Performance Estimation of Multistreamed, Superscalar Processors

Wayne Yamamoto wayne@mimd.ucsb.edu
Mauricio J. Serrano mauricio@misd.ucsb.edu
Adam R. Talcott talcott@mimd.ucsb.edu
Roger C. Wood wood@hub.ucsb.edu
Mario Nemirovsky mario@misd.ucsb.edu

Department of Electrical and Computer Engineering
University of California, Santa Barbara
Santa Barbara, CA 93106-5130

Abstract

Multistreamed processors can significantly improve processor throughput by allowing interleaved execution of instructions from multiple instruction streams. In this paper, we present an analytical modeling technique to evaluate the effect of dynamically interleaving additional instruction streams within superscalar architectures. Using this technique, estimates of the instructions executed per cycle (IPC) for a processor architecture are quickly calculated given simple descriptions of the workload and hardware characteristics. To validate this technique, estimates of the SPEC89 benchmark suite obtained from the model are compared to results from a hardware simulator. Our results show the technique produces accurate estimates with an average deviation of ~4% from the simulation results. Finally, we demonstrate that as the number of functional units increases, multistreaming is an effective technique to exploit these additional resources.

1 Introduction

Today's processors employ multiple functional unit designs capable of dispatching several instructions every cycle. Two factors limiting the number of instructions dispatched per cycle are: 1) the number of functional units available (hardware) and 2) the amount of parallelism in the workload (software). While the peak throughput of a processor is determined by the number of functional units, the actual performance obtained is limited by the amount of instruction level parallelism. Data dependencies and control breaks constrain instruction level parallelism resulting in a sustained system performance that is well below the peak. Combining hardware and compiler techniques can increase the level of parallelism. Register renaming, out-of-order execution, branch prediction, and speculative execution are some of the hardware techniques that have been used. On the compiler side, loop unfolding, software pipelining, and instruction reordering are some of the techniques employed. However, as the number of functional units increases within processors, it is unlikely that enough parallelism can be extracted from a single stream to effectively utilize the additional resources.

The ability to simultaneously execute multiple instruction streams, referred to as *multistreaming*, significantly increases the number of independent instructions capable of being issued per cycle. A multistreamed, superscalar processor can dispatch instructions from multiple streams simultaneously (i.e., within the same cycle). By storing each stream context internally, the scheduler can select independent instructions from all active streams and dispatch the maximum number of instructions on every cycle. A dynamically interleaved processor adjusts the scheduling policy as the workload changes to maximize throughput. If the active streams are independent then the total number of instructions that can be dispatched per cycle will increase as the number of active streams increases.

A simulation study demonstrating performance benefits of multistreamed, superscalar processors was performed by emulating a multistreamed version of the IBM RS/6000 [10]. Due to the cost of the simulations in this study, both in complexity and execution time, we developed a simple analytical model to estimate the overall performance of these architectures. The model produces instructions executed per cycle (IPC) estimates for an architecture as the number of streams is varied. As input parameters, the model uses simple workload and architectural descriptions that are easily estimated or obtained using commonly available tools. We compare the results from the model using the SPEC89 benchmark characteristics as workloads with results from a hardware simulator executing the real SPEC89 benchmarks. Our results show that the model produces estimations that differ from the simulation results by just over 4% on average. In addition, the execution time required for the analytic model is negligible compared to the time required for simulation.

2 History

Even though the CDC 6600 made use of a multiple functional unit design, the idea of multiple instruction issue was not pursued until many years after its introduction in 1964 [12]. This was due to several studies in the early 70's that concluded the amount of parallelism at the instruction level was very small [3]. However, by

Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, 1994

the mid 80's, new work showed that there was significant parallelism available. By the late 80's, several papers proposed techniques for multiple issue and gave them the name *superscalar*. As compilers got smarter, they were able to extract higher levels of parallelism than was originally thought. Several compiler techniques are used to increase the amount of parallelism, for example loop unfolding, software pipelining, and instruction reordering [5]. At the hardware level, techniques such as speculative execution and register renaming [5] are employed.

Another technique used to increase throughput has been instruction interleaving. Interleaving is a way to share the processor resources between multiple streams. A pipeline is interleaved if an instruction from a different instruction stream enters the pipe at every cycle, and there are at least as many instruction streams as pipe stages. Instruction interleaving was introduced in the peripheral processor of the CDC6600[12]. The main feature of an interleaving architecture is its ability to hide high latency operations. For instance, the CDC6600 used interleaving because the memory latency was large. It interleaved ten processes in a cyclic way and the cycle time for the ten processes was equal to the memory access time. Other examples of interleaved architectures include the multiple instruction stream processor of Kaminsky and Davidson[7], the Denelcor HEP computer [8], the UCSB CCMP computer [11], MASA [4], and APRIL [1]. Dynamic instruction interleaving [9] has been proposed as a way to increase throughput in real time system. In this technique, the interleaving between the processes is controlled by a scheduler and dynamic reallocation takes place when a process is blocked.

In this paper, we study multistreamed, superscalar processors. These processors combine the multiple instruction issue per cycle concept (superscalar) with the ability to dynamically interleave the execution of multiple streams.

3 Multistreamed, Superscalar Processors

A multistreamed, superscalar processor is organized to support the simultaneous execution of several instruction streams. Software threads are assigned to hardware streams that can be viewed as *logical* superscalar processors. Each stream has an associated program counter, register set, and instruction prefetch buffer. The multistreamed fetch unit brings instructions from the instruction cache to the various prefetch buffers. Within these buffers, instructions are checked for dependencies on previously issued instructions that have not yet completed. The scheduler dispatches instructions that are free of dependencies to the appropriate functional unit provided structural hazards do not exist. Functional units are shared by all streams and return results to the corresponding register file based upon a stream identification tag appended to each instruction. Figure 1 shows a functional block diagram of a multistreamed, superscalar processor.

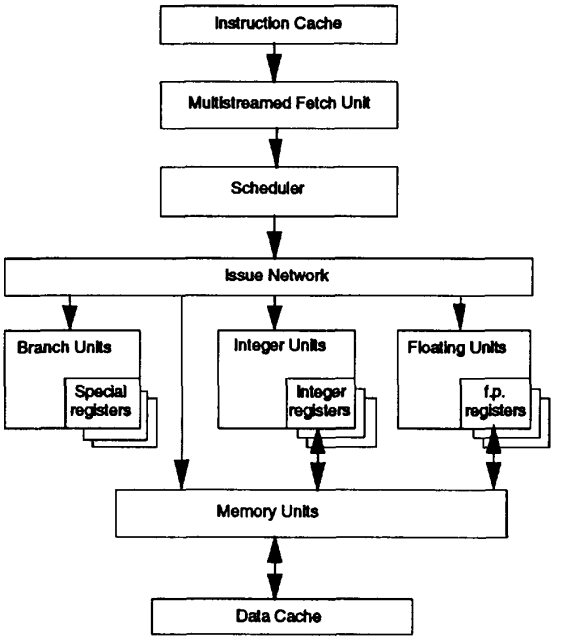


Figure 1. Functional block diagram of a multistreamed, superscalar processor.

Our model of a multistreamed, superscalar processor, is partitioned into 3 major parts: the instruction streams, the scheduler, and the functional units. An instruction stream consists of the context of a process and a buffer containing the next instructions to be executed. The section of the buffer considered by the scheduler for dispatch is called the *stream issue window*. Instructions within the stream issue window can be dispatched out-of-sequence provided no dependencies exist. The scheduler checks, in round-robin fashion, the instructions in each stream's issue window for data and control hazards and then moves the unblocked instructions into the global issue window. The global issue window contains all the instructions in the stream issue windows that are ready to execute, i.e., that have no data or control dependencies. From the global issue window, instructions are dispatched to the appropriate functional units provided no structural hazards are present. All functional units are considered to be fully pipelined and are capable of accepting a new instruction on every cycle. However, different instruction types have different execution times, i.e., an integer divide instruction takes 17 cycles to complete. Figure 2 shows the basic structure of a multistreamed, superscalar processor model. The processor has the capacity to execute N instruction streams simultaneously. The functional units are shared between the streams and multiple copies of certain functional unit can improve overall performance as the number of streams is increased.

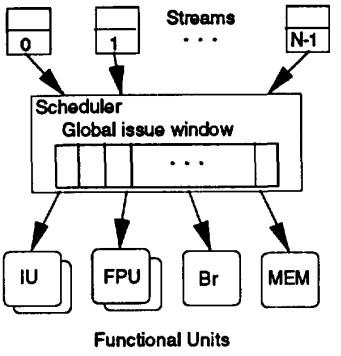


Figure 2. Basic structure of a multistreamed, superscalar processor model.

4 The Analytic Model

The analytic model is a simple probabilistic technique based upon a Markov model. Given a high-level description of the processor hardware and the characteristics of the workload to be executed, the model calculates the expected overall performance. Performance is measured by the overall number of instructions that can be executed per cycle (IPC) by the processor. The performance of individual streams is not calculated by the model since we are attempting to calculate the best possible overall system performance independent of individual stream performance.

We compute the overall performance in IPC using the following formula:

$$IPC = \sum_{w=1}^{G} P_w \cdot IPC_w$$

where P_W is the probability of having w instructions in the global window and IPC_w is the expected IPC measured for a global issue window of size w. The technique is separated into two major parts: one modeling structural hazards and the other modeling the control and data hazards. IPC_w models the effect of structural hazards while P_w is a scaling factor used to model the performance degradation due to the data and control hazards. The rationale behind this division is that the modeling of structural hazards is primarily dependent on the architectural (hardware) configuration while the modeling of control and data hazards is primarily dependent on the workload. In addition, since we employ a Markov chain in our technique, combining the structural hazards as well as dependencies into a single chain results in an extremely large number of states. Our division dramatically simplifies the chain and makes the problem tractable.

The model requires both an architectural description and the workload characteristics as input. The goal in selecting these parameters was to provide a high level of abstraction in describing the processor architecture and ease in obtaining workload characteristics without compromise to the accuracy of the prediction.

4.1 Workload Characterization

The workload of the system is characterized by its runtime instruction mix vector (V), the number of active streams (N), and a data and control dependency degradation factor (A). The instruction mix specifies the percentage of instructions executed in each functional unit type. For example, the instruction mix for the benchmark eqntott is: 39% integer, 36% branch, and 25% memory instructions (e.g., $V = [v_{integer}=0.39, v_{branch}=0.36, v_{memory}=0.25]$). The workload is restricted to a single instruction mix even though more than one stream may be executing. The data and control dependency degradation factor is used to model the effects of these dependencies. A complete description of the degradation factor will be presented in the subsequent sections. Table 1 summarizes the workload characterization parameters.

$V = [v_1 v_2 .. v_T]$	Runtime instruction mix vector. v_i is the probability of an instruction requiring execution in a functional unit of type i.
N	Total number of streams.
A	Data/control dependency degradation factor

Table 1. Workload characterization parameters.

4.2 Architectural Specification

The architecture of the processor is specified by a stream issue window size (S), the number of distinct functional unit types (T), and a functional unit configuration vector (C). The stream issue window is the buffer consisting of instructions that can be issued within a given cycle. We assume that all the stream window sizes are equal, i.e., $S_1 = S_2 = .. S_N = S$. The functional unit configuration vector (C) contains T elements. Each element of the configuration vector, c_i, contains the number of functional unit of type i. For example, the IBM RS/6000, which can issue a maximum of 4 instructions per cycle (integer, floating point, branch, and CR [condition register]), has the following architectural specification: S=4, T=4, $C=[c_{integer}=1, c_{floatingpoint}=1, c_{branch}=1, c_{CR}=1]$. Table 2 summarizes the architectural specification parameters.

S	Stream issue window size.
T	Number of distinct functional unit types.
$C = [c_1 c_2 .. c_T]$	Functional unit configuration vector.

Table 2. Architectural specification parameters.

4.3 Structural Hazard Modeling

In the modeling of structural hazards, we calculate the expected IPC (IPC_w) of a workload for a given processor

configuration as a function of the global issue window size. The global issue window contains all the instructions that are ready to be dispatched to a functional unit, i.e., that have no data or control dependencies. Thus, only the effects of structural hazards are calculated; data and control dependencies are not taken into account.

The state of the global issue window of size w can be represented by a vector M where each element m_i contains the number of instructions of type i in the window. Therefore, the sum of all the elements of M is equal to w:

$$M_w = [m_1 m_2 \ldots m_T],\ w = \sum_{i=1}^{T} m_i .$$

Given a global issue window of size w and T functional unit types, the total number of states for the window is calculated as the combination of $w+T-1$ objects taken $T-1$ at a time or $\binom{w+T-1}{T-1}$.

For a global window state M_w, the number of instructions that can be dispatched in a cycle is determined by the number and type of functional units available. For example, consider a machine with 1 floating point unit, 2 integer units (C=[1,2]), and global window size of 3 as shown in Figure 3. If no floating point and 3 integer instructions are in the global window (M=[0,3]), then only 2 integer instructions can be dispatched in that cycle. In general, for a given functional unit configuration and global window state, the number of instructions of type x dispatched, referred to as i_x, is the smaller of the number of functional units of type x and the number of instructions of type x within the window. We define I_M, the total number of instructions that can be dispatched given the global issue window is in state M, as the sum of i_x over all functional unit types:

$$I_M = \sum_{x=1}^{T} i_x,\ \ i_x = \min(c_x, m_x).$$

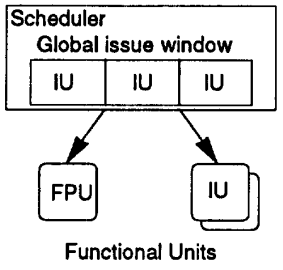


Figure 3. An example of a global window of size 3 containing three integer instructions (M=[0,3]) and a configuration with a floating point unit and two integer units (C=[1,2]).

The expected IPC given a window of w instructions is calculated as the sum of the instructions issued, I_M, times the probability of being in state M, defined as Q_M, for all possible M:

$$IPC_w = \sum_{M} I_M Q_M . \qquad (1)$$

We obtain Q_M by calculating the steady state probabilities of a Markov chain involving the states of the global issue window, M. For a given processor configuration and global window state, all possible next states are determined by the number of instructions that can be executed. For example in Figure 3, two integer instructions are dispatched and one remains. Therefore, the next state must contain at least 1 integer instruction ([2,1], [1,2], and [0,3]). Table 3 lists all states, the possible next states, and the number of instructions dispatched in each state for the example in Figure 3. In general, the next state is dependent on both the current state of the global issue window (M) and the functional unit configuration (C).

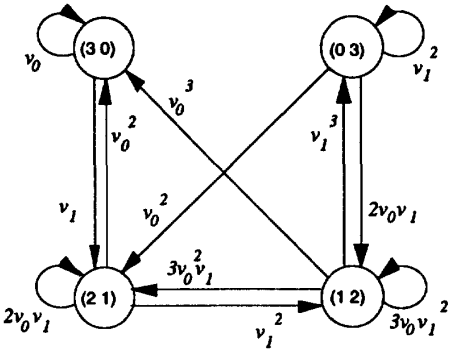
Current State	*Possible Next States*	I_M
[3, 0]	[3, 0] [2, 1]	1
[2, 1]	[3, 0] [2, 1] [1, 2]	2
[1, 2]	[3, 0] [2, 1] [1, 2] [0, 3]	3
[0, 3]	[2, 1] [1, 2] [0, 3]	2

Table 3. State table for the example in Figure 3. I_M is the total number of instructions dispatched for the current state.

The state transition probabilities are based upon the runtime instruction mix for the workload (V). We assume that instructions of different types are uniformly distributed throughout the execution of the workload. For instance, in the example in Figure 3, since 2 instructions are executed in state [0, 3] (0 floating point and 3 integer instructions) the probability of a transition to state [2, 1] (2 floating point and 1 integer instruction) is equal to the probability of filling the global issue window with 2 floating point instructions. This probability, obtained from the runtime instruction mix (V=[v_0, v_1]), is just the square of the floating point probability, v_0. Figure 4 shows the Markov chain model and the state transition matrix STM for the example in Figure 3.

After solving for the steady state probabilities ($Q_{[x,y]}$), we calculate the expected IPC for a window of size w, IPC_W, using equation (1). For the example in Figure 3, the expected IPC for a window size of 3 is calculated as follows:

$$IPC_3 = 1 \cdot Q_{[3,0]} + 2 \cdot Q_{[2,1]} + 3 \cdot Q_{[1,2]} + 2 \cdot Q_{[0,3]} .$$



$$STM = \begin{bmatrix} v_0 & v_1 & 0 & 0 \\ v_0^2 & 2v_0v_1 & v_1^2 & 0 \\ v_0^3 & 3v_0^2v_1 & 3v_0v_1^2 & v_1^3 \\ 0 & v_0^2 & 2v_0v_1 & v_1^2 \end{bmatrix}$$

Figure 4. Markov chain and the state transition matrix (STM) for the example in Figure 3.

4.4 Data and Control Hazard Modeling

In modeling the effects of data and control hazards, an estimated degradation factor is used to scale the performance obtained by the structural hazard models. We assume that the probability of a stream being affected by a data or control hazard is independent of other streams and is represented by a constant, A. To obtain A, the single stream performance, IPC_{sim}, for the benchmark is measured using the actual machine or, in our case, a simulator. Then we take the ratio of IPC_{sim} divided by the IPC_S, from the structural hazard model where S is the stream issue window size, as the value for A.

$$A = \frac{IPC_{sim}}{IPC_S}$$

We assume that when a stream is ready-to-run (i.e., free of any dependencies) it can issue every instruction in its window (S) and, when the stream is blocked, it cannot issue any instructions. For a workload of N streams, the probability that x streams are ready-to-run is assumed to be binomial with probability A. Since each ready-to-run stream dispatches a total of S instructions, the probability of having w instructions ready within the global issue window, P_w, is given by:

$$P_{w=i\cdot S} = \binom{N}{i} A^i (1-A)^{N-i}, \quad i = 1,2,..,N$$

where i is the number of ready-to-run streams and S is the stream window size.

This information is combined with the structural performance estimates (IPC_w) to provide an overall performance estimate as follows:

$$IPC = \sum_{w=S,2S,..,NS} P_w \cdot IPC_w .$$

For instance, in the configuration with three instruction streams (N=3) and a stream window size of 1 (S=1) as in Figure 3, P_w is calculated as:

$$P_0 = (1-A)^3 \qquad P_1 = 3A(1-A)^2$$

$$P_2 = 3A^2(1-A) \qquad P_3 = A^3$$

and the overall IPC is:

$$IPC = P_1 \cdot IPC_1 + P_2 \cdot IPC_2 + P_3 \cdot IPC_3.$$

5 Validation

We validated the analytic technique by comparing its estimations to the results of a multistreamed hardware simulator. We begin this section by describing our simulation environment and benchmark suite. Next, we present the results from both the analytical technique and the hardware simulator. Finally, we quantify the differences between the model and the simulator and discuss the discrepancies.

5.1 The Simulation Environment

Our hardware simulator emulates a multistreamed version of the IBM RS/6000 instruction set architecture [6]. Three different configurations of the machine are studied. Table 4 shows the functional unit count, types, and latencies for the various configurations. The functional units are independent and perfectly pipelined (i.e., capable of accepting an instruction on every cycle). The integer unit executes all integer arithmetic and logic instructions except multiply and divide which have a longer latency and are executed in the multiply and divide units, respectively. Memory instructions execute in the load/store unit where perfect cache hit ratio is assumed. All branch outcomes are predicted correctly and no branch delays exist. The CR (conditional register logic) unit executes the conditional register instructions of the RS/6000 instruction set.

		Configuration		
functional unit	latency	C1	C2	C3
integer	1	1	2	6
float	2	1	2	3
multiply	5	1	1	1
divide	17	1	1	1
CR logic	1	1	1	1
branch	1	1	1	2
load/store	1	1	2	3

Table 4. Configurations Evaluated

In the simulations, the scheduler may dispatch instructions within the stream issue window out-of-sequence in the absence of data hazards. An instruction is removed from the stream issue window as soon as the scheduler dispatches it to a functional unit. The window is then compacted and refilled with new instructions. A stream issue window of 4 instructions is used in the simulations.

The scheduler employs a *fair* scheduling algorithm. In this algorithm, priority is alternated among the streams on every cycle in a round-robin fashion. The scheduler dispatches as many instructions from the high priority stream's window as possible in the current cycle. If *free* slots in functional units still exist, the scheduler attempts to dispatch instructions from the other streams to fill the slots.

To reduce the simulation time, the benchmarks were not run to completion in all cases. In addition, a statistical sampling method was employed to was used further reduce the time of the simulations [2]. However, the time given to each benchmark to run was based on observations of program behavior and in all cases is not less than one hundred million (100 M) simulated cycles.

5.2 Benchmarks

A set of common benchmark programs was used as the workload. The benchmarks include most of the SPEC89 benchmark suite and the Dhrystone and Whetstone benchmarks. Table 5 shows the runtime instruction mixes obtained by the simulator for all the benchmarks.

benchmark	integer	float	multiply	divide	cr logic	branch	memory
dhrystone	46.5	0	0.9	0.3	0.9	20.0	31.5
whetstone	16.6	24.1	1.7	0	1.3	10.8	45.6
eqntott	39.3	0	0	0	0	35.5	25.2
espresso	51.5	0	0.1	0	0.7	21.9	25.8
li	36.4	0	0	0	1.3	21.2	41.0
doduc	17.3	27.8	0.2	0	3.3	10.2	41.2
fpppp	3.8	35.3	0	0	0.4	2.4	58.1
matrix300	1.4	16.2	0.1	0	0	16.7	65.5
spice2g6	43.5	1.9	0.3	0	0.7	19.6	33.8
tomcatv	6.1	45.1	0	0	3.4	9.3	36.1

Table 5. Runtime instruction mixes (percentage).

5.3 Comparison to the Simulation Results

Using the analytic technique, we predicted the overall performance of the benchmarks executing on each configuration and compared the estimates to the simulation results. Table 6 shows the results of the analytic technique, the hardware simulator, and the difference between the two. Graphs of the overall performance (IPC) versus number of active streams for each benchmark and configuration are shown in Figure 5.

		C1			C2			C3		
program	streams	predicted	sim	difference	predicted	sim	difference	predicted	sim	difference
dhry	1	1.68	1.68	0.0%	2.41	2.41	0.0%	2.65	2.65	0.0%
	2	2.05	2.06	-0.5%	3.49	3.45	+1.2%	4.97	4.99	-0.4%
	3	2.13	2.13	-0.0%	3.98	4.00	-0.6%	6.92	6.25	+10.7%
	4	2.15	2.15	0.0%	4.18	4.20	-0.5%	8.19	7.66	+6.9%
whet	1	1.32	1.32	0.0%	1.83	1.83	0.0%	1.93	1.93	0.0%
	2	1.85	1.79	+3.2%	2.94	2.70	+8.8%	3.48	3.62	-3.8%
	3	2.06	1.86	+ 10.6%	3.58	3.39	+5.7%	4.60	4.44	+3.6%
	4	2.14	1.92	+11.5%	3.95	3.63	+8.8%	5.35	5.12	+4.5%
doduc	1	1.57	1.57	0.0%	2.01	2.01	0.0%	2.17	2.17	0.0%
	2	2.13	2.11	+0.8%	3.26	3.44	-5.2%	3.97	4.05	-2.1%
	3	2.32	2.30	+1.0%	3.99	4.16	-4.1%	5.25	5.53	-5.1%
	4	2.39	2.36	+1.4%	4.40	4.50	-2.3%	6.09	6.41	-5.1%
eqntott	1	1.76	1.76	0.0%	2.37	2.37	0.0%	2.75	2.75	0.0%
	2	2.27	2.24	+1.4%	2.75	2.93	-6.0%	4.42	4.91	-10.0%
	3	2.44	2.39	+1.9%	2.81	3.01	-6.6%	5.17	5.72	-9.6%
	4	2.50	2.46	+1.5%	2.82	3.05	-7.6%	5.47	5.87	-6.8%

Table 6. Comparison of the analytic model (predicted) and the simulation results (sim).

program	streams	C1			C2			C3		
		predicted	sim	difference	predicted	sim	difference	predicted	sim	difference
espresso	1	1.59	1.59	0.0%	2.06	2.06	0.0%	2.23	2.23	0.0%
	2	1.88	1.98	-5.1%	3.03	3.14	-3.5%	4.23	4.31	-1.8%
	3	1.93	2.05	-5.9%	3.48	3.45	+1.0%	5.82	5.90	-1.4%
	4	1.94	2.09	-7.2%	3.70	3.53	+4.7%	6.98	6.73	+3.7%
fpppp	1	1.44	1.45	-0.5%	1.93	1.94	-0.3%	2.05	2.05	-0.1%
	2	1.67	1.73	-3.5%	2.78	3.16	-12.0%	3.36	3.89	-13.7%
	3	1.71	1.74	-2.0%	3.15	3.44	-8.5%	4.14	4.90	-15.4%
	4	1.71	1.74	-.7%	3.31	3.48	-5.0%	4.59	5.16	-11.0%
li	1	1.68	1.68	0.0%	2.32	2.32	0.0%	2.67	2.67	0.0%
	2	2.18	2.06	+5.7%	3.46	3.44	+0.6%	4.74	4.88	-2.8%
	3	2.34	2.19	+6.8%	4.02	3.90	+3.1%	6.00	6.19	-3.0%
	4	2.40	2.26	+6.1%	4.34	4.06	+6.9%	6.68	6.79	-1.6%
matrix300	1	1.23	1.23	0.0%	1.58	1.58	0.0%	1.64	1.64	0.0%
	2	1.47	1.50	-2.1%	2.34	2.96	-20.8%	2.72	3.09	-12.0%
	3	1.51	1.51	+0.3%	2.71	2.99	-9.3%	3.41	4.14	-17.6%
	4	1.52	1.51	+0.9%	2.89	3.02	-4.3%	3.85	4.47	-13.8%
spice	1	1.66	1.66	0.0%	2.10	2.10	0.0%	2.19	2.19	0.0%
	2	2.12	2.02	+4.8%	3.24	3.38	-4.0%	4.13	4.23	-2.4%
	3	2.25	2.15	+4.5%	3.89	3.89	0.0%	5.72	5.91	-3.2%
	4	2.28	2.19	+4.2%	4.24	4.10	+3.4%	6.89	7.15	-3.6%
tomcatv	1	1.47	1.47	0.0%	1.91	1.91	0.0%	2.09	2.09	0.0%
	2	1.96	1.74	+12.6%	3.03	3.14	-3.6%	3.71	4.10	-9.6%
	3	2.13	1.78	+19.5%	3.66	3.53	+3.8%	4.83	4.97	-2.8%
	4	2.19	1.78	+22.8%	4.02	3.53	+13.8%	5.56	5.26	+5.8%

Table 6. (cont) Comparison of the analytic model (predicted) and the simulation results (sim).

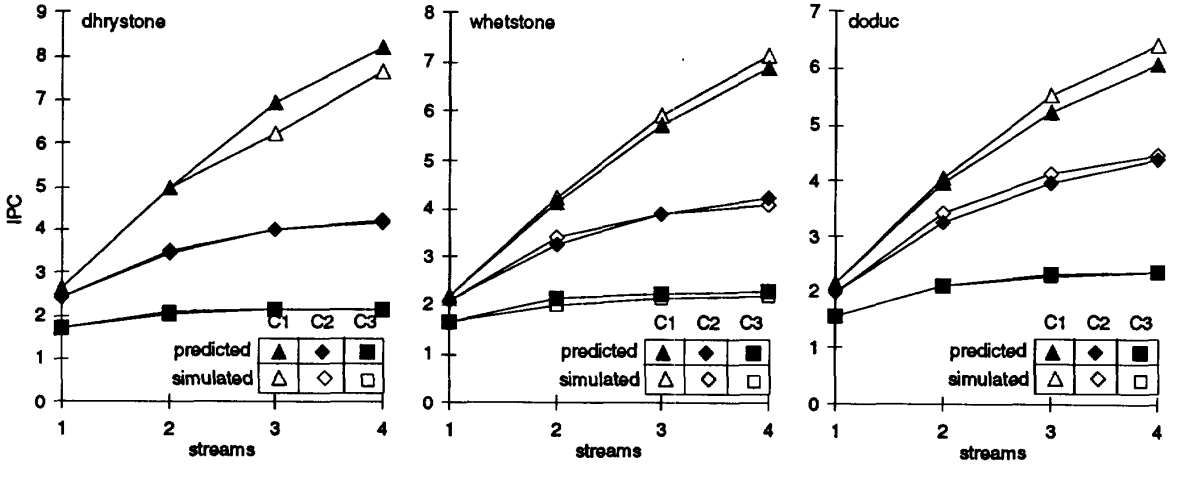
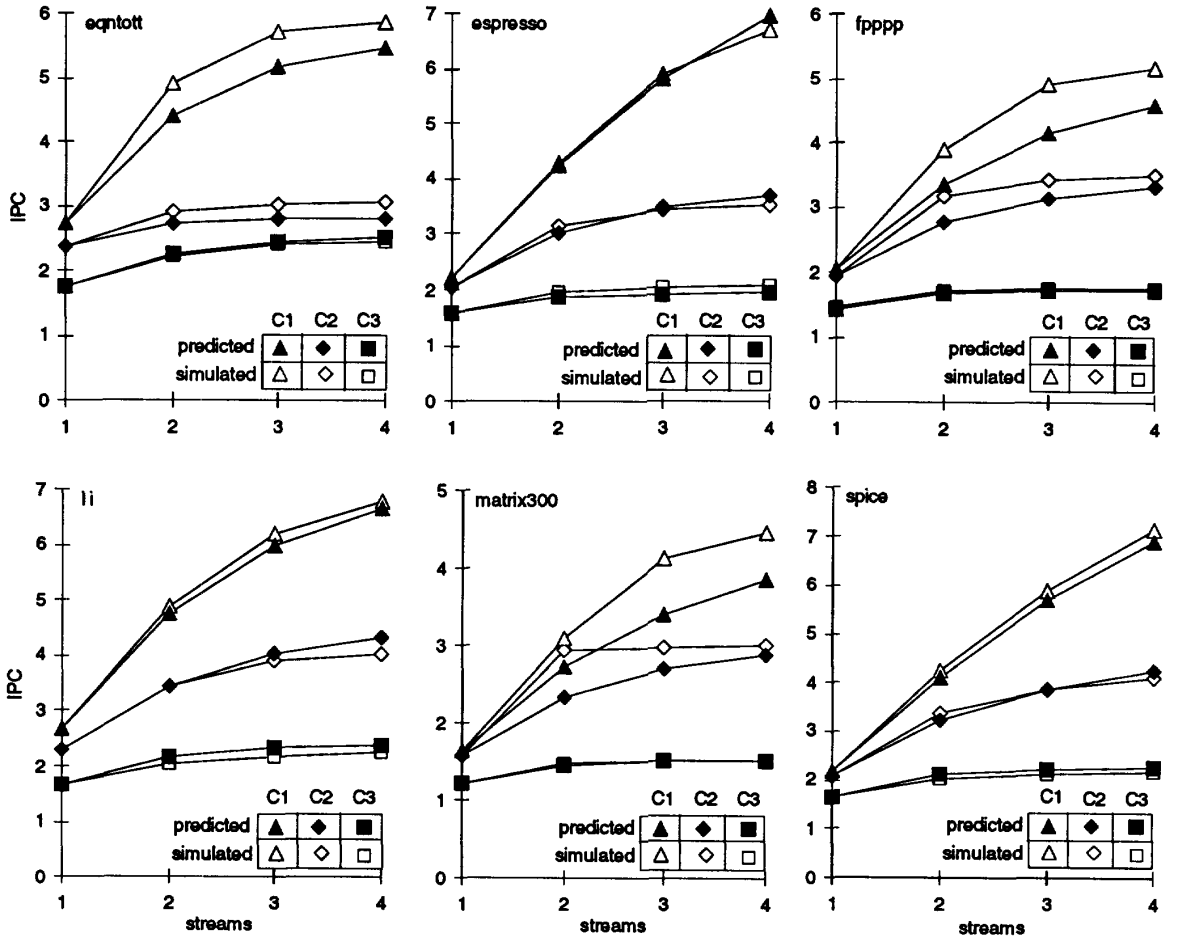


Figure 5. Graphs of IPC versus number of active streams.



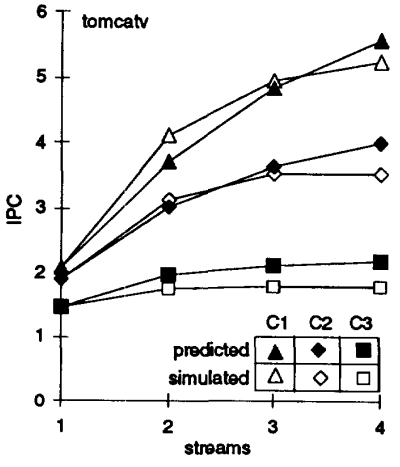


Figure 5. (cont.) Graphs of IPC versus number of active streams.

5.4 Discussion

Our results show that the analytical technique produces estimates very close to those of the simulation results. Over the 120 total estimates, the average deviation from the simulation results was 4.2%. Figure 6a plots the average difference for each benchmark. The average error was below 8% for all benchmarks and below 5% for 6 of the 10.

Figure 6b plots the distribution of the deviation between the analytic model and the hardware simulator. The plots show that 68% of the estimates deviated by less than 5% from the simulation result and just under 90% of the estimates deviated by less than 10%. While the maximum deviation from the simulation was 22.8%, an error over 20% occurred only twice.

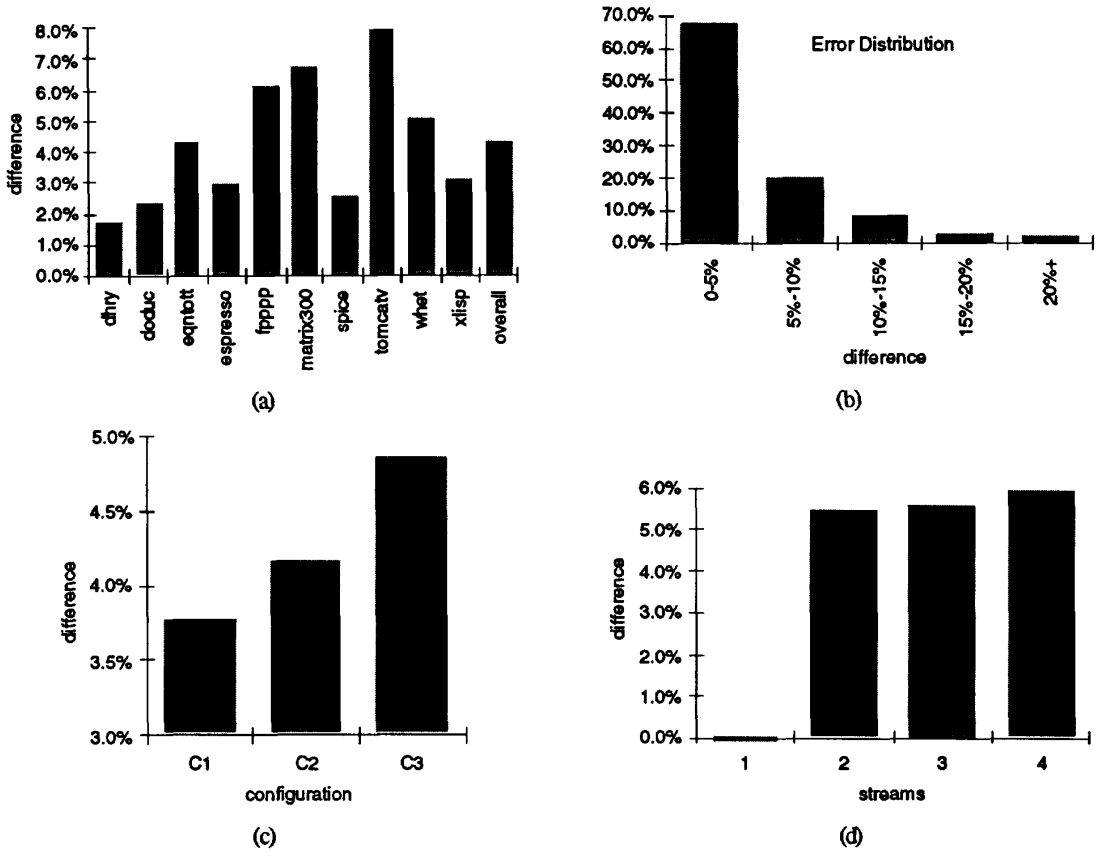


Figure 6. Comparison to the simulation results. (a) chart of the average difference versus program; (b) difference distribution; (c) chart of the average difference versus configuration; (d) chart of the average difference versus number of active streams

On average, the analytic technique produced fairly even results across the various configurations and number of streams employed within a workload. Figure 6c plots the average difference versus the processor configuration and Figure 6d charts the average difference versus the number of active streams.

While the model makes many generalizations about the architecture and workload that may not seem to be representative of most programs, our results show the contrary. A closer examination of the results for each benchmark shows the predicted performance to be very close to the simulation results for all configurations in six of the ten benchmarks (dhrystone, whetstone, doduc, espresso, li, and spice). The others exhibited larger differences from the model estimates for only particular configurations. Tomcatv is an example of a benchmark that the model predicts well for configurations C2 and C3 but over estimates the performance on configuration C1. In contrast, the model under estimates the performance of matrix300, eqntott, and fpppp on configuration C2 and C3 but is very close on configuration C1.

Although the results were collected by running a homogeneous workload (i.e., all streams executing the same benchmark), the model can be extended to model heterogeneous workloads. This is accomplished by using an *averaging* technique over the heterogeneous workload to obtain *average* homogenous workload characteristics (V_{ave} and A_{ave}). These characteristics are entered into the analytical model to obtain performance estimates of heterogeneous workloads.

An examination of the IPC versus the number of streams graphs (Figure 5) reveals the overall performance gain levels out the number of streams increases. In our simulations, this is due to the contention for functional unit resources. The phenomenon is most obvious in configuration C1 where saturation occurs when executing as few as two streams. For the larger configurations (C2 and C3), saturation occurs when executing a larger number of streams. The graphs show that the gain from

multistreaming is largest in configuration C3 where the largest number of functional units is present. In contrast, when the number of functional units is not large enough to support the execution of multiple streams, the gain from multistreaming is minimal. Figure 10 summarizes the performance improvements as the number of streams is increased.

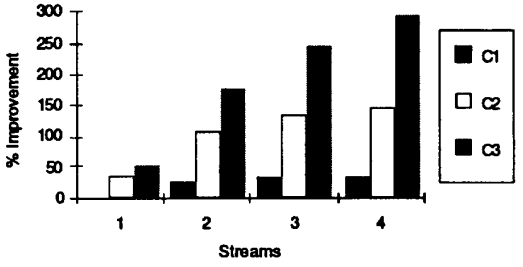


Figure 10. Average Percentage Improvement from Multistreaming, compared to Configuration C1, single-stream.

6 Conclusions

We have presented an analytic technique for evaluating the performance of multistreamed, superscalar processors. Our results demonstrate that the technique produces accurate instructions-per-cycle (IPC) estimates of the overall performance through comparison with a multistreamed, RS/6000 simulator. We found an average deviation from the simulation results was just above 4% when predicting the performance of a benchmark suite that included most of the SPEC89 programs.

The analytical technique provides a quick way to examine the many variations of an architecture at a high level of abstraction. The technique only requires simple descriptions of the workload and architectural configuration as input parameters. These parameters are easily measured or estimated using tools that are commonly available. In addition, the simplicity of the model makes it easy to implement and much faster to execute than a hardware simulator.

As the trend moves towards integrating more functional units within the processor, designers face the challenge of utilizing these additional resources effectively. Superscalar processors are not capable of exploiting the additional functional units due to the inherent limit of instruction-level parallelism within a single instruction stream. Multistreaming provides a technique to overcome the superscalar limitations. Integrating multistreaming within superscalar architectures is an effective method for maximally exploiting the additional functional units emerging in new processors. Our results show that a multistreamed, superscalar processor executing 4 streams can improve the overall machine throughput by factor of 3 over an equivalent single stream machine (Figure 10).

References

[1] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," *Proc. of the 17th Symposium on Computer Architecture*, May 1990, pp. 104-114.

[2] T. M. Conte, "Systematic Computer Architecture Prototyping," PhD Thesis, Electrical Engineering,

[3] M. J. Flynn, A. Podvin, and K. Shimizu, "A Multiple Instruction Stream Processor with Shared Resources," *Parallel Processor System*, C. Hobbs, Washington D.C., Spartan, 1970.

[4] R. H. Halstead and T. Fujita, "MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing," *Proc. of the 15th Symposium on Computer Architecture*, June 1988. pp. 443-451.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishers, 1990.

[6] IBM AIX Version 3.2 for RISC System/6000 Assembler Language Reference. International Business Machines Corp., Austin, Texas, Jan. 1992.

[7] W. J. Kaminsky and E. S. Davidson, "Developing a Multiple-Instruction-Stream Single-Chip Processor," *IEEE Computer Magazine*, Dec. 1979.

[8] J. S. Kowalik, ed., *Parallel MIMD Computation: HEP Supercomputer and its Applications*, MIT Press, 1985.

[9] M. D. Nemirovsky, "DISC, A Dynamic Instruction Stream Computer," Ph.D. Dissertation, University of California, Santa Barbara, September 1990.

[10] M. Serrano, M. D. Nemirovsky, and R. C. Wood, "A Study on Multistreamed Superscalar Processors," Technical Report #93-05, Department of Electrical and Computer Engineering, University of California, Santa Barbara, March 1993.

[11] C. A. Staley, "Design and Analysis of the CCMP: A Highly Expandable Shared Memory Parallel Computer," Ph.D. Dissertation, University of California, Santa Barbara, August 1986.

[12] J. E. Thornton, "Parallel Operation in the Control Data 6600," Proceedings-Spring Joint Computer Conference, 1964.