture

The PowerPC architecture [18] is the result of the PowerPC alliance among Apple, IBM, and Motorola. It is based on IBM's RS/6000 POWER architecture [17], designed to facilitate parallel instruction execution and be able to scale well with advancing technology. Motorola and IBM are the chief designers of the PowerPC architecture and the family of PowerPC chips, while Apple is focussing on PowerPC systems and software.

As of this writing, the PowerPC alliance has released three chips and announced a fourth. The first, which provided a transition from the POWER architecture to PowerPC, is the PowerPC 601 [16]. The second, low-power chip, is the PowerPC 603 [13]. Recently, a more advanced chip for desktop systems has begun production, the PowerPC 604 [14]. The fourth chip, the last one that will be designed by the alliance, is a high-performance chip. This chip, the PowerPC 620 [15], was only recently announced.

The PowerPC architecture contains 32 integer registers (GPRs) and 32 floating point registers (FPRs). It also contains 32 condition register bits which can be addressed as one 32-bit register (CR), as a register file with 8 four-bit fields (CRFs), or as 32 single-bit fields (CRBs). It also contains a count register (CTR) and a link register (LR), both primarily used for branch instructions, and an integer exception register (XER) and a floating point status and control register (FPSCR), which is used to control the operation and record the exception status of the appropriate instruction types. PowerPC instructions are typical RISC, with the addition of floating point multiply-add fused (FMA) instructions and instructions to set, manipulate, and branch off of the condition register bits.

The most dominant architecture today, in terms of installed base, is the Intel x86. However, the PowerPC appears to be making a serious challenge to this dominance. IBM is looking to the PowerPC as the new ISA for the entire company. Motorola has an agreement to supply Ford with PowerPC-based processors for future automotive electronics. Motorola is also planning on employing PowerPC processors in their products ranging from hand-held devices to multiprocessor servers. Future systems from Apple will employ PowerPC processors. In addition, Apple, IBM and Motorola have recently announced their agreement on developing common/compatible hardware platforms based on the PowerPC processors.

2.2 The PowerPC 620

The 620 is a 4-wide superscalar machine. It uses aggressive branch prediction [7] to fetch instructions as soon as possible and a generalized dispatch scheme to get those instructions to the execution units.

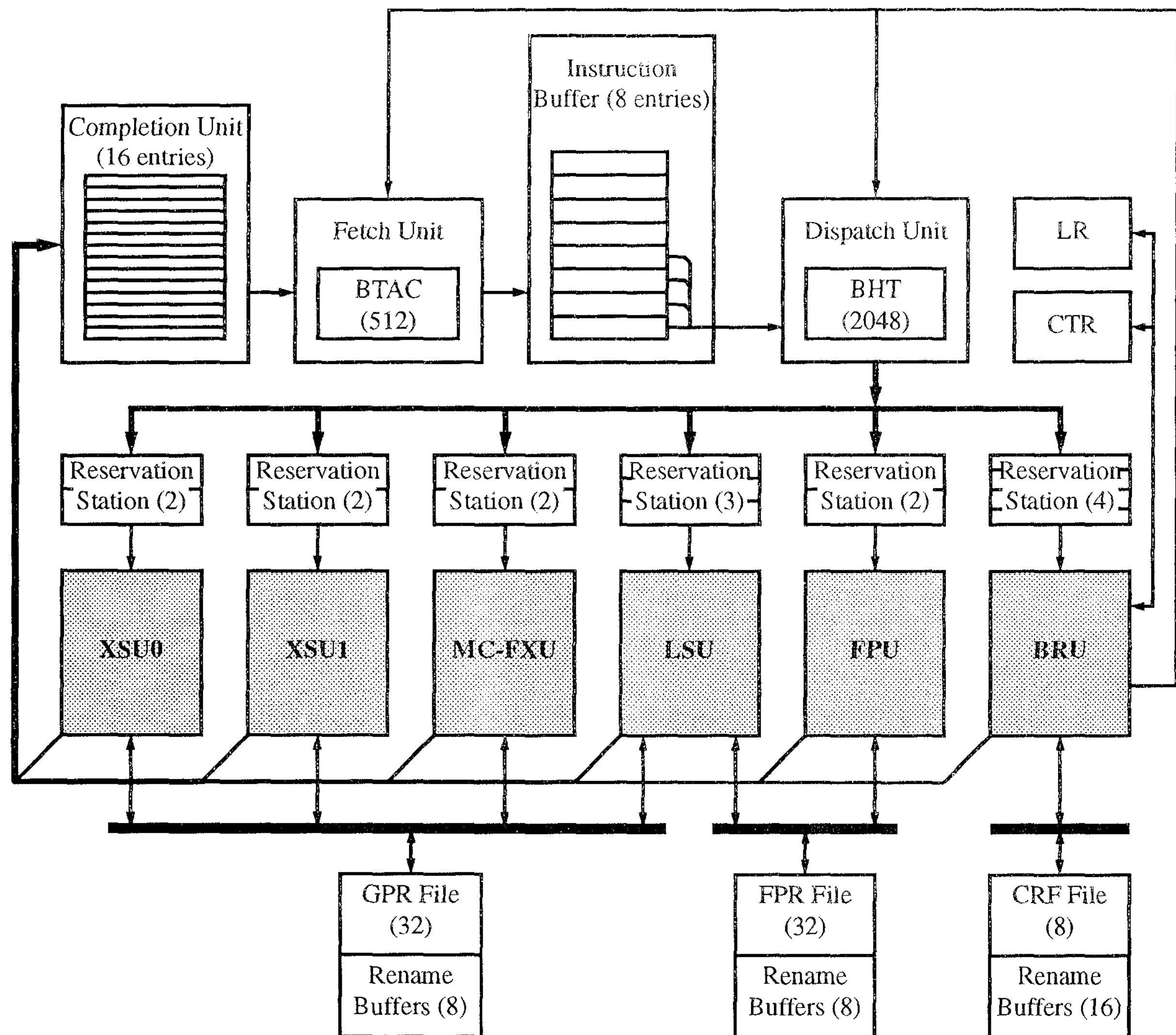

Figure 1: PowerPC 620 Microarchitecture Diagram

The PowerPC 620 uses six parallel execution units. It has two simple (single-cycle) integer units, one complex (multi-cycle) integer unit, one floating-point unit (3 stages), one load/store unit (2 stages), and a branch unit. The branch unit accepts condition register logical instructions as well as branches. To keep these execution units as full as possible, the 620 uses distributed reservation stations [9] and register renaming to implement an aggressive out-of-order execution scheme.

The 620 processes instructions in five major stages, some of which are separated by buffers to take up slack in dynamic variations of available parallelism. Major stages can require multiple cycles, while minor stages are one cycle each. The pipeline stages are **Fetch**, **Dispatch**, **Execute**, **Complete**, and **Writeback**. The first three stages are followed by the Instruction Buffer, Reservation Stations, and the Completion Buffer, respectively. See Figure 2.

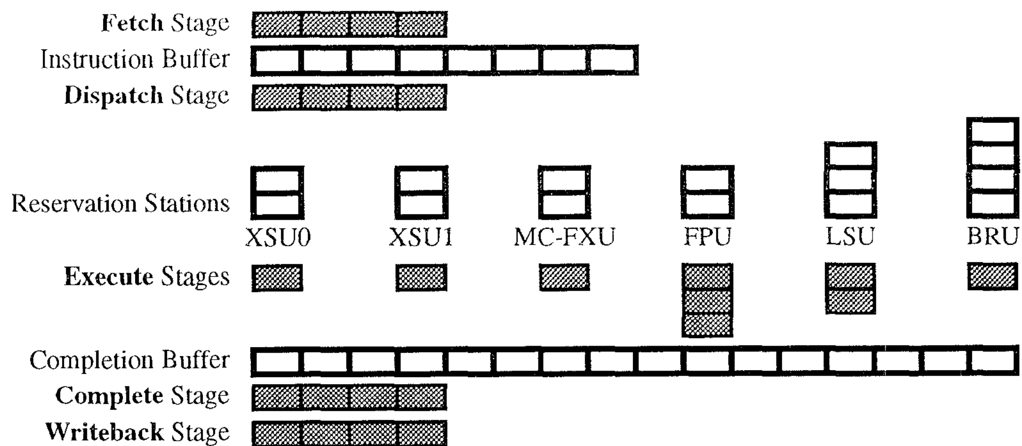

Figure 2: PowerPC 620 Instruction Pipeline

Fetch Stage: The fetch unit accesses the instruction cache to retrieve up to four instructions per cycle into the instruction buffer. The end of a cache line or a taken branch can prevent the fetch unit from getting four useful instructions in a cycle, and a mispredicted branch can waste fetch cycles while fetching down the wrong path. Also during the fetch stage a preliminary branch prediction is made via the Branch Target Address Cache (BTAC), and the predicted next address is used for fetching in the next cycle.

Instruction Buffer: The instruction buffer holds instructions between the fetch and dispatch stages. If the dispatch unit cannot keep up with the fetch unit, instructions will stay here for multiple cycles until the dispatch unit can get to them. A maximum of eight instructions can be buffered at a time. Instructions are buffered and shifted in groups of two to simplify the logic.

Dispatch Stage: The dispatch unit decodes instructions in the instruction buffer and checks if they can be dispatched to reservation stations. If all dispatch conditions are fulfilled for an instruction, the dispatch stage will allocate a reservation station entry, a completion buffer entry, and any needed destination rename registers for it. If these resources are available and each instruction goes to a different execution unit and all special serialization constraints are met, up to four instructions may be dispatched in order per cycle.

There are eight general-purpose register rename buffers, eight floating-point register rename buffers, and sixteen condition register field rename buffers. During dispatch, the necessary number of these buffers is allocated for the results of the instruction. Also during dispatch, any source operands which have been renamed by previous instructions are marked with the tags of the associated rename buffers. The reservation stations will then watch the appropriate result buses for forwarded results.

If a branch is being dispatched, resolution of the branch is attempted immediately. If the branch depends on an operand that is not yet available, it is predicted via the Branch History Table (BHT). If the predicted or resolved address does not match that made by the BTAC during the fetch stage, the speculatively fetched instructions are canceled and fetching restarts at the new address.

Reservation Stations: Each execution unit contains a reservation station to hold those instructions waiting to execute there. Each reservation station can hold two to four instruction entries, depending on the execution unit. Each dispatched instruction waits in a reservation station until all of its source operands have been read or forwarded and the function unit is ready for execution. An instruction may then leave the reservation station and enter its function unit out of order. Each execution unit contains one reservation station and one function unit.

Execute Stage: A function unit computes the results of an instruction. This major stage can require multiple minor stages (cycles) to produce its results, depending on the type of instruction being executed. At the end of execution, the instruction results are sent to the destination rename buffers and forwarded to any waiting instructions, and the instruction is marked finished in the completion buffer.

Completion Buffer: The completion buffer holds the states of the in-flight instructions until they are architecturally complete. The completion buffer has sixteen entries with which to hold instruction information. An entry is allocated for each instruction during the dispatch stage. The execute stage then marks it as finished when the unit is done executing the instruction. Once an instruction is finished, it is eligible for completion.

Complete Stage: During the completion stage, finished instructions are removed from the completion buffer in order, up to four at a time, and passed to the writeback stage. Fewer instructions will complete in a cycle if there are an insufficient number of write ports or if fewer than four instructions are finished and ready to complete in order. By holding instructions in the completion buffer until writeback, the 620 guarantees that the architected registers hold the correct state up to the most recently completed instruction, thus allowing precise interrupts even with aggressive out-of-order techniques [3].

Writeback Stage: During this stage, the writeback logic moves the results of those instructions completed by the completion unit in the last cycle from the rename buffers to the architected register files.

3 Experimental Framework

To analyze the PowerPC 620 microarchitecture, we used the Visualization-based Microarchitecture Workbench (VMW) [20], a retargetable performance simulator. The targeting of VMW for a specific processor

is performed by specifying a set of machine description files. Once the files are specified, VMW provides the generic simulation and visualization engines and automatically compiles the files into a performance simulator for the specified machine.

3.1 Trace Generation

Each instruction trace is a record of the dynamic path that a particular execution of a program took through that program's static code. This record depends solely on the machine architecture, not on its implementation or timing. Instruction traces are generated on an existing PowerPC 601 machine via software instrumentation.

For this paper, traces for eight SPEC 92 benchmarks are used, four integer and four floating point. Small data sizes are used. The benchmarks and their dynamic instruction mixes are given below.

SPEC 92 Benchmarks	Integer Benchmarks (SPECint 92)				Floating-Point Benchmarks (SPECfp 92)			
	026. compress	*023. eqntott*	*008. espresso*	*022. li*	*052. alvinn*	*094. fpppp*	*090. hydro2d*	*047. tomcatv*
Integer Arithmetic (single cycle)	41.4%	48.6%	47.7%	29.5%	37.5%	4.6%	26.2%	19.9%
Integer Arithmetic (multi-cycle)	0.9%	1.5%	1.4%	5.1%	0.3%	0.3%	1.2%	0.0%
Integer Load	26.1%	23.4%	24.2%	28.5%	0.3%	3.0%	0.5%	0.3%
Integer Store	17.9%	6.9%	8.8%	18.6%	0.2%	0.0%	0.2%	0.2%
Floating-Point Load	0.0%	0.0%	0.0%	0.0%	26.8%	33.3%	22.5%	27.8%
FP Store	0.0%	0.0%	0.0%	0.0%	12.0%	18.0%	7.7%	9.1%
FP Arithmetic (pipelined)	0.0%	0.0%	0.0%	0.0%	12.2%	38.4%	27.0%	37.8%
FP Arithmetic (non-pipelined)	0.0%	0.0%	0.0%	0.0%	0.1%	0.3%	1.9%	0.7%
Branch	13.7%	19.6%	17.9%	18.2%	10.6%	2.1%	12.8%	3.9%
Total Instructions	1,652,555	1,815,098	1,506,871	3,375,006	2,920,877	5,520,737	4,114,602	6,858,618

Table 1: Dynamic Instruction Mixes for All Benchmarks

Most of the integer benchmarks are fairly similar to each other in terms of instruction mix, though *li* contains more instructions that use the multi-cycle integer unit. Most of these instructions move values to and from special purpose registers.

The floating-point benchmarks differ not only in their use of floating-point instructions, but they also contain significantly fewer branches, especially *fpppp* and *tomcatv*. *Fpppp* is most floating point intensive, as it uses few integer instructions. Finally, note that *hydro2d* uses more non-pipelined floating-point instructions. Although the number is small, these instructions are all FP divides, which take 35 cycles each to execute in the PowerPC 620.

3.2 Machine Specification

VMW is configurable to a specific machine implementation through four template files and a fifth C++ behavior file. Two of the template files specify the syntax and semantics of the architected instructions, while the other two specify the organization and timing of the microarchitecture. The C++ behavior file uses the specifications in the template files along with special-case routines that cannot be coded in the template files to model the execution of each instruction through the machine.

To compare the microarchitectural complexity of the PowerPC 620 with that of various other machines, we show a comparison with other simulators targeted using VMW.

Microprocessor	Alpha 21064	RS/6000	PowerPC 601	PowerPC 620
Instruction Syntax (Templates)	487	405	383	383
Instruction Semantics (Templates)	37	125	205	205
Instruction Timing (Templates)	24	87	112	113
Machine Organization (Templates)	39	60	51	75
Machine Behavior (Lines of Code)	668	1132	1,086	3,093

Table 2: Comparison of Sizes of Machine Description Files

The five machine description files for the PowerPC 620 are generated based on internal confidential design documents [15] provided and periodically updated by the PowerPC 620 design team. Correct interpretation of the design documents is checked by a member of the design team through a series of refinement cycles as the 620 design is finalized. Through this process, even late changes to the 620 design were incorporated into our machine description files and modeled by our performance simulator. Validation of the simulation model includes syntactic checking of the description files using tools provided by VMW, and timing checking by comparing timing results on small benchmarks with lower-level hardware models.

3.3 Trace-Driven Simulation

Trace-driven, or performance, simulation is much faster than functional simulation. However, with performance simulation, instructions with variable latency such as integer multiply/divide and floating-point

divide cannot be simulated accurately. For these, we assumed the minimum latency. The frequency of these operations and the amount of variance in the latencies are both quite low.

Furthermore, the trace only contains those instructions that are actually executed. For this reason, no speculative instructions that are later discarded can be included in the performance analysis. This is helpful because it prevents those useless instructions from showing up and influencing the design analysis. Certainly, speeding up the processing of instructions that will ultimately be cancelled will not increase performance any. In this paper, I-cache and D-cache activity are not simulated. By assuming perfect caches, simulation time is reduced, and the results are more indicative of the core CPU performance and not a function of the external cache size or memory speed. Library code linked into the benchmarks was not included in the instruction traces.

To collect the data on all the benchmarks, we ran our 620 simulator on a variety of workstations, including DECstation 3100s, SparcStation 5's & 20's, and an HP 715. These workstations could simulate the traces at a rate of between 750-3000 simulated cycles per minute, with a total simulation time of about 170 hours for all the benchmarks.

Table 3 presents the total number of instructions simulated for each benchmark and the total number of 620 machine cycles required. The IPC achieved by the 620 for each benchmark is also shown. The IPC rating provides an overall assessment of the effectiveness of the 620 microarchitecture, the detailed analysis of which is presented in Sections 4 through 7.

Benchmarks	*compress*	*eqntott*	*espresso*	*li*	*alvinn*	*fpppp*	*hydro2d*	*tomcatv*
Dynamic Instructions	1,652,555	1,815,098	1,506,871	3,375,006	2,920,877	5,520,737	4,114,602	6,858,618
Cycles to Execute	987,560	1,124,107	1,002,956	2,363,456	1,507,978	4,239,087	4,211,441	5,120,970
IPC	1.67	1.61	1.50	1.43	1.94	1.30	0.98	1.34

Table 3: Summary of Benchmark Performances

It is surprising to see a lower IPC for the floating-point benchmarks than for the integer benchmarks, as the more even distribution of instruction types ought to help boost performance. Although the poor performance of *hydro2d* can be anticipated due to the large number of floating-point divides it contains, a closer inspection is necessary to find the sources of bottlenecks in the other benchmarks.

4 PowerPC 620 Instruction Fetching

Provided that there is enough room in the instruction buffer, the 620's fetch unit is capable of fetching four instructions every cycle. That is, up until it encounters a branch. If the fetch unit were to wait for branch

resolution before continuing to fetch non-speculatively, or if it were to bias naively for branch not-taken, machine execution would be drastically slowed by the bottleneck in fetching down taken branches. Because of this, accurate branch prediction in a superscalar machine as wide as the PowerPC 620 is extremely important.

To demonstrate the branch prediction and speculation characteristics of the 620, we chose one of the integer benchmarks as an example case. We chose *espresso* because it has the most interesting characteristics, but summary information on all benchmarks can be found at the end of this section.

4.1 Branch Prediction

Branch prediction takes place in two phases. The first prediction, done during the fetch stage via the BTAC, provides a preliminary target guess given very little information. The second, more accurate, prediction is done during the dispatch stage via the BHT, using additional information available after instruction decoding. We first show the distribution of architecture-related branch characteristics in the traces, then examine the implementation-dependent branch prediction statistics.

4.1.1 Branch statistics

Branch instructions are separated into four categories based on mnemonic: branch (b), branch conditional (bc), branch conditional to count register (bcctr), and branch conditional to link register (bclr). The bc branches are by far the most common, and their resolutions are fairly evenly split between not taken and taken. Table 4 shows a sample breakdown for the *espresso* benchmark as percentages of all branches.

Branch Type	Not Taken	Taken	Total
b	0.0%	9.4%	9.4%
bc	37.6%	45.8%	83.4%
bcctr	0.0%	0.9%	0.9%
bclr	0.0%	6.3%	6.3%
Total	37.6%	62.4%	100.0%

Table 4: Branch Resolution for Espresso

4.1.2 BHT effectiveness

During branch dispatch, the 620 attempts to immediately resolve the branch based on available information. If the branch is unconditional, or if the condition register has the appropriate bits ready, then no branch prediction is necessary; the branch is effectively executed immediately, without speculation. If, on the other hand, the source condition bits are unavailable (the instruction creating them is still executing),

then the branch is predicted. The Branch History Table contains two history bits per entry that are accessed during branch dispatch to predict whether the branch will be taken or untaken. Upon resolution of the predicted branch, the BHT is updated according to the actual direction of the branch.

The 2048-entry BHT is not accessed as a cache, but as a simple table look up. There is no concept of a hit or miss. If two branches that update the BHT are an exact multiple of 2048 instructions apart, i.e. aliased, they will alter each other's predictions.

The following table shows the relative frequency of immediate resolution and the two outcomes of predicted branches. Note that unconditional branches never need to be predicted. Note also that the percent of predicted branches that are predicted correctly is about 85%. Adding in the immediately resolved branches, you find that 88% of all branches in *espresso* can get a correct fetch address at dispatch without waiting until resolution. These percentages are even higher for the other benchmarks; see Table 8

Branch Type	Immediately Resolved	Predicted Correctly	Predicted Incorrectly
b	9.4%	0.0%	0.0%
bc	2.7%	69.0%	11.7%
bcctr	0.9%	0.0%	0.0%
bclr	5.6%	0.6%	0.0%
Total	18.6%	69.6%	11.8%

Table 5: BHT Effectiveness in Espresso

4.1.3 BTAC effectiveness

The 620 can resolve or predict the branch at the dispatch stage, but even that will incur at least a cycle of delay until the new target of the branch can be fetched. For that reason, the 620 makes a preliminary prediction during the fetch stage, based solely on the addresses of the instructions it is currently fetching. If one of these addresses hits in the Branch Target Address Cache, the target address stored in the BTAC is used as the fetch address in the next cycle. The BTAC is smaller than the BHT, with 512 entries, but it only needs to hold the targets of those branches that are predicted taken. Branches that are predicted not taken (fall through) are not included in the BTAC.

Unconditional and PC-relative conditional branches use the BTAC. Branches to the count register or the link register have unpredictable target addresses, so are never put in the BTAC. This has the effect of causing these branches to always be predicted not taken. This lowers the BTAC's prediction accuracy, but is largely unavoidable.

There are four possible branch prediction cases in the BTAC: a BTAC miss for which the branch is not taken (correct), a BTAC miss for which the branch is taken (incorrect), a BTAC hit for a taken branch (correct), and a BTAC hit for an untaken branch (incorrect). Note that the BTAC can never hit on a taken branch but get the wrong target address, as only PC-relative branches can hit in the BTAC and therefore must always use the same target address.

A verbose table of BTAC behavior is given below for the *espresso* benchmark.

Branch Type	BTAC miss (predict not taken)		BTAC hit (predict taken)	
	Correct	Incorrect	Correct	Incorrect
b	0.0%	0.8%	8.7%	0.0%
bc	31.1%	5.7%	40.1%	6.5%
bcctr	0.0%	0.9%	0.0%[1]	0.0%[1]
bclr	0.0%	6.3%	0.0%[1]	0.0%[1]
Total	31.1%	13.6%	48.8%	6.5%

Table 6: BTAC Effectiveness in Espresso

1. Bcctr and bclr can never hit in the BTAC.

4.1.4 BHT and BTAC interaction

Branch prediction is done twice for each branch, once by the BTAC and again later by the BHT. If the BHT prediction disagrees with the BTAC prediction, the BTAC prediction is thrown away and the BHT prediction is used. However, if the predictions agree and are correct, then there is no interruption in fetching whatsoever. In combining the possible predictions and resolutions of the BHT and BTAC, there are six major categories. The frequency of these cases are shown in Table 7.

BHT prediction	BTAC prediction		Total
	Correct	Incorrect	
Resolved Immediately	9.2%	9.4%	18.6%
Predicted Correctly	68.3%	1.3%	69.6%
Predicted Incorrectly	0.6%	11.2%	11.8%
Total	78.1%	21.9%	100.0%

Table 7: Branch Prediction Interaction in Espresso

Notice that for this example benchmark, few of these branches can be resolved in the dispatch stage. Of those that are guessed, however, 85% are guessed correctly by each of the BTAC and the BHT. Notice also that although the BHT attempts to keep the BTAC correlated, there are a very few cases where they make

different predictions. A few of those are even cases in which the BTAC predicts correctly, but the BHT predicts incorrectly and throws away the BTAC prediction! This is quite rare, however.

The question could be asked, why have the BHT at all if the BTAC gets almost equivalent prediction accuracy? The answer is that it is the responsibility of the BHT to update the BTAC with what to predict next. Without the BHT to save history, the BTAC's prediction accuracy would probably be much lower. Another question is why aren't the two merged, so that the BTAC contains multiple bits of branch history and can make accurate predictions on its own?

4.2 Speculation

The PowerPC 620 predicts branch outcomes in order to keep its pipeline full. This means that the 620 must keep track of those instructions speculatively fetched and executed and be able to cancel them if a prediction turns out to have been made incorrectly. The 620 has enough hardware to speculate past four predicted branches before stalling the fifth at the dispatch stage. We analyze what performance increase is gained from this.

4.2.1 Distribution of number of branches bypassed

Although none of the FP benchmarks bypass many branches, most of the integer benchmarks do. Figure 3 shows the distribution of the number of branches bypassed in the *espresso* benchmark. Notice that the 620 is speculatively executing beyond one or more branches in over 50% of the cycles.

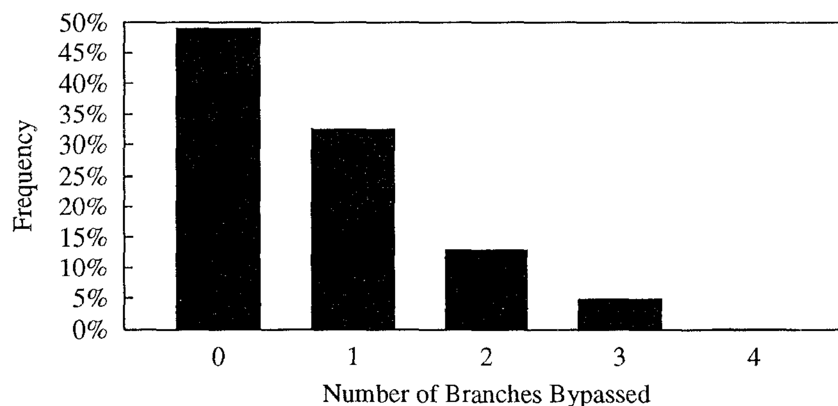

Figure 3: Distribution of Number of Branches Bypassed in Espresso

Also note the rarity of 4 bypassed branches. What is not shown is the frequency of use of the fourth branch reservation station entry for non-speculative (immediately resolved) branches or condition register logical instructions. Although the frequency of branch RS saturation is low, decreasing the number of reservation station entries to three or fewer would probably be a bad idea.

4.2.2 Total number of fetch penalty cycles due to misprediction by the BHT

The number of instructions that the BHT mispredicts and the average length of time required to resolve and correct these mispredictions determine how many fetch cycles are wasted speculating down the wrong path. Despite the 620's branch accuracy, it still loses 137,424 fetch cycles in the *espresso* benchmark due to branch mispredictions. This is 13.7% of the total fetch cycles. Because an accurate stream of fetching is so vitally important to a wide superscalar machine, branch prediction is obviously still an area that has room for improvement.

4.3 Instruction Fetch Summary

The following table summarizes the branch prediction statistics for all eight benchmarks. Most percentages are out of the total number of branches in the benchmark. The percentage of fetch cycles lost to misprediction is out of the total number of cycles in the simulated trace.

Benchmarks		*compress*	*eqntott*	*espresso*	*li*	*alvinn*	*fpppp*	*hydro2d*	*tomcatv*
Branch Resolution	Not Taken	40.0%	29.4%	37.6%	32.7%	5.3%	42.4%	16.8%	5.8%
	Taken	60.0%	70.6%	62.4%	67.3%	94.7%	57.6%	83.2%	94.2%
BHT prediction	Immediately Resolved	22.6%	19.4%	18.6%	27.2%	8.1%	61.0%	27.7%	45.4%
	Correct	66.9%	72.0%	69.6%	64.1%	90.9%	37.1%	66.7%	52.6%
	Incorrect	10.5%	8.6%	11.8%	8.7%	0.9%	1.8%	5.6%	2.0%
BTAC prediction	Correct	83.5%	83.0%	79.9%	74.7%	94.5%	87.9%	88.4%	93.3%
	Incorrect	16.5%	17.0%	20.1%	25.3%	5.5%	12.1%	11.6%	6.7%
Average Number of Branches Bypassed		0.539	0.802	0.744	0.804	0.197	0.087	0.501	0.174
Fetch Cycles Lost to Misprediction		7.8%	11.5%	13.7%	13.2%	0.4%	0.5%	5.0%	0.8%

Table 8: Branch Prediction Summary

The floating-point benchmarks tend to have fewer branch instructions, and those branches tend to be taken much more often than in the integer benchmarks. For this reason, the BHT and BTAC can predict much more successfully for the FP benchmarks and lose fewer cycles to misprediction.

5 PowerPC 620 Instruction Dispatching

The PowerPC 620 dispatches instructions in order. This causes the dispatching unit to be a potential bottleneck. To begin with, however, the dispatch unit must be suitably supplied with instructions by the fetch unit. The rest of the examples in Sections 5 through 7 are from the *alvinn* benchmark in order to display the use of the floating-point functional units. Summary information for all the benchmarks are shown at the end of each section.

5.1 Fetching Bandwidth

For maximum effectiveness, the fetch unit should make four instructions available to the dispatch unit at all times. As can be seen in Figure 3, four or more instructions are in the instruction buffer about 85% of

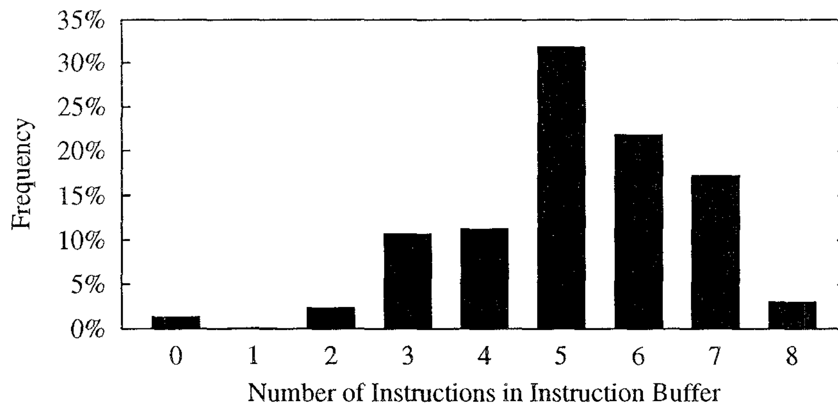

Figure 4: Distribution of Utilization of Instruction Buffer in Alvinn

the time for *alvinn*. What is available in the instruction buffer is not necessarily what is available to the dispatch unit, however. The dispatch buffer subset contains only the first four entries of the instruction buffer. Figure 3 shows that only about 55% of the time are there four instructions available in the dispatch buffer.

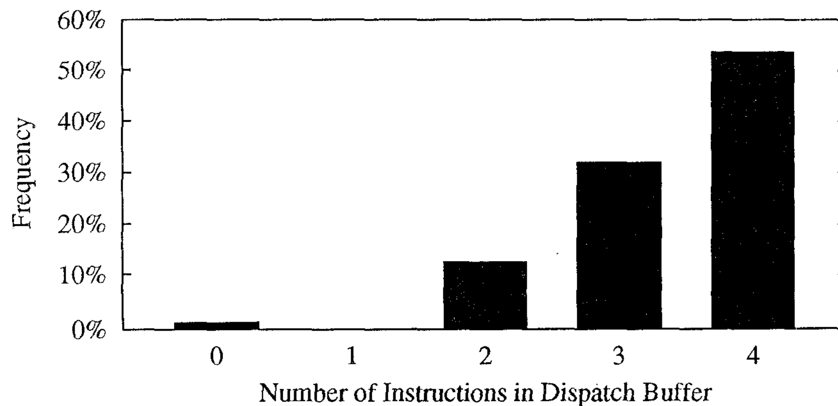

Figure 5: Distribution of Utilization of Dispatch Buffer in Alvinn

The reason for the discrepancy is that the instruction buffer is arranged in groups of two. These groupings allow holes to appear in the dispatch buffer, thus introducing inefficiency.

5.2 Dispatch Stalls

The 620 checks the entire dispatch buffer in parallel for any condition that could cause dispatch to stall for a particular instruction, and thus for all subsequent instructions due to in-order dispatching. The 620 simulator checks the dispatcher in order, one hazard at a time. As such, it will identify exactly zero or one hazard types as stalling dispatch in each cycle. The order of hazard checking and the frequency of occurrence are shown in the following table. Each hazard is explained in detail in the following paragraphs.

Dispatch Hazard Types	Frequency
Serialization constraints	0.0%
Branch wait for mtspr (CTR)	0.0%
Register read port saturation	0.8%
Reservation station saturation	4.2%
Rename buffer saturation	1.2%
Completion buffer saturation	9.3%
Already dispatched to unit	51.8%
No hazards	32.6%

Table 9: Frequency of Dispatch Hazards in Alvinn

Serialization constraints: Certain instructions cause "Single Instruction Serialization." That is when all previously dispatched instructions must complete before the serializing instruction can begin execution, and all following instructions must wait until it is finished before they can dispatch. This condition, though extremely disruptive to performance, is fortunately quite rare.

Branch wait for mtspr (CTR): Because branch instructions access the count register during the dispatch phase, a Move To Special Purpose Register (mtspr) instruction that writes to the count register will cause subsequent branch instructions to delay dispatching until it is finished. This condition is also rare.

Register read port saturation: The number of read ports of the GPR and FPR register files, though large, is finite. There are seven GPR and four FPR read ports available. Occasionally, they can be exhausted during dispatch. Note that there are enough condition register field read ports (three) that those are never exhausted.

Reservation station saturation: As instructions are dispatched, they are placed into reservation stations, where they are held until they can execute. There is one reservation station per execution unit, each of which contains multiple entries. The table below shows the maximum number of instructions that can be

held in each unit's reservation station, and the average utilization of those entries. Although the utilization of each RS seems low, note that the number of reservation station entries for each unit cannot be reduced below two without effectively disabling out-of-order execution for that unit. Also, by having more reservation station entries than each unit needs, the level of reservation station saturation is kept low, even given the "bursty" nature of instruction types. It is interesting that the two execution units which get some degree of saturation in this benchmark are the FPU and LSU. Each of these units had its number of reservation station entries decreased in the final stages of design.

Units	XSU0	XSU1	MC-FXU	FPU	LSU	BRU
Number of RS Entries	2	2	2	2	3	4
Average Number of RS Entries Utilized	0.53	0.26	0.01	0.80	1.04	0.21
Frequency of RS Saturation	0.0%	0.2%	0.0%	1.5%	2.5%	0.0%

Table 10: Reservation Station Utilization and Saturation in Alvinn

Rename buffer saturation: As each instruction is dispatched, any registers it uses as destinations are renamed into the appropriate rename buffer files. There are three rename buffer files, for general purpose registers, floating-point registers, and condition register fields. Because they are a centralized resource with relatively long-term usage, they can handle the bursty nature of instructions more effectively than the reservation stations. For this reason, their utilization can be a higher percentage of total without saturating often. For the condition register fields, however, a large number of rename buffers were allocated by the 620 design team. Although sixteen condition field rename buffers are hardly necessary, they are quite small, and the assurance of never running out of them can simplify other logic.

Register Types	GPR	FPR	CR
Number of Rename Buffers	8	8	16
Average Number of Rename Buffers Utilized	3.32	4.35	0.99
Frequency of Rename Buffer Saturation	0.4%	0.8%	0.0%

Table 11: Rename Buffer Utilization and Saturation in Alvinn

Completion buffer saturation: Completion buffer entries are also allocated from dispatch until write-back. Since the PowerPC 620 has sixteen completion buffer entries, no more than sixteen instructions may be in flight at once. Attempted dispatch beyond sixteen will stall.

A related statistic, the distribution of utilization of the completion buffer entries, is presented below for the *alvinn* benchmark. Note that for this benchmark there are almost always nine or more instructions in flight or waiting to write back.

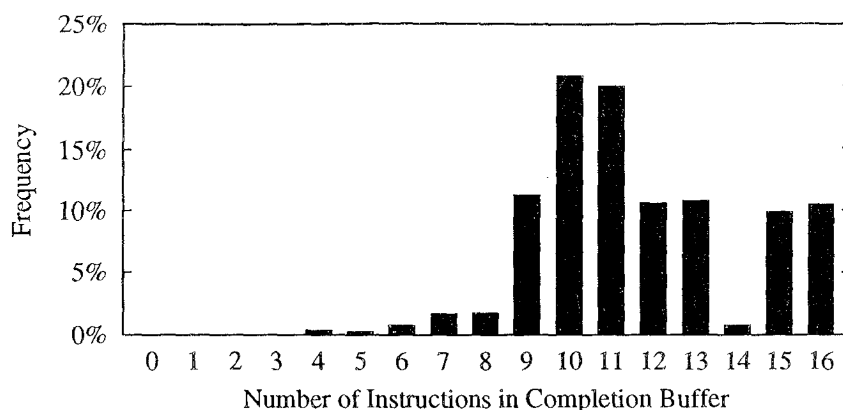

Figure 6: Distribution of Completion Buffer Utilization in Alvinn

Already dispatched to unit: Although each reservation station has multiple entries, each RS can still only receive one instruction per cycle. If an earlier instruction is being dispatched to a unit in a cycle, no more instructions of that type may dispatch until the next cycle even though multiple entries might be available in that RS. Even though this hazard is the last one checked by our simulated dispatch unit, it is recorded as causing by far the largest number of dispatch hazards, and perhaps unnecessarily so. This restriction is for logic simplicity only, avoiding multiple write ports to each RS, and is not required for program correctness. Without this restriction, many more instructions could enter the reservation stations in each cycle, and thus the chance of finding a large number of instructions to issue in subsequent cycles would also be increased.

5.3 Dispatching Bandwidth

As Figure 3 shows, *alvinn* does fairly well at avoiding crippling dispatch hazards, having very few cycles in which no instructions are dispatched. This is, in fact, a highly unusual distribution. Most traces saw a much higher instance of zero instructions dispatched. These traces also had a lower IPC, of course. The

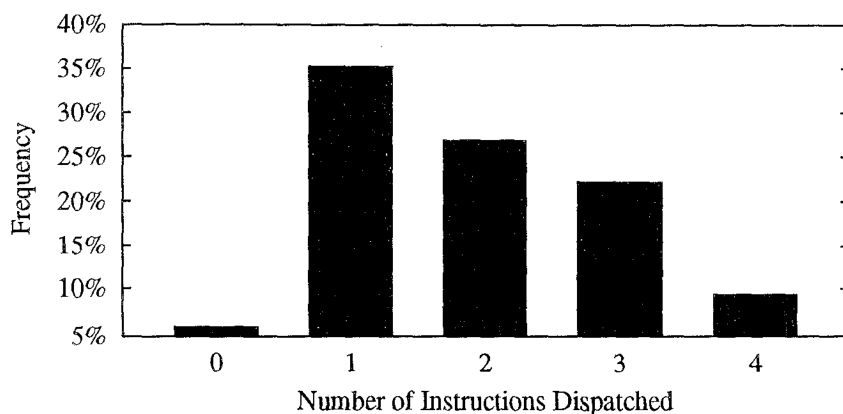

Figure 7: Distribution of Instruction Dispatch Parallelism in Alvinn

distribution from one to four tends to look about the same for all traces, however, as the 620 encounters fewer constraints when dispatching three instructions than when dispatching four, two are easier than three, and one is easier than two.

5.4 Instruction Dispatch Summary

Table 12 summarizes the dispatch resource usage for the eight benchmarks. Each number is the average number of entries or buffers that a benchmark uses per cycle. Distribution data for all benchmarks are obtained in our experiments. However, due to space limitation, only average values are shown here.

Benchmarks	*compress*	*eqntott*	*espresso*	*li*	*alvinn*	*fpppp*	*hydro2d*	*tomcatv*
Instruction Buffers	5.16	4.48	4.40	4.06	5.18	6.34	5.48	6.48
Dispatch Buffers	3.09	2.76	2.76	2.63	3.37	3.43	3.07	3.52
XSU0 RS Entries	0.51	0.76	0.78	0.53	0.53	0.05	0.23	0.14
XSU1 RS Entries	0.67	0.58	0.70	0.48	0.26	0.05	0.18	0.13
MC-FXU RS Entries	0.05	0.09	0.08	0.29	0.01	0.01	0.10	0.00
FPU RS Entries	0.00	0.00	0.00	0.00	0.80	1.04	1.23	0.69
LSU RS Entries	0.85	0.59	0.78	1.19	1.04	0.72	0.34	0.51
BRU RS Entries	0.61	0.90	0.83	0.93	0.21	0.10	0.54	0.20
GPR Rename Buffers	3.17	3.73	3.44	3.39	3.32	0.47	1.43	1.51
FPR Rename Buffers	0.00	0.00	0.00	0.00	4.35	3.21	3.13	3.35
CR Rename Buffers	0.79	1.19	1.12	1.03	0.99	0.18	1.01	0.52
Completion Buffers	7.21	7.48	7.39	7.93	11.61	7.50	8.21	9.44

Table 12: Summary of Average Dispatch Resource Usage

Table 13 summarizes the frequency of dispatch stalls for the eight benchmarks. The percentages are out of the total number of simulated cycles per benchmark. Any cycle in which the entire dispatch buffer can be emptied is marked as having no hazards.

Benchmarks	*compress*	*eqntott*	*espresso*	*li*	*alvinn*	*fpppp*	*hydro2d*	*tomcatv*
Serialization	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Move to CTR	0.0%	4.3%	1.2%	3.5%	0.0%	0.9%	0.9%	0.2%
Read port saturation	0.4%	0.0%	0.0%	0.0%	0.8%	16.9%	2.9%	9.4%
RS saturation	9.8%	13.8%	14.6%	8.6%	4.2%	12.0%	44.1%	3.7%
Rename buffer sat.	8.5%	2.3%	7.4%	2.9%	1.2%	18.5%	7.7%	28.3%
Completion buf. sat.	0.0%	0.0%	0.1%	1.1%	9.3%	0.4%	2.1%	1.7%
Already one to unit	46.1%	36.1%	37.8%	41.4%	51.8%	47.9%	25.0%	48.9%
No hazards	35.2%	43.5%	38.9%	42.6%	32.6%	3.4%	17.2%	7.8%

Table 13: Summary of Dispatch Stall Cycles

There are no big surprises on the integer side of the table, but some of the floating-point results are quite interesting. *Hydro2d* lived up to its instruction mix and stalled a very large number of instructions in the floating-point reservation station as it executed FP divide instructions. *Fpppp* and *tomcatv* exhibited a high degree of FP read port and rename buffer saturation. Although it is understandable for this to be the case for *fpppp*, as it has fewer integer instructions in its mix with which to balance its resource utilization, *tomcatv* is a bit more surprising. Since *tomcatv* does not have a high utilization of rename buffers, it can only be assumed that *tomcatv* gets its floating-point instructions in bursts, rather than interspersed with integer instructions, thus causing temporary resource hot spots. Finally, notice that the extremely disruptive dispatch serialization constraints are also extremely rare, which is good.

6 PowerPC 620 Instruction Execution

This section covers instruction flow from the reservation stations, through issuing to the function units, and within the execution stages. Because execution is a distributed process, inter-instruction constraints and major bottlenecks are far fewer here than they were in dispatch.

6.1 Issue Stalls

Once instructions have been dispatched to reservation stations, they usually must simply wait for their source operands to become available, then begin execution. There are a few other constraints, however. The full list of issuing hazards is described below.

Out of order disallowed: Although out-of-order execution is usually allowed from reservation stations, it is sometimes the case that certain instructions may not proceed past a prior instruction in the RS. This is usually the case in the branch unit, where all instructions must execute in order. It occasionally happens in the floating-point unit as well, if the prior instruction changes the FP execution mode.

Serialization constraints: Instructions which read or write non-renamed registers (such as the CTR or LR), or which read or write renamed registers in a non-renamed fashion (such as L/S multiple instructions), or which change or synchronize machine state (such as the eieio instruction, which enforces in-order execution of I/O) must often wait for all prior instructions to complete before executing. These instructions stall in the reservation stations until their serialization constraints are satisfied.

Waiting for source: The primary purpose of reservation stations is to hold instructions until all of their sources are ready. If an instruction requires a source that is not available, it must stall here until the data is forwarded to it.

Waiting for functional unit: Occasionally, two or more instructions will be ready to begin execution in the same cycle. In this case, the first will be issued, but the second must wait, as the functional unit is no longer available. This condition also applies when an instruction is executing in the MC-FXU (a non-pipelined unit) or when a floating-point divide instruction puts the FPU into non-pipelined mode.

Issue hazard frequencies: The following table shows the number of issue hazards of each type that occur per cycle in the *alvinn* benchmark. Because issuing is allowed to proceed out-of-order, each instruction in the reservation stations must checked for hazards. The complement of these numbers yields the IPC. For this benchmark it is 1.94.

Issue Hazard Type	Number
Out of order disallowed	0.001
Serialization constraints	0.000
Waiting for source	0.696
Waiting for functional unit	0.220

Table 14: Average Number of Issue Hazards per Cycle in Alvinn

By converting these numbers into percentages of total reservation station entry usage, we can get a more telling picture of issuing for all benchmarks in the following table. We also drop branch instructions from this table, as most of them have already been resolved or correctly predicted, so issue problems here will not hurt performance. And, since branches account for almost all occurrences of disallowed out-of-order issuing, that category has also been dropped from the table.

Benchmarks	*compress*	*eqntott*	*espresso*	*li*	*alvinn*	*fpppp*	*hydro2d*	*tomcatv*
Serialization	1.0%	1.6%	2.1%	6.0%	0.0%	0.3%	4.1%	0.0%
Waiting for Source	26.1%	30.9%	38.6%	36.3%	26.0%	24.5%	35.2%	9.6%
Waiting for FU	3.2%	3.3%	6.7%	10.8%	8.3%	6.7%	19.8%	3.6%
Issued	69.6%	64.3%	52.6%	46.9%	65.6%	68.5%	41.0%	86.8%

Table 15: Percentage of Stall Types for All Issuable Instructions

The frequency of issue stalls is reasonably consistent across all of the benchmarks, though once again *hydro2d* shows up as blocking a lot of instructions waiting for the FP function unit. Note also that the number of issue serialization stalls is roughly proportional to the number of multi-cycle integer instructions in the benchmark's instruction mix. This is because most of these multi-cycle instructions access the special purpose registers or the condition register as a non-renamed unit, which requires serialization.

6.2 Execution Units

Once an instruction is issued, it will execute to finish without further interruption. The following subsections show the propagation of instructions through issuing, execution, and finish.

6.2.1 Distribution of issuing parallelism

The following graph shows the distribution of issuing parallelism (the number of instructions issued per cycle). Though the issuing parallelism and the dispatch parallelism distributions must necessarily have the same average value, issuing is less centralized and has fewer constraints, and can therefore achieve a more consistent distribution. In most cycles, the number of issued instructions approximates the overall IPC, while dispatch has more extremes in its distribution.

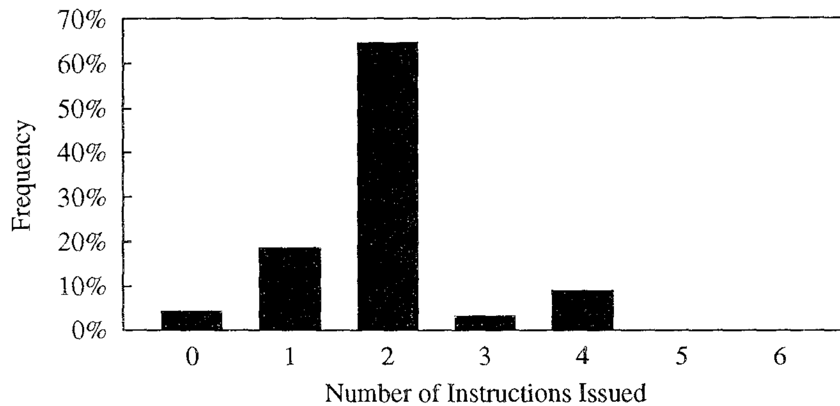

Figure 8: Distribution of Instruction Issuing Parallelism in Alvinn

6.2.2 Utilization of execution stages

Each function unit has a certain number of stages which determine the maximum number of instructions in flight in that unit in each cycle. The following table shows the number of execution stages in each function unit, and their average utilization. Not surprisingly, for *alvinn* the L/S unit is the most heavily utilized of the six units, being on average approximately 75% utilized.

Unit	XSU0	XSU1	MC-FXU	FPU	LSU	BRU
Number of Execution Stages	1	1	1	3	2	1
Average Number of Stages Utilized	0.50	0.23	0.02	0.74	1.52	0.20

Table 16: Execution Stage Utilization in Alvinn

6.2.3 Distribution of finishing parallelism

We expected the distribution of finishing parallelism to look exactly like the distribution of issuing parallelism because once an instruction issues, it cannot be delayed and must finish a certain number of cycles

later. Yet the following graph of finishing parallelism in *alvinn* is plainly not identical to the issuing parallelism graph shown earlier.

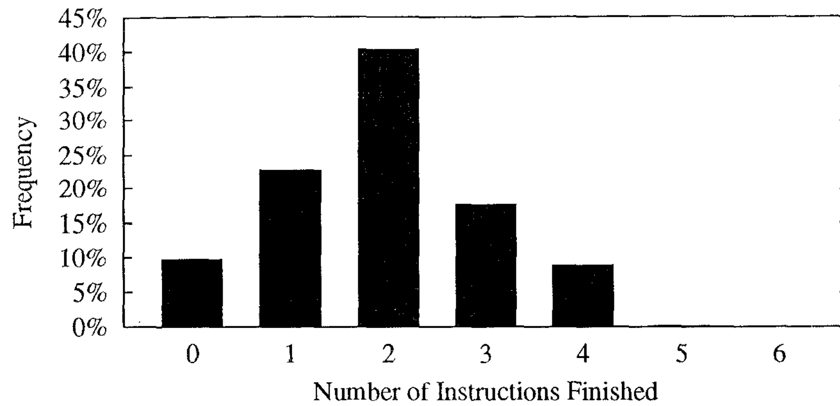

Figure 9: Distribution of Instruction Finishing Parallelism in Alvinn

The difference comes from the high frequency of L/S and FP instructions. Since these instructions do not take the same amount of time to finish after issuing as the integer instructions, they tend to randomize, and thus smooth, the issuing parallelism curve. The integer benchmarks, with their more consistent instruction execution latencies, have much more similar issuing and finishing parallelism distributions.

6.3 Distribution of Execution Times

The following table shows the distribution of the number of cycles instructions taken from dispatch to finish in each unit for the *alvinn* benchmark. Because each instruction spends a minimum of one cycle in a reservation station and one cycle being executed, two cycles is the minimum latency recorded. Notice that the FPU and the LSU, with their multiple stage pipelines, have a higher minimum latency. Although this

Latency	XSU0	XSU1	MC-FXU	FPU	LSU	BRU
1 cycle	——	——	——	——	——	——
2 cycles	25.2%	11.0%	0.0%	——	——	10.2%
3 cycles	0.0%	0.2%	0.0%	——	29.0%	0.3%
4 cycles	0.0%	0.0%	0.2%	0.1%	7.8%	0.1%
5 cycles	0.5%	0.5%	0.1%	0.6%	1.6%	0.0%
6 cycles	0.0%	0.0%	0.0%	9.0%	0.0%	0.0%
7 cycles	0.0%	0.0%	0.0%	1.8%	0.8%	0.0%
8 cycles	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%
9 cycles	0.0%	0.0%	0.0%	0.6%	0.0%	0.0%
Total % in Unit	25.7%	11.8%	0.3%	12.3%	39.3%	10.6%

Table 17: Distribution of Execution Times (Dispatch to Finish) per Unit in Alvinn

table only stretches as far as nine cycles, a very few instructions in the floating-point unit took as many as 26 cycles to finish. Each percentage in the table is out of the total number of dynamic instructions.

The distribution of execution times in the other benchmarks varies considerably. To impart a feel for this variation, the following table shows the average execution time (in cycles) within each unit for each benchmark.

Functional Units	*compress*	*eqntott*	*espresso*	*li*	*alvinn*	*fpppp*	*hydro2d*	*tomcatv*
XSU0	2.52	2.66	2.93	3.35	2.06	2.47	2.48	2.01
XSU1	2.87	2.75	3.23	3.44	2.16	2.56	2.78	2.03
MC-FXU	4.33	5.78	5.65	5.11	4.48	6.04	10.45	5.21
FPU	——[1]	——[1]	——[1]	——[1]	6.46	5.18	8.42	4.64
LSU	3.15	3.21	3.58	3.77	3.36	3.01	3.11	3.02
BRU	3.65	3.83	4.10	4.59	2.05	4.80	5.35	4.71

Table 18: Average Execution Time per Unit for All Benchmarks

1. Very few instructions are executed in this unit for these benchmarks.

7 PowerPC 620 Instruction Completion

Once instruction execution is finished, the 620 simply has to complete the instructions and write back the results in order.

7.1 Distribution of completion parallelism

The average completion parallelism is equal to the average dispatching, issuing, and finishing parallelisms which are all equal to the IPC of the benchmark. However, the actual distribution of the parallelism is determined by the method used to process instructions. In the case of instruction completion, instructions are allowed to finish out of order, but can only complete in order. This means that the 620 will occasionally have to wait for one slow instruction to finish, but then will be able to complete its maximum of four at once. The distributions of completion parallelism for all benchmarks are shown in Figure 3.

Note that the distribution for *alvinn* looks markedly different from the distribution for any other benchmark, as it has a good mix of instructions with various latencies. Thus, many of these instructions finish out of order, causing the sporadic completion described above. The integer benchmarks, with their more consistent execution latencies, most often have one instruction completed per cycle. *Hydro2d* completes zero instructions in a large percentage of cycles because it must wait for a FP divide to finish.

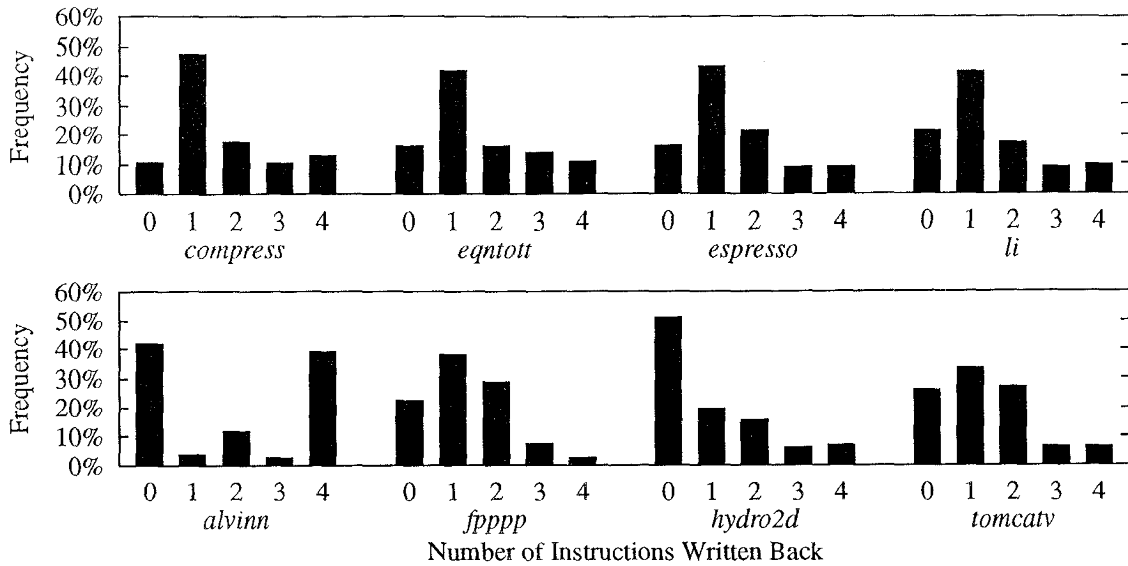

Figure 10: Distribution of Instruction Completion (Writeback) Parallelism for all Benchmarks

7.2 Completion stalls

Most often instructions cannot complete simply because they, or instructions preceding them, are not finished yet. Occasionally, though, they must wait to complete for other reasons. Although there are always enough condition register field writeback ports, occasionally too many instructions will need to write back to too many integer or floating-point registers. The 620 has four integer and two floating-point writeback ports. Note that integer load with update instructions write back two integer registers per instruction. There is also a restriction that only one instruction that reads or writes the link register can complete per cycle. Neither of these restrictions takes place very often.

Benchmark	*compress*	*eqntott*	*espresso*	*li*	*alvinn*	*fpppp*	*hydro2d*	*tomcatv*
GPR Write Port Saturation	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
FPR Write Port Saturation	0.0%	0.0%	0.0%	0.0%	1.1%	7.8%	4.6%	4.6%
LR Involved Serialization	0.0%	0.0%	0.1%	0.3%	0.0%	0.0%	0.0%	0.0%

Table 19: Frequency of Completion Hazards

It is very rare to run out of GPR write ports, and LR involved serialization is almost as rare. FPR write port saturation occurs occasionally, however. It occurs most frequently in the benchmark with the highest percentage of floating point instructions, *fpppp*.

8 Conclusion

A summary of key observations, ideas for further study, and an informal comparison of the PowerPC and the Alpha AXP are included in this section.

8.1 Summary

The PowerPC 620 does very well at branch prediction, often having zero delay cycles even for a taken branch. And even though precise interrupt is implemented, there is still a high degree of parallelism and out-of-order execution, especially with a well mixed stream of instructions, such as that in the *alvinn* benchmark. If the instruction mix is more bursty, it tends to create a bottleneck in the dispatch unit. Even so, the integer benchmarks do quite well, and even the *hydro2d* benchmark, with its high number of floating point divide instructions, still manages to achieve an IPC that RISC designers of a few years ago could only dream about.

One hot spot is the load/store unit. Although the difficulties of designing a two load/store unit system are myriad, the load/store bottleneck in the 620 is evident, and future, wider, processors will need that second unit. Having only one floating-point unit for four integer units is also a source of bottlenecks. The integer benchmarks rarely stall on the integer units, but the floating-point benchmarks stick waiting for FP resources often. The single dispatch to each reservation station in a cycle is also a serious source of dispatch stalls which can reduce the number of instructions considered for out-of-order execution.

8.2 Further Study

The results of this paper suggest multiple avenues for additional inquiry. At the time of the writing of this paper, the only compilers we had available were targeted for the PowerPC 601. Retesting the benchmarks with a compiler that optimizes for the 620, or even the 604, could reveal interesting results. Changes to the simulator can also be made. A cache analysis would be useful. Even more intriguing is to expand on the 620 and analyze the potential results of a 620^{+} system.

8.3 PowerPC vs. Alpha

The PowerPC and the Alpha AXP clearly represent two different approaches to achieving high performance in superscalar machines. The two approaches have been dubbed "brainiacs vs. speed demons" [1]. The PowerPC brainiacs attempt to achieve the highest level of IPC without overly compromising the clock speed. On the other hand, the Alpha AXP speed demons go for the highest possible clock speed while achieving a reasonable level of instruction-level parallelism. Which is the better approach in getting more

performance? Although the current versions of the PowerPC 620 and the Alpha AXP 21164 seem to indicate that the speed demons are winning, there is no strong consensus on the answer for the future. In an interesting way, the two rivaling approaches resemble, and perhaps is a reincarnation of, the CISC vs. RISC debate of a decade earlier.

An informal comparison of the two machines is of some interest. It appears that the 21164 is achieving 300 MHz clocking speed without any problem. Assume that the fastest 620 can be clocked at 150 MHz, which is half that of the 21164. Based on our results, the 620 is achieving an average IPC of about 1.53 (1.39) for integer (floating-point) benchmarks. There is a difference between the granularities of the two instruction sets. The more RISC-ish Alpha requires a higher number of instructions for the same benchmark. Table 20 below shows the number of dynamic instructions executed by an Alpha machine for the same set of benchmarks. The instruction counts are 3%-48% higher than that of the PowerPC, with an average of 24%. Scaled by this factor of 1.24, the IPC of 1.53 (1.39) achieved by the PowerPC 620 for integer (floating-point) benchmarks is effectively 1.90 (1.72) in terms of Alpha instructions. There is still no published IPC number for the 21164. Extrapolating from the reported SPEC ratings, it should be 1.1 for integer benchmarks and 1.7 for floating-point benchmarks. As compared to 1.90/2, the IPC of 1.1 gives the 21164 a slight edge over the 620 for integer benchmarks. For floating-point benchmarks, the 21164 is a clear win over the 620. Of course, there are many other issues involved such as scalability of IPC vs. MHz, compilation support for IPC enhancement, power, cooling and other system design issues, fabrication yield, and most importantly profitability. The debate continues.

Benchmarks	*compress*	*eqntott*	*espresso*	*li*	*alvinn*	*fpppp*	*hydro2d*	*tomcatv*
PowerPC	1,652,556	1,815,099	1,506,872	3,375,007	2,920,878	5,520,738	4,114,603	6,858,618
Alpha	2,172,362	2,680,047	1,593,372	3,600,169	3,721,157	7,631,144	5,330,292	7,064,977
Alpha / PowerPC	1.31	1.48	1.06	1.07	1.27	1.38	1.30	1.03

Table 20: Comparison of Instruction Counts

9 References

[1] Linley Gwennap. "Comparing RISC Microprocessors." *Proceedings of the Microprocessor Forum,* October 1994.

[2] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990

[3] Wen-Mei Hwu and Yale Patt. "Checkpoint Repair for High-Performance Out-of-Order Execution Machines." *IEEE Transactions on Computers,* December 1987.

[4] Raj Jain. *The Art of Computer Systems Performance Analysis,* John Wiley & Sons, 1991.

[5] Mike Johnson. *Superscalar Microprocessor Design*, Prentice-Hall, 1990.

[6] Norman Jouppi. "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance." *IEEE Transactions on Computers*, December 1989.

[7] Johnny K. F. Lee and Alan Jay Smith. "Branch Prediction Strategies and Branch Target Buffer Design." *Computer*, January 1984.

[8] Paul Rubinfeld. "An Overview of the Alpha AXP 21164 Microarchitecture." *Proceedings of the Hot Chips VI Symposium*, October 1994.

[9] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units." *IBM Journal of Research and Development*, January 1967.

[10] *IBM Assembler Language Reference Manual*, 1990.

[11] *Motorola Optimizing C and Fortran Compilation System User's Manual*, 1992.

[12] *PowerPC 601 RISC Microprocessor User's Manual*, 1993.

[13] *PowerPC 603 Microprocessor Implementation Definition, Book IV*, 1992.

[14] *PowerPC 604 Microprocessor Implementation Features Book IV*, 1993.

[15] *PowerPC 620 Microprocessor Implementation Definition*, 1992.

[16] *PowerPC Implementation Definition for the 601 Processor, Book IV*, May, 1992

[17] *RS/6000 Special Issue of the IBM Journal of Research and Development*, January 1990

[18] *PowerPC User Instruction Set Architecture, Book I*, November 1993

[19] *SPEC Newsletter*. Systems Performance Evaluation Cooperative, 1992.

[20] *VMW: A Visualization-based Microarchitecture Workbench*. Internal Technical Report, August 1994.