Limits on Multiple Instruction Issue

Michael D. Smith, Mike Johnson, and Mark A. Horowitz
Center For Integrated Systems
Stanford University, Stanford CA 94305

April 1989

1 Introduction

The performance of microprocessors has increased dramatically over the past few years. Part of the performance gain has come from reducing the cycle time, while the rest has come from decreasing the average number of cycles needed to complete an instruction. Current RISC processors already achieve an instruction-execution rate of nearly one instruction per cycle [Henn 86]. To continue improving processor performance, we are motivated to explore processors capable of executing more than one instruction per clock cycle; we call these *super-scalar processors*.

The problem of exploiting instruction-level concurrency has been extensively studied for scientific applications. In fact, there are already on the market simple super-scalar machines which allow concurrent integer and floating-point execution[Apol 88]. This paper demonstrates that highly-optimized, non-scientific applications also contain ample instruction-level concurrency to sustain an execution rate of two instructions per clock cycle. However, the cost requirements necessary to provide the instruction bandwidth needed by the instruction-execution unit make this performance difficult to achieve.

Our research uses trace-driven simulation both to evaluate the amount of instruction-level concurrency available in general-purpose applications and to determine how this concurrency is exploited by different hardware organizations. Section 2 introduces our approach to super-scalar processor design and discusses the aspects of our simulation methodology. Section 3 presents the available instruction-level concurrency found in highly-optimized, non-scientific applications executing on ideal super-scalar processors. The specifics of these processors are refined in Sections 4 and 5 to demonstrate that it is the cost and efficiency of the instruction-fetch unit, and not the instruction-execution unit, that limit performance.

2 Methodology

2.1 Super-Scalar Techniques

Many researchers have investigated super-scalar-like processors, and their research has produced seemingly contradictory results. Table 1 presents the reported performance advantages of some super-scalar-like processors. The wide range of potential speedups in Table 1 arises from differences both in the hardware of the simulated machine and in the applications being simulated.

The most glaring difference in the reported results is between those researchers who assume that their machines have an infinite number of functional units and those that assume some small number of functional units. To make use of a large number of functional units, the machines require very good branch prediction mechanisms. In fact, perfect branch prediction is assumed in the two cases with the highest speedups. Low speedups result from a limited number of functional units and simple branch handling schemes.

Other major hardware differences between researchers include the assumed issue and result latencies of the functional units and whether or not register renaming is performed to eliminate register storage conflicts. These factors can significantly affect the available concurrency. For example, the "slots" that arise in a pipelined

Research	Speedup
Weiss/Smith (1984)	1.58
Tjaden/Flynn (1970)	1.8
Sohi (1987)	1.8
Acosta et al (1986)	2.7
Kuck et al (1972)	8
Riseman/Foster (1972)	51
Nicolau/Fisher (1984)	90

Table 1: Reported Speedups of Super-scalar-like Machines

machine because of the result latencies use up some of the available concurrency. Ignoring this effect results in optimistic predictions of speedup.

The target application also has a significant influence on the available instruction-level concurrency. A significant portion of the published research strives to use super-scalar techniques to enhance the performance of vectorizable code. We suspect that one reason for the popularity of such benchmarks as the Livermore Loops and Linpack in measurements of instruction-level concurrency is that these applications have a high ratio of computation to dynamic branches. This high ratio in turn leads to large potential concurrency. All of the prior work in Table 1 experimented with this type of application.

For this study, we look at a set of common C programs representing a wide class of general, non-vectorizable applications. Since these applications, with their complex control flow, are thought to have little instruction-level concurrency, they form a good test of the generality of super-scalar techniques.

In order to exploit as much instruction-level concurrency in these applications as possible, we employ four organizational techniques. First, we assume that within the super-scalar processor there are multiple functional units which are independent and pipelined; this allows concurrent operation on different instructions and reduces the instruction-issue latencies. Second, the processor control can issue instructions out-of-order to allow the functional units to be better utilized in the presence of data dependencies. Third, it can also issue more than one instruction per cycle to maintain the instruction bandwidth needed by the instruction-execution unit. Finally, some branch prediction is used to reduce the cost of correctly-predicted branches and—more importantly—to allow the instruction-fetch unit to maintain the instruction bandwidth needed by the execution unit. For this paper, we consider these features to be implemented directly in hardware, though we also mention some possibilities for supporting these functions in software.

2.2 Simulation Method

To evaluate the feasibility of super-scalar processors for the benchmark applications, we built a trace-driven simulation system. The tracing system is based upon the MIPS R2000 RISC processor[1]. We chose the R2000 processor for two important reasons. First, MIPS machines are shipped with an excellent optimizing compiler, and second, MIPS provides analysis tools for their machines that are essential to our investigations. For instance, these analysis tools allow the easy generation of dynamic address and data traces and the convenient collection of dynamic instruction statistics [MIPS 86].

All simulations are performed as shown in Figure 1. We first take the optimized object code for a particular benchmark and run it through *pixie* [MIPS 86], a program which annotates the object code. When the annotated code is executed, it produces a dynamic trace stream of basic block entry points and load/store addresses, in addition to the program's normal output. This dynamic trace stream and the original object file are then fed into a simple instruction-lookup program. This program looks up the instructions given the basic block entry point and decodes the bit fields to produce the dynamic instruction/data trace needed by the instruction scheduler.

The instruction scheduler simulates the functionality of the scalar and super-scalar machines using the specified machine configuration file and some scheduling parameters. The instruction scheduler uses this information

[1]R2000 is a Trademark of MIPS Computer Systems, Inc.

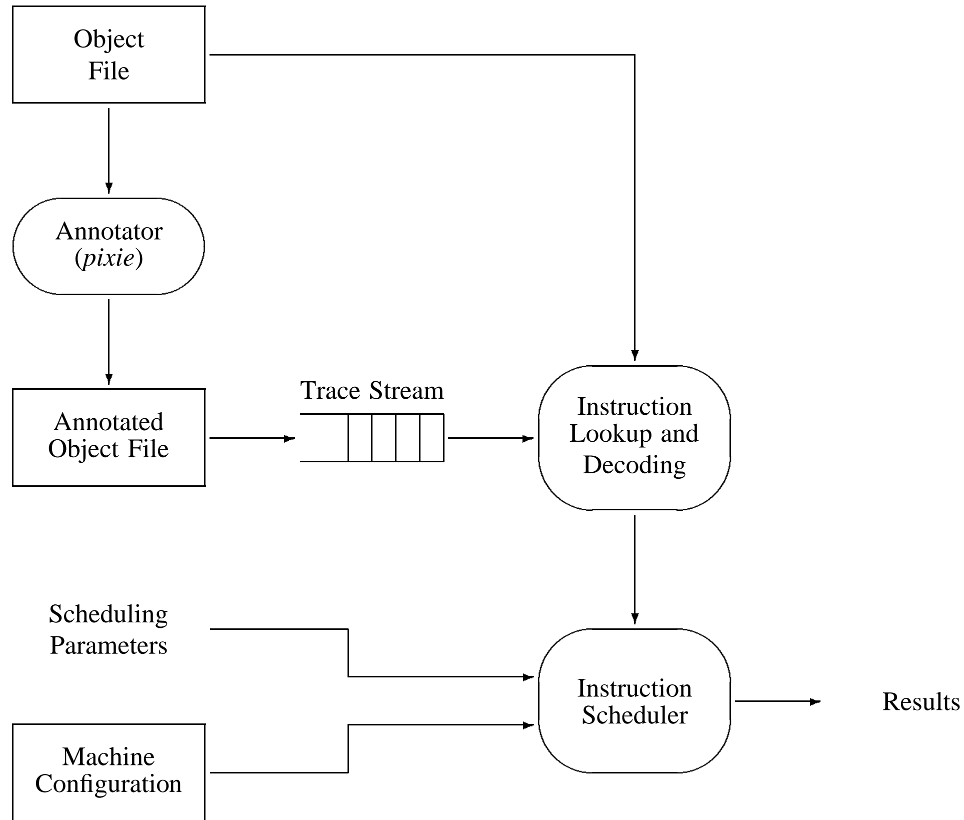

Figure 1: Flowchart of Trace-Driven Simulator

to model the execution of the scalar and super-scalar processors at a functional level, with time being recorded in terms of machine cycles. The cycle time of the super-scalar processor is assumed to be identical to that of the scalar processor (the question of whether the additional hardware in a super-scalar processor increases the cycle time is not considered in this paper). By keeping the number of instructions completed by each processor model equal, the simulator is able to compare the cycle counts to determine the performance advantage of the super-scalar organization over a basic RISC machine.

The machine configuration file specifies the number and type of available functional units along with their issue and result latencies. We based our simulations on functional units that are quite similar to those found in the R2000 processor and its associated floating-point coprocessor. The functional units are listed in Table 2. All floating-point result latencies in Table 2 are specified for double-precision operands. The FP Convert unit converts floating point numbers to integer format and vice-versa. Independent load and store pipes to memory are included so that the processor can issue a load and a store simultaneously. This provision, of course, does not imply that the machine must have two ports to memory/cache since the store pipe may be implemented as a buffer in front of a single memory pipe. The buffering of each store allows the store issue latency to be one cycle.

2.3 Simulation Data

The simulation traces are generated from a number of integer programs, such as compilers and text formatters, and some scalar floating-point applications, such as circuit and timing simulators. Table 3 lists the programs which were used as benchmarks for our simulations. All simulations used highly-optimized versions of these programs. To obtain processor measurements, each program is traced for five million instructions. In practice, the results do not change significantly after the first million instructions.

Functional Unit	Issue Latency (cycles)	Result Latency (cycles)
Integer ALU	1	1
Barrel Shifter	1	1
Load Pipe	1	2
Store Pipe	1	-
Branch Unit	1	1
FP Adder	1	6
FP Multiplier	1	6
FP Divider	1	12
FP Convert	1	4

Table 2: Functional Units with Issue and Result Latencies

Program	Description
5diff	compares two text files and reports all differences
awk	pattern scanning and processing language
ccom	front-end of a C compiler
compress	file compression using Lempel-Ziv encoding
espresso	minimizes a two-level representation of a Boolean function
gnuchess	computer chess program with computer playing itself
grep	reports all occurrences of a string within a text file
irsim	delay simulator for VLSI layouts
latex	document preparation system based upon Knuth's TEX
nroff	formats text for a typewriter-like device
troff	formats text for printing on a phototypesetter
wolf	TimberWolf standard cell placement program developed at UC Berkeley
yacc	compiles a context-free grammar into LR(1) tables

Table 3: Program Descriptions

The results in the section 3 are based upon the harmonic mean of all programs in the benchmark set. Since all of the benchmarks exhibited similar speedups, we chose four programs from Table 3 – *ccom*, *irsim*, *troff*, and *yacc* – as a representative sample for demonstrating and clarifying the points made in Sections 4 and 5. This sample simply limits the amount of data presented for purposes of illustration.

3 Available Concurrency

Since there is little previous work that deals specifically with general-purpose applications, our initial experiments focused on determining the amount of instruction-level concurrency in these applications and on identifying any major performance bottlenecks. For these experiments, we built a simulator that modelled the effects of four main factors we felt would limit performance: data dependencies and register storage conflicts, the number and latency of the functional units, the instruction fetch width, and the accuracy of predicted branches. The simulator was not intended to precisely model a real machine. It ignored such performance constraints as load-store dependencies and cache misses. Once we better understood the problem domain, we wrote a more detailed machine simulator for both the instruction-execution and the instruction-fetch unit. These are described in Sections 4 and 5.

For the super-scalar processor model, the instruction scheduler performs the following functions during each cycle: (1) it prefetches a block of instructions from the decoded, dynamic trace stream and attempts to place

these instructions in the *instruction window*; (2) it issues instructions from this window to available functional units and records their result latency; and (3) it removes completed instructions from the instruction window.

An instruction prefetch buffer in the simulator limits the number of instructions placed into the instruction window during a single cycle. The prefetch buffer is refilled from the instruction trace only when the buffer is completely empty.

As instructions are placed in the window, they are checked for data dependencies and register storage conflicts on all other instructions already residing in the window. The instruction window keeps a scoreboard which can track either *all dependencies* or just *true dependencies* between instructions. The term *all dependencies* means that the simulator checks for true, anti-, and output data dependencies; anti- and output data dependencies are due to register storage conflicts. The term *true dependencies* means that the simulator checks only for true dependencies and assumes that all register storage conflicts are handled by register renaming[2].

An instruction is issued when it is not dependent upon any other instruction in the window and its required functional unit is available. Instructions can be issued to the functional units only from the instruction window, and not from the instruction prefetch buffer. An instruction remains in the window until it has completed execution. Upon completion, the instruction is removed from the window and all data dependencies upon this instruction are cleared.

3.1 Ideal Fetch Unit Results

The first set of results is based upon an ideal instruction-fetch unit so that we may independently investigate the effects of both the instruction window size and the number of functional units on instruction-level concurrency. Concurrency is measured in terms of speedup, where speedup is defined to be the total number of cycles that the scalar architecture needs to execute a benchmark divided by the total number of cycles that the super-scalar architecture needs. The fetch unit is ideal in that it performs perfect branch prediction and has a prefetch buffer capable of placing as many instructions into the instruction window during a single cycle as there are empty slots in the window. In other words, the instruction window is always kept full, and control dependencies are ignored.

In each of the following graphs of available concurrency, the vertical axis corresponds to the mean speedup of the super-scalar machine over the scalar machine. This speedup is the harmonic mean of the total benchmark set. The horizontal axis is indexed by a range of super-scalar machines that differ in the number of functional units available. The machine organizations along the horizontal axis correspond to those listed in Table 4. Machine configuration 1 contains one of each of the functional units listed in Table 2. This machine is the functional-unit equivalent of the original scalar processor with a super-scalar architecture. Machine configuration 2 is identical to configuration 1 except that it has an extra load pipe; machine configuration 3 is identical to configuration 1 except that it has an extra integer ALU. Machine configuration 4 is a combination of configurations 2 and 3 in that it adds an extra load pipe and integer ALU to the basic machine configuration. A different instruction window size is used for each shaded bar in the graphs.

Figure 2 presents the mean speedup for a machine with an ideal fetch unit that checks for all data dependencies. The mean speedup varies only slightly, from 1.9 to 2.5, over a range of window sizes and machine configurations. The most noticeable increase in performance comes from adding an extra integer ALU in machine configuration 3. A second load pipe (machine configuration 2) increases performance only by a small amount. Even though the machines with the larger window sizes and greater number of available functional units can look further into the instruction stream and theoretically execute more instructions concurrently, the additional hardware goes unused because these larger machines are limited by data dependencies.

As stated previously however, register storage conflicts can be removed through register renaming. Figure 3 contains results for the same machine configurations shown in Figure 2, when checking only for true data dependencies between instructions. As can be seen, the speedups have increased dramatically to a range of 2.3

[2]As an aside, it is interesting to note that register renaming is necessary not because of the inherent structure of the code itself, but as Slavenburg [Slav 88] points out, because of how the compiler tries to minimize the number of live registers at any given time. Often a compiler will choose a limited number of registers to be used as temporaries or to hold the most active values in a loop [Aho 86]. Thus, when one of these values dies, its register is immediately reused for a new temporary or active value. This method of register allocation produces many register storage conflicts in the code, and causes register renaming to be necessary in super-scalar architectures. If the method of register allocation in the compiler could be changed, the need for register renaming might be reduced.

Machine	Configuration
1	One of each of the available functional units
2	Machine 1 plus an extra Load Pipe
3	Machine 1 plus an extra Integer ALU
4	Machine 1 plus an extra Load Pipe and Integer ALU

Table 4: Simulated Super-Scalar Machine Configurations

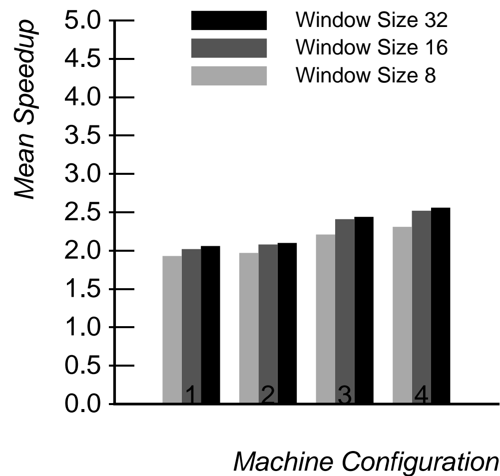

Figure 2: Mean Speedup for a Super-Scalar Machine Checking All Dependencies with an Ideal Fetch Unit

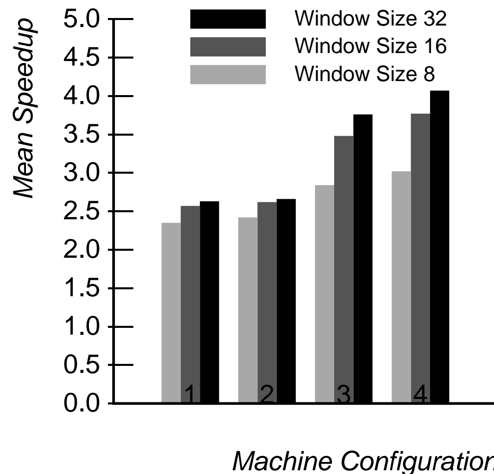

Figure 3: Mean Speedup for a Super-Scalar Machine Checking True Dependencies with an Ideal Fetch Unit

to 4.1 times over the spectrum of machine configurations and window sizes. The largest gain in performance still results from the addition of a single integer ALU rather than a load pipe. Now, however, the relative performance change between window sizes increases as the number of available functional units is increased. This indicates that the machine must look farther and farther ahead in the instruction stream to find operations to keep the additional functional units busy. If the window is kept small, data dependencies between instructions prevent the additional units from being effectively utilized.

3.2 Non-Ideal Fetch Unit Results

The previous results are very optimistic in that we used a perfect instruction-fetch unit. The performance was limited by execution hardware and data dependencies; control dependencies were ignored. Including these control dependencies has a dramatic effect on performance, especially since branches are sometimes mispredicted. Given the small average number of instructions between branch instructions (five) and the even smaller median (three) for our benchmarks, branch instructions are encountered frequently. Branch instructions reduce the machine's ability to look far ahead in the instruction stream and limit the available concurrency. In addition, taken branches cause instruction fetches to addresses that are not always fetch-aligned. Unaligned fetches lower the effective fetch bandwidth, and cause performance to decrease.

Figure 4 shows performance when control dependencies are considered and the prefetch size is limited. The super-scalar model is limited to prefetching a maximum of four instructions per cycle, and the branch prediction accuracy is set to an 85% correct prediction rate[3]. The range of speedups is reduced to 1.9 to 2.3 times that of a scalar processor. Furthermore, the addition of functional units and the enlargement of the instruction window do not significantly increase performance, as they did in Figure 3.

These results indicate that there exists enough instruction-level concurrency in highly-optimized code to support an approximate two-times speedup in performance. To exploit this level of concurrency requires only a limited number of functional units. In fact, for this class of applications, fetch issues seem to make machines with larger numbers of functional units not cost-effective. Using this information as a starting point, we proceeded to implement a more detailed super-scalar machine model to ensure that there were no problems that these initial simulations overlooked.

[3]An 85% prediction rate is comparable to what can be obtained by a hardware branch target buffer [Lee 84] or software profiling [McFa 86]. The simulator, though, just randomly chose 85% of the branches and predicted them correctly to generate the results in Figure 4.

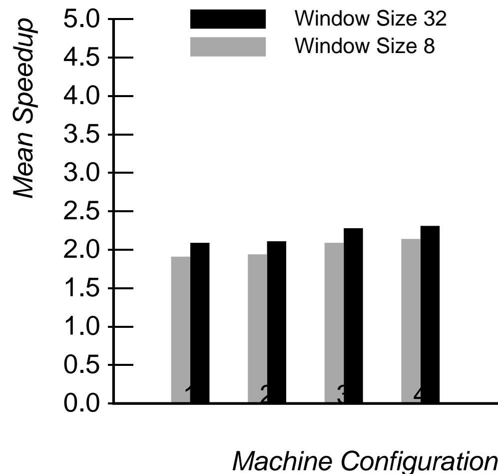

Figure 4: Mean Speedup for a Super-Scalar Machine Checking True Dependencies with an 85% Branch Prediction Accuracy and a Prefetch Width of Four Instructions

Program	Number of instructions issued per cycle								avg. inst/ cycle
	0 inst	1 inst	2 inst	3 inst	4 inst	5 inst	6 inst	7+ inst	
ccom	.071	.18	.22	.25	.19	.071	.013	.001	2.57
irsim	.079	.18	.14	.34	.21	.053	.007	.001	2.62
troff	.073	.17	.20	.32	.16	.066	.014	.000	2.58
yacc	.090	.21	.20	.27	.19	.039	.005	.000	2.39

Table 5: Instruction-Issue Distribution and Resulting Instruction-Execution Rate

4 Machine Implementation

There are a great number of proposed super-scalar (or conceptually similar) architectures, so we were faced with a large number of possible implementations. Since we could not implement a simulator for every possibility, our approach was to select a reasonable class of machines; this allowed us to examine in more detail the effects of execution constraints on a super-scalar processor. This section briefly presents the implementation choices we made, and why we chose this organization.

4.1 Motivation

The designer of a super-scalar machine is faced with a cost-performance dilemma. The peak number of instructions per cycle that the machine can execute determines its cost, but the average number of instructions per cycle that the machine executes determines its performance. For speedups of two, the ratio between peak and sustained performance can be over a factor of two, as is shown in Table 5. This table gives the fraction of total cycles in which 0, 1, 2, 3, etc. instructions are issued. The aggregate instruction-execution rate is also shown. This data is for machine configuration 4 with an 85% branch prediction accuracy and an infinite prefetch buffer. We present this data to make the point that the maximum number of instructions issued in one cycle can be significantly higher than the aggregate instruction-issue rate.

However, issuing many instructions per cycle can be quite expensive, because of the cost incurred in communicating operand values to the functional units. Each instruction issued in a single cycle must be

accompanied by all required operands, so issuing N instructions in one cycle requires access ports and routing buses for as many as $2 * N$ operands. To reduce this cost, we could limit the number of instructions issued per cycle to two, but—as mentioned above—this would decrease the performance of the machine.

Instead of directly limiting the instruction issue, we chose to distribute the instruction window among the functional units in reservation stations [Toma 67]. With this organization, the number of instructions decoded is set according to the average instruction-execution rate, which in turn sets the number of buses and ports on the register file. However, each functional unit can start the execution of an instruction in each cycle. The data needed for instruction issue comes from the local reservation stations, rather than a global register file.

4.2 Implementation Description

In the hardware simulator, the execution unit is comprised of a number of functional units, each with an associated reservation station. The instruction decoder places instructions and operands into the reservation stations of the appropriate functional units. A functional unit can issue an instruction in the cycle following decode if the instruction has no data dependencies and the functional unit is not busy; otherwise, the instruction is stored in the reservation station until all dependencies are released and the functional unit is available.

Register renaming [Kell 75, Logr 72] is used to eliminate register storage conflicts. To implement register renaming, the processor incorporates a result buffer containing a number of storage locations that are dynamically allocated. When an instruction is decoded, its result value is assigned a result-buffer location, and its destination-register number is associated with this location (i.e. the destination register is renamed). A subsequent reference to the renamed destination register obtains the value stored in the result buffer.

The result buffer is implemented as a content-addressable memory. It is accessed using the register number as a key, and returns the latest value written into the register. This organization performs name mapping and operand access in a single cycle, mimicking the register file.

During instruction decode, the result buffer is accessed in parallel with the register file. Then, depending on which one has the most recent value, the desired operand is selected. The operand value—if available—is copied to the reservation station. If the value is not available (because it has not been computed yet), an identifier for the result-buffer entry is copied to the reservation station. This procedure is carried out for each operand required by each decoded instruction.

If a register mapped by the result buffer is the destination of a decoded instruction, the previous mapping is marked as invalid, so that subsequent instructions obtain the result of the new instruction. At this point, the old register value can be discarded, but we chose to preserve it to simplify interrupt and exception handling [Sohi 87].

When a result becomes available, it is written to the result buffer and to any reservation stations containing identifiers for the result-buffer entry (note that this requires content-addressable memory in the reservation stations). Subsequent instructions continue to fetch the value from the result buffer—unless the entry is superseded by a new value—until the value is retired by writing it to the register file. Retiring occurs in the order given by sequential execution, which preserves the sequential state for interrupts and exceptions.

Our design of the super-scalar pipeline closely parallels the design of a sequential RISC pipeline. The objective is to keep the execution rate of sequential instructions as high as the execution rate in a sequential processor. The parallels between the pipeline of the sequential processor and of the super-scalar processor are shown in Table 6.

Table 6 does not show every function performed in the pipeline stages, but does illustrate how the additional functions required by the super-scalar processor fit into the pipeline stages of the sequential processor. In essence, the result buffer augments the register file and operates in parallel with it. The reservation stations replace the functional-unit input latches. The distribution of operands and writing of results are similar for both processors, except that the super-scalar processor requires more hardware, such as buses and write ports.

Loads and stores occur on a single, 32-bit bus to a perfect data cache. Stores are buffered to resolve contention for the data-cache interface; loads are given priority for the use of this interface, since an uncompleted load is more likely to stall computation. A load is issued in program-sequential order with respect to other loads, and likewise for a store. Furthermore, a store is issued only after all previous instructions have completed, to preserve the processor's sequential state in the data cache. Finally, this simulator addresses a shortcoming of

Pipeline Stage	Sequential Processor	Super-Scalar Processor
Fetch	fetch one instruction	fetch multiple instructions
Decode	decode instruction access operands from register file copy operands to functional unit input latches	decode instructions access operands from register file and result buffer copy operands to functional-unit reservation stations
Execute	execute instruction	execute instructions arbitrate for result buses
Write-back	write result to register file forward results to functional-unit input latches	write results to result buffer forward results to functional-unit reservation stations
Result Commit	n/a	write results to register file

Table 6: Parallels Between the Pipelines of a Sequential Processor and a Super-Scalar Processor

Benchmark	Speedup	Predicted Speedup
ccom	2.17	2.18
irsim	2.28	2.26
troff	2.12	2.12
yacc	2.16	2.18

Table 7: Hardware Simulation Results—Instruction Fetch Effects Disabled

our initial simulator in that it correctly detects and resolves memory dependencies between loads and stores.

Table 7 shows the performance obtained by simulating our benchmarks on this hardware configuration. The processor organization is comparable to machine configuration 1. It has one each of the functional units listed in Table 2, a 16-entry result buffer for integer results, and an 8-entry result buffer for floating-point results. Table 7 also shows the performance predicted by our preliminary simulator for machine configuration 1 with a window size of 8. For all benchmarks, the overhead introduced by real hardware (e.g. the result buffer) has had negligible effect on performance[4].

The purpose of Table 7 is to demonstrate that the execution unit—with correctly sized reservation stations—almost never limits performance. It is important to note that we disabled the effects of instruction fetching to obtain these results. Branches were perfectly predicted, and instruction alignment was ignored, as was done in the original experiment. Unfortunately, unlike the execution unit, the instruction fetch unit often starves the decoder, and is the primary limit to performance.

5 Fetch Limitations

Up to this point, we have established that there is sufficient instruction-level concurrency in optimized RISC code to support an instruction-execution rate of about two instructions per cycle. Furthermore, we have established

[4]The *irsim* benchmark is slightly better on real hardware than predicted because the reservation stations do not correspond exactly to the instruction window of the original simulator. The additional reservation stations on the floating-point functional units effectively increase the window size, and cause the results to be slightly better than predicted.

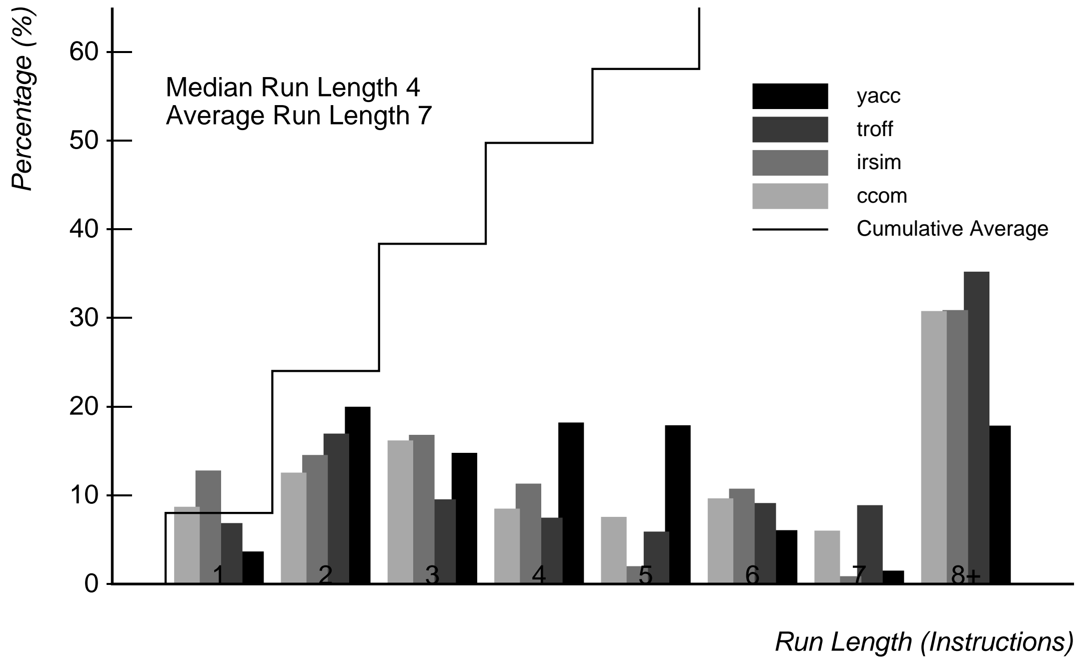

Figure 5: Dynamic Run Length Distribution between Taken Branches

that even limited execution hardware is sufficient to exploit the available concurrency. However, we have only briefly touched on the subject of branches; this was not because we think that branches are unimportant, but because we want to emphasize the fact that fetch limitations caused by branches—rather than execution-hardware constraints—limit the performance of the super-scalar processor. The effects of branches are considered in this section.

Branches affect the performance of the super-scalar architecture in the usual way, by introducing a pipeline dependency between the execution stage and the instruction-fetch stage. This dependency can prevent the instruction-fetch stage from fetching the proper instructions, starving the execution unit. But branches also affect the execution rate in a way that is unique to the super-scalar processor. Even if branches can be perfectly predicted, they disrupt the sequentiality of instruction addressing, causing instructions to be misaligned with respect to the instruction decoder. This in turn inhibits the ability of the instruction fetcher to keep the machine supplied with instructions at the required rate.

For example, consider a case where there are three instructions between branch points, and assume that the decoder processes two instructions per cycle. Since preceding and succeeding instructions are not at sequential addresses, the instruction fetch unit requires at least two cycles to supply these three instructions to the decoder. The resulting fetch efficiency is 1.5 instructions per cycle—well below the target execution rate of 2. Even this rate is optimistic, because the fetch unit must decode the branch and calculate its destination address. There is at least one idle cycle before the instruction at the target address can be fetched. With this additional cycle, the fetch efficiency for a three-instruction run drops to 1 instruction per cycle, or about the same as the sequential machine.

As the above example illustrates, fetch efficiency is quite dependent on the number of sequentially-fetched instructions between branches. We call these instructions a *run*, and refer to the number of instructions fetched sequentially as the *run length*. The run length does not include no-ops which are included for pipeline scheduling. The larger the run length, the better the ability of the instruction fetcher to keep the execution unit satisfied with instructions.

Figure 5 shows the distribution of run lengths for the benchmarks presented previously. The presence of a few large blocks makes the average run length (seven) a poor indicator of run length, since the median run (four) is about half this size. Table 8 shows the average fetch efficiencies by run length measured during program execution. Although the actual fetch efficiencies are slightly program dependent (the alignment of the runs is not perfectly random), they are quite close to the average values given in Table 8.

Fetch Block Size	Number of Instructions in Run							
	1 inst	2 inst	3 inst	4 inst	5 inst	6 inst	7 inst	8+ inst
2	0.50	0.77	1.00	1.16	1.25	1.40	1.40	1.62
4	0.50	0.91	1.21	1.57	1.67	1.91	1.98	2.58

Table 8: Fetch Efficiency Including 1 Cycle Branch Stall (inst/cycle)

Fetch Constraints	Fetch/Decode Width (inst)	Fetch Efficiency (inst/cycle)				
		ccom	irsim	troff	yacc	avg.
One cycle delay for taken branches, and no branch prediction	2/2	1.39	1.31	1.38	1.28	1.34
	4/4	1.97	1.84	1.95	1.77	1.88
One cycle delay for taken branches with branch prediction	2/2	1.64	1.68	1.67	1.65	1.66
	4/4	2.53	2.67	2.60	2.58	2.60
	4/2	1.81	1.79	1.83	1.80	1.81

Table 9: Aggregate Fetch Efficiencies for Various Instruction Fetchers

The aggregate fetch efficiency is computed by multiplying the fetch efficiency of each run length by the fraction of all runs having the corresponding length, and summing the results. The aggregate efficiency is presented in Table 9. A two-instruction fetch unit with no branch prediction can supply only 1.34 inst/cycle on average, while a similar four-instruction unit can supply 1.88 inst/cycle. These results are discouraging, since—from a hardware perspective—we would like to keep the fetch and decode widths small.

We can consider adding a *branch target buffer* [Lee 84] to predict branches, eliminating the branch delay cycle when a branch is correctly predicted. The buffer holds the destination of the branch, so when an instruction hits in the cache, the target can be fetched immediately. In this case, instruction runs are sometimes shorter, because the branch target buffer may incorrectly predict that a non-taken branch is taken. On the other hand, eliminating the delay cycle for many taken branches yields an overall benefit, as shown in the second block of Table 9.

However, there are two problems with this approach. As shown in Figure 6, a large buffer is required to achieve good branch prediction because of the poor hit rate in this type of structure. The other problem with this approach is that it does nothing to improve instruction alignment. Even with a large, 2048-entry buffer, the efficiency of a two-instruction fetch unit is still a discouraging 1.66 inst/cycle. If branches were perfectly predicted, and there were no branch delay cycles, we have found that misalignment still limits the fetch efficiency of a two-instruction decode stage to 1.70-1.75 instructions per cycle.

The essential problem with a two-instruction decode stage is that the instruction fetcher can never exceed two instructions per cycle—the fetch efficiency is always below this limit. However, even though a four-instruction decoder can overcome this limit, it is difficult to see how this can be cost-effective, given the processor organization discussed in Section 4. To decode four instructions with this organization, we require eight read ports on both the register file and the result buffer, and eight buses for distributing operands. The content-addressable accessing of operands during decode requires twelve comparators per result-buffer location (eight for source-register numbers and four for destination-register numbers)[5]. Checking dependencies between

[5]The result buffer has 16 locations for the simulation results in Section 4. To decode four instructions, we would require 192 comparators.

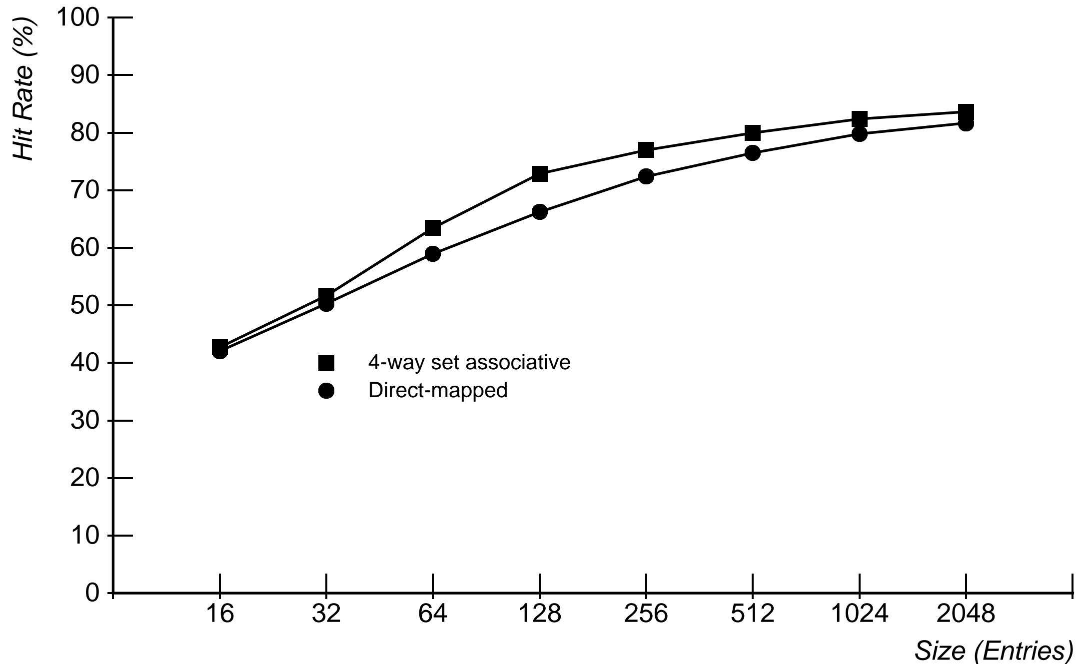

Figure 6: Branch Target Buffer Hit Rates

the decoded instructions requires another eighteen comparators. At the same time, the maximum performance improvement that can be achieved is probably not more than about 20–25% over that achieved with a two-instruction decode stage (see Table 7).

The efficiency of a two-instruction decoder can be improved by fetching four instructions in one cycle and then selecting the desired two instructions for the decoder. This arrangement reduces the probability that the desired words cross a fetch boundary and slightly increases the fetch efficiency. But this scheme still requires a large branch target buffer to remove the cycle delay for branches, and yields an efficiency of only 1.81 inst/cycle, as shown in the last line of Table 9.

A more hardware efficient method to improve the fetch efficiency relies heavily on the compiler to reduce the problems associated with branches. Recent work in static branch prediction using profiling has reported prediction rates of over 85% [McFa 86]. If the compiler can effectively predict the outcome of a branch, it can move instructions from the path of the likely outcome to pad one, two, or three instructions after the branch. These instructions are executed if the prediction is correct, and are nullified if the prediction is not correct.

This code motion removes both the penalties due to misalignment and to fetch stalls when the branch outcome is properly predicted. With the high prediction accuracy, a two-word fetch unit can achieve 1.8 to 1.9 inst/cycle, if all the branch targets are aligned on double word boundaries. This is comparable to the efficiency obtained by fetching four instructions with no branch prediction. We are currently working on a system to look at the code expansion when these optimizations are done to see if this amount of code motion is feasible. We are also looking at simple hardware methods for nullifying the effect of a variable number of instructions following the branch, depending on the alignment and outcome of the branch.

6 Future Directions

The trace-driven simulations show that highly-optimized, general-purpose applications contain enough instruction independence to sustain an execution rate of about two instructions per cycle. In fact, it is not too difficult or expensive to build the execution hardware necessary to obtain this performance advantage. The real difficulty lies in providing the instruction bandwidth required by the execution unit given the frequency of branches and the random alignment of instructions in memory.

Techniques used to sustain fetch bandwidth requirements in typical pipelined RISC processors do not provide an adequate solution for super-scalar architectures. Not only do branches cause potential pipeline "bubbles" more often in a super-scalar processor, but each bubble prevents the execution of multiple instructions rather than a single instruction. Hardware solutions, such as large branch target buffers and early evaluation of branch conditions, can help reduce the pipeline bubbles. However, these solutions do nothing to alleviate the penalty due to instruction alignment.

A possibly less expensive solution would involve the use of the compiler to do significant amounts of code motion and code duplication to align and pad out basic blocks. This solution is intriguing in that there exists potential opportunities for additional compiler optimizations to increase the instruction-level concurrency. Several current compiler optimizations, such as loop unrolling, are already used to reduce the penalties associated with branches in scalar processors. These optimizations also reduce the penalties due to instruction alignment. Other compiler optimizations can be envisioned to decrease the number of data dependencies and the randomness of the instruction-level concurrency in the code. This compiler technology could lead to an even smaller and cheaper super-scalar implementation which still retains its original performance advantages.

7 Acknowledgements

Mike Johnson was supported by Advanced Micro Devices, Inc. This work was supported in part by DARPA, under contract number N00014-87-K-0828.

We are indebted to Earl Killian of MIPS Computer Systems, Inc. for his assistance in explaining the mysteries of the *pixie* tracing facility.

References

[Acos 86] R.D. Acosta, J. Kjelstrup, and H.C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors". *IEEE Transactions on Computers*, Vol. C-35 (September 1986), pp. 815-828.

[Aho 86] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[Apol 88] Apollo Computer Inc. Marketing Brochure, *The Series 10000 Personal Supercomputer*. Chelmsford, MA, 1988.

[Fost 72] C.C. Foster and E.M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution". *IEEE Transactions on Computers*, Vol. C-21 (December 1972), pp. 1411-1415.

[Henn 86] J.L. Hennessy, "RISC-Based Processors: Concepts and Prospects". *New Frontiers in Computer Architecture Conference Proceedings* (March 1986), pp. 95-103.

[Kell 75] R.M. Keller, "Look-Ahead Processors". *Computing Surveys*, Vol. 7, No. 4 (December 1975), pp. 177-195.

[Kuck 72] D.J. Kuck, Y. Muraoka, and S. Chen, "On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resulting Speedup". *IEEE Transactions on Computers*, Vol. C-21 (December 1972), pp. 1293-1310.

[Lee 84] J.K.F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design". *IEEE Computer* (January 1984), pp. 6-22.

[Logr 72] L. Logrippo, "Renaming in Program Schemas". *Proceeding of the IEEE 13th Annual Symposium on Switching and Automata Theory*, (October 1972), pp. 67-70.

[McFa 86] S. McFarling and J. Hennessy, "Reducing the Cost of Branches". *Proc. 13th Annual Symposium on Computer Architecture* (June 1986), pp. 396-404.

[MIPS 86] MIPS Computer Systems, Inc., *MIPS Language Programmer's Guide* (1986).

[Nico 84] A. Nicolau and J.A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures". *IEEE Transactions on Computers*, Vol. C-33 (November 1984), pp. 968-976.

[Rise 72] E.M. Riseman and C.C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps". *IEEE Transactions on Computers*, Vol. C-21 (December 1972), pp. 1405-1411.

[Slav 88] G.A. Slavenburg, Phillips Research Laboratories Sunnyvale, Signetics Corporation, Sunnyvale, CA. Personal Correspondence, 12 May 1988.

[Smit 87] J.E. Smith, et al, "The ZS-1 Central Processor". *Proceedings, Second International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1987), pp. 199-204.

[Sohi 87] G.S. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors". *Proceedings, 14th Annual International Symposium on Computer Architecture* (June 1987), pp.27-34.

[Tjad 70] G.S. Tjaden and M.J. Flynn, "Detection and Parallel Execution of Independent Instructions". *IEEE Transactions on Computers*, Vol. C-19 (October 1970), pp. 889-895.

[Toma 67] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". *IBM Journal*, Vol. 11 (January 1967), pp. 25-33.

[Weis 84] S. Weiss and J.E. Smith, "Instruction Issue Logic in Pipelined Supercomputers". *IEEE Transactions on Computers*, Vol. C-33 (November 1984), pp. 1013-1022.

[Wulf 88] W.A. Wulf, "The WM Computer Architecture". *Architecture News* (January 1988), pp. 70-84.