Effects of Processor Parameter Selection on Simulation Results

Joshua J. Yi and David J. Lilja
Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota - Twin Cities
Minneapolis, MN 55455
{jjyi, lilja}@ece.umn.edu

Abstract

Due to cost, time, and flexibility constraints, simulators are needed to explore the design space when developing a new processor architecture as well as when evaluating the performance of new compiler-based and microarchitectural mechanisms. However, improperly choosing the processor parameters, such as the cache size, the associativity, the number of functional units, and so forth, can significantly affect the simulation results, independent of any new mechanism being evaluated. As a first step in developing a methodology to appropriately choose the processor parameters to be used in a simulation, we quantify the effect of a select group of ten key parameters on the total execution time (measured in simulated cycles) using the statistical analysis of variance (ANOVA) technique. Our results show that, for the commonly used SimpleScalar simulator, the number of reorder buffer entries, the L1 D-Cache associativity and size, and the memory latency account for approximately 75% of the total observed variation in the execution time. Conversely, the branch predictor has surprisingly little effect, only about 1%. In addition, we find that interactions between parameters also have little effect on the total execution time. This work clearly demonstrates that poor choices of simulated processor parameter values, or even a single poor choice, can significantly affect the simulation results and thereby lead to erroneous conclusions about the processor or mechanism being simulated.

1 Introduction

Simulators are an extremely valuable tool for computer architects. They reduce the cost and time of a project by allowing the architect to quickly evaluate different processor implementations without having to fabricate a chip each time. Additionally, a simulator allows the architect to easily determine the expected performance improvement of a new compiler-based or microarchitectural mechanism.

However, due to this large dependence on simulators, one must carefully choose the simulated processor parameters when evaluating the performance of a new architecture or the effect of a new mechanism. Otherwise, the simulation results may not accurately reflect the true potential of the new architecture or mechanism. However, the problem in appropriately selecting

parameters is that it is not known which parameters have the greatest impact on the processor's execution time.

To illustrate the importance of choosing "good" parameter values for a simulation, consider the results shown in Table 1. This table shows the speedup due to value reuse [Sodani97] on a 4-way issue superscalar processor when using three different reorder buffer (ROB) sizes (16, 32, and 64 entries). The sim-outorder simulator from the SimpleScalar tool suite [Burger97] was used as the base simulator. For each ROB size, the number of load-store queue (LSQ) entries was fixed to half the number of ROB entries. The other processor parameters did not change for all simulations. For all simulations, the base processor, i.e. the processor without value reuse, had the same number of ROB entries as the processor with value reuse.

Benchmark	16 ROB Entries	32 ROB Entries	64 ROB Entries
164.gzip	1.96	2.31	1.72
175.vpr-Place	7.10	7.64	7.39
175.vpr-Route	8.11	11.32	10.27
177.mesa	20.23	29.80	24.46
179.art	0.74	0.67	0.75
181.mcf	3.09	4.42	4.72
183.equake	10.30	14.29	16.72
188.ammp	0.86	1.20	1.25
197.parser	6.98	6.86	6.42
255.vortex	9.13	10.26	9.98
300.twolf	6.90	7.85	7.20

Table 1: Percent Speedup Due to Value Reuse for Different Reorder Buffer (ROB) Sizes

The key result here is that there is no clear pattern to the speedups in Table 1. For some benchmarks, such as 179.art, the speedup is essentially the same for all ROB sizes. For others, such as 177.mesa, the speedup increases when the number of ROB entries increases from 16 to 32, but then decreases when the number of ROB entries increases from 32 to 64. Finally, for the remainder, the speedup continually increases or decreases with an increasing number of ROB entries. Therefore, while these three ROB sizes may be reasonable, the actual size that is used can significantly affect the speedup for this microarchitectural mechanism.

The key point to be drawn from Table 1 is that a single processor parameter (or an interaction between two or more parameters) can have a significant impact (positive or negative) on the speedup results. Choosing the value of a parameter to be too high or too low can favorably or adversely affect the simulated speedup. However, testing the effect of each parameter and the effect of each interaction between parameters is a tremendously time intensive process (to which the authors of this paper can attest). Therefore, it is not practical from a time standpoint to determine the "best" value for each parameter from a range of "reasonable" values. Note that value reuse is only one representative microarchitectural mechanism. Other mechanisms, such as prefetching and value prediction, could show even more dramatically the impact of parameter value selection on the speedup.

To produce more realistic and accurate simulation results, it is imperative to understand the effect of the different parameters on the execution time, the effect of the interactions between parameters, and, finally, which parameters have the most impact (i.e. need to be chosen with the most care). Furthermore, these results are not only germane to processor simulation, but also to processor design in general.

To address this gap in knowledge, we determined the effect of what we judged to be the ten most important processor parameters in the SimpleScalar architecture. Specifically, we examined the effect of the branch predictor; the number of Integer ALUs; the number of ROB entries; the L1 D-Cache size, block size, and associativity; the L2 Cache size, associativity, and latency; and the main memory latency.

This paper makes the following contributions:

1. It determines the effect of the ten most significant processor parameters, and their interactions, on the simulated execution time (cycles) by using sim-outorder from the SimpleScalar tool suite.
2. It determines which processor parameters and interactions have the greatest impact on the simulated execution time.
3. It makes some precise recommendations about simulation and performance testing methodology for computer system hardware and software designers.

The remainder of this paper is organized as follows: Section 2 describes some related work while Section 3 describes which processor parameters were considered in this paper and the final values that were used. Section 4 then describes the analysis of variance (ANOVA) technique used to determine the effect of each parameter and their interactions. Section 5 presents and explains the results that were produced with the ANOVA technique while Section 6 describes some future work. Finally, Section 7 concludes and gives some recommendations concerning simulation methodology.

2 Related Work

Black and Shen [Black98] described the types of errors that are common to performance modeling and a method of validation that iteratively improves the accuracy of the performance model, as compared to the actual processor. Their method of validation compared the cycle count that was produced by a simulator that was targeted at a specific architecture (in this case the Power PC 604) against the cycle count that was produced by the actual hardware. If the cycle counts differed, then there was at least one error in the simulator.

Their results show that all three types of errors (modeling, specification, and abstraction) were still present in their simulation model, even after a long period of debugging. Some of these errors could be revealed only after comparing the performance model to the actual processor. Furthermore, the magnitude of the IPC error did not continually decrease to zero throughout the validation process. Instead, it fluctuated through each progressively more accurate model before finally settling in within the desired error range. The desired error range is the amount of error (in this case, the percent difference in the IPC between the simulator and the actual hardware) that the designer is willing to tolerate. This previous work showed the need for extensive, iterative validation before the results from a performance model can be trusted.

Gibson et al [Gibson00] described the types of errors that were present in the FLASH simulator when compared to the implemented FLASH processor. To determine which errors were present in the FLASH simulator, they compared the execution time of the FLASH simulator against the execution time of the FLASH processor. If the execution times differed,

then there was at least one error in the simulator. They tested several versions of their simulator to evaluate the tradeoff of a faster, but less complex simulator versus a slower, but more complex simulator in terms of simulator accuracy.

Their results showed that most simulators can accurately predict the architectural trends, but only if all the important components have been accurately modeled. However, the caveat is that it is very hard to identify the important components without having the actual processor as a reference. Their results also showed that a faster, less complex simulator that uses a scaling factor for the results often did a better job of predicting a processor's performance than a slower, more complex simulator. Finally, their results showed that the margin of error (the percentage difference in the execution time) of some simulators was more than 30%, which is higher than the speedups that are often reported in other papers. This work is significantly different from other previous work that examined simulator validation in that it did not model the simulator after an existing target architecture, but rather implemented the simulator first and then fabricated the processor itself.

Desikan et al [Desikan01] measured the amount of error that was present in an Alpha version of SimpleScalar. The amount of error was the difference in the execution time of the simulator and the execution time of the processor itself. They first validated the functionality and performance of each major component (such as the fetch stage or memory hierarchy) in the simulator with microbenchmarks before determining the difference in the execution times between the simulator and the processor using macrobenchmarks.

They found that the simulators that model a generic machine (such as SimpleScalar) generally reported higher IPCs than simulators that were validated against a real machine. In other words, a simulator that does not target a specific architecture will generally report higher IPCs for the same benchmarks as compared to a validated simulator that targets a specific architecture. On the other hand, unvalidated simulators that targeted a specific machine usually *underestimated* the performance. Finally, they recommended that processor parameters should be chosen against "reference machines, common models, or communal parameter sets." One of

the contributions of our study is to provide an analytical background for choosing a set of communal parameters by determining the effect of the major parameters.

In conclusion, previous work in this area focused primarily on performance analysis and simulation validation. In contrast, this study focuses primarily on why choosing processor parameters is exceedingly important and the impact that those parameters can have on the simulated processor's reported execution time. Therefore, this paper complements previous work by examining the potential "error" of improper parameter value selection and its impact on the simulated execution time.

3 Candidate Parameters

To determine the effect of various processor parameters and their interactions, we used sim-outorder from the SimpleScalar tool suite (version 3.0, PISA). This simulator has several processor parameters that can be easily changed. Table 2 lists the 38 parameters that we initially considered analyzing, i.e. to determine their effect on the execution time (cycles).

Number	Parameters
1 – 5	L1 I-Cache Size, Associativity, Block Size, Replacement Policy, Latency
6 – 7	Branch Predictor Type and Configuration
8	# of Instruction Fetch Queue Entries
9 – 11	Decode, Issue, Commit Widths
12	# of ROB Entries
13	# of Int ALUs
14	# of FP ALUs
15	# of Int Multipliers/Dividers
16	# of FP Multipliers/Dividers
17	Functional Unit Latencies
18	Degree of Pipelining in the Functional Units
19	# of LSQ Entries
20	# of Memory Ports
21 – 25	L1 D-Cache Size, Associativity, Block Size, Replacement Policy, Latency
26 – 30	L2 Cache Size, Associativity, Block Size, Replacement Policy, Latency
31 – 32	Memory Latency, First Block; Memory Latency, Following Blocks
33	Memory Bus Width
34 – 38	Instruction and Data TLB Entries, Page Size; TLB Latency

Table 2: Initial List of Processor Parameters

Simultaneously trying to measure the effect of 38 different parameters and all their possible combinations is an intractable problem. For example, in this paper, we ultimately considered only 10 parameters. Simulating all possible combinations of those 10 parameters required 4 months of time, even though we ran 16 simulations at a time for the entire 4-month duration. Testing the effect of a single additional parameter at least *doubles* the simulation time. Performing the same set of simulations for the parameters shown in Table 2 at the same rate would have required approximately 20 *billion* years, or more time than the sun will shine. Therefore, to reduce the simulation time to a manageable level, we had to reduce the number of free parameters (i.e. parameters with non-fixed values), from among those that could be easily altered, to the most important dozen or so. The parameters that we chose to eliminate – which was accomplished by fixing their values – along with our reasons for eliminating them are described below.

3.1 Reducing the Number of Free Parameters From 38 to 21

The first parameters that we chose to eliminate from further consideration were the decode, issue, and commit widths. These parameters were eliminated by setting their values to 4. We chose to eliminate these parameters for two reasons. First of all, we assumed that the effect of each processor parameter would remain roughly the same as the issue width increases. While the processor width could affect how much impact each parameter had, the impact for each parameter probably would not radically differ for each width because the parameter values are scaled appropriately.

Secondly, substantial previous work [Bannon97, Christie96, Edmondson95, Horel99, Kessler98, Kessler99, Kumar97, Leiholz97, Matson98, Normoyle98, Papworth96, Silc99, Sima97, Tremblay96, and Yeager96] described the architecture of several 4-way issue processors that were implemented along with the values for many important parameters. Therefore, we were able to gather a set of values that accurately reflected the actual values that are used in commercially available processors, which gave us a good initial starting point.

We also decided to eliminate the L1 I-Cache parameters, the TLB parameters, and all the cache replacement policies. While Gibson et al [Gibson00] found that the TLB had a significant impact on the processor performance, we eliminated both the L1 I-Cache and the TLB parameters because we considered these parameters to be less important in comparison to the remaining ones. The L1 I-Cache size was set to 32KB, the associativity to 4-way, the block size to 32 bytes, and the latency to 1 cycle. The instruction TLB and the data TLB were both set to their default values (instruction: 64 entries and a 4K page size; data: 128 entries and a 4K page size; and a 30 cycle latency for both). The replacement policies for all caches (L1 I/D and L2) were set to least-recently used (LRU). The values for these parameters were set to values that were slightly more aggressive than was what found in our survey of state-of-the-art commercial processors or – if we could not find definitive documentation – to what we thought were reasonable parameters.

Finally, we eliminated the functional unit latency and degree of pipelining as parameters under consideration. The degree of pipelining is the minimum number of cycles that separates the start of two instructions on the same functional unit. The first reason for eliminating these two parameters was that we considered these parameters to be tightly linked to the SimpleScalar architecture. Secondly, we thought that these parameters, while important, were not as important as any of the remaining 21 parameters, especially since they are so highly correlated to the architecture. For each functional unit, we used the values shown in Table 3.

Functional Unit	Latency	Cycles Between Instructions
Int ALU	1	1
Int Multiply	3	1
Int Divide	19	20
Floating-Point ALU	2	1
Floating-Point Multiply	4	1
Floating-Point Divide	12	12
Floating-Point Square Root	24	24

Table 3: Functional Unit Latencies and Degree of Pipelining

3.2 Reducing the Number of Free Parameters From 21 to 13

In the second round of parameter selection, we eliminated the following as free parameters: the branch predictor configuration, the number of FP ALUs, the number of Integer and FP Multipliers, the L1 D-Cache latency, the L2 Cache block size, the memory bus width, and the memory latency of the blocks following the first block. These parameters were eliminated as free parameters because they: 1) were associated with another parameter (branch predictor configuration), 2) could be linked to another parameter (number of FP ALUs, number of Integer and FP Multipliers, and memory latency of the following blocks), 3) could be set to a commonly accepted value (L1 D-Cache latency), or 4) were judged to have less impact on the execution time than the remaining parameters (L2 Cache block size and the memory bus width). Since the branch predictor configuration is dependent on the branch predictor itself, we set each configuration based on the branch predictor (perfect or 2-level).

After looking at the instruction mixes of the benchmark programs, we set the number of Integer Multipliers equal to half the number of Integer ALUs. We also set the number of FP ALUs and Multipliers equal to the number of Integer ALUs and Multipliers, respectively. While this could overstate the impact of the Integer ALUs and Multipliers on the execution time, we thought that the amount of floating-point computation was low enough to avoid this problem. The memory latency of the blocks following the first block was set to 2% of the latency of the first block. Therefore, if the first block's latency was 50 cycles, all subsequent blocks will have a latency of 1 cycle. The 2% multiplier was chosen since it generated reasonable values for the memory latency of the following blocks.

Since the L1 D-Cache latency in most of the processors that we surveyed was a single cycle, we thought it was important to set the L1 D-Cache latency to 1 cycle and set the other L1 D-Cache parameters (size, associativity, and block size) around that latency.

Finally, we set the L1 D-Cache and L2 Cache block sizes based on the typical L1 D-Cache and L2 Cache block sizes (32 and 64 bytes, respectively) in commercial processors. Accordingly, the memory bandwidth was set to 64 bytes to allow an L2 cache block to be transferred from memory in a single cycle.

3.3 Reducing the Number of Free Parameters From 13 to 10

After the second round of parameter selection, we were left with 13 parameters. From previous experience, we wanted to trim our list to only 10 parameters to limit the total simulation time to a manageable amount of time. After reviewing our list, we felt that the following three parameters would have the *least* impact: the number of instruction fetch queue (IFQ) entries, the number of LSQ entries, and the number of memory ports. However, while we thought these three parameters would have least amount of impact, setting them to unreasonable values could affect our final results since they are still very significant. Therefore, to determine what these values should be, we ran several sets of preliminary simulations.

Our initial thought was to set the number of IFQ entries to 16, 32, or 64. Since the instruction fetch queue decouples the fetch stage from the rest of the processor, to test the efficacy of each IFQ size, we configured the fetch stage to fetch instructions at a very high rate (perfect branch prediction, large and highly associative L1 I-Cache, etc.) while varying the rate at which the processor consumed the instructions (ROB size, D-Cache size and associativity, etc.). Our results showed that the IFQ was full 68%, 48%, and 25% of the cycles for a 16-entry, 32-entry, and a 64-entry IFQ, respectively. Therefore, balancing the cost of a larger IFQ against the higher fraction of cycles that the IFQ is full for a smaller IFQ, we chose a 32-entry IFQ.

Our intuition for the number of LSQ entries was that it should be a 1:1, 1:2, or 1:4 ratio of the number of ROB entries. To evaluate those three ratios, we set the number of ROB entries to 32 and the number of LSQ entries to be 32, 16, and 8. Then, to determine the performance, in terms of the fraction of cycles in which the LSQ was full, we varied the L1 D-Cache size and associativity. The different L1 D-cache sizes and associativities varied the L1 D-cache miss rate, which then directly affected the number of load and store instructions in the LSQ. When the ratio was 1:1, the LSQ was never full. When the ratio was 2:1, the LSQ was full 11% (median) of the time. Finally, when the ratio was 4:1, the LSQ was full 62% of the time. Therefore, factoring in the cost of a larger LSQ, we chose a 2:1 ratio.

Finally, to determine the number of memory ports that should be used, we evaluated the effect of 1, 2, and 4 memory ports on the execution time when key memory sub-system parameters were varied. These results showed that the "knee" of the curve for almost all benchmarks and all memory configurations was at 2 memory ports. Thus, this is the number of memory ports that we used.

Table 4 summarizes the parameters that were set to constant values for all of the remaining simulations and their final values while Table 5 shows which parameters were chosen to be varied.

Parameters	Final Values
L1 I-Cache Size, Associativity, Block Size, Replacement Policy, Latency	32KB, 4-Way, 32 Bytes, LRU, 1 cycle
# of IFQ Entries	32
Decode, Issue, Commit Width	4-Way
# of FP ALUs	Equal to Number of Int ALUs
# of Int Multipliers	Half the Number of Int ALUs
# of FP Multipliers	Equal to Number of Int Multipliers
Functional Unit Latency	See Table 3
Degree of Pipelining in Functional Units	See Table 3
# of LSQ Entries	Half of ROB Entries
# of Memory Ports	2
L1 D-Cache Replacement Policy, Latency	LRU, 1 Cycle
L2 Cache Block Size	64 Bytes
Memory Bus Width	64 Bytes
TLB Entries; Page Size, Latency	I: 64, D: 128; 4KB, 30

Table 4: Final List of Fixed Parameters (Parameters NOT Under Test) and Their Constant Values

4 Analysis of Variance (ANOVA)

After determining which parameters to vary, we needed to decide how many values to test for each parameter and what those values should be. However, testing several values for each parameter would exponentially increase the simulation time. For instance, simulating the effect of 3 values for each of the 10 parameters, instead of just 2, increases the number of simulations from 2^{10} simulations per benchmark to 3^{10} simulations per benchmark. Therefore, in addition to limiting the number of parameters that can be varied, restricting the number of values

for each parameter reduces the number of simulations that are necessary. Since we have already decided on our final list of parameters, we must determine how many values are needed for each parameter and what those values should be. To determine that, we use the ANOVA technique to guide our decision-making [Lilja00].

The ANOVA technique is a statistical technique that allows one to separate the effect of a single parameter (or the combination of multiple parameters) on the output, which in our case was the total execution time in cycles. There is no restriction on the number or type of values for each parameter. For example, the values for a parameter could be "On" or "Off," as could be the case for prefetching. Similarly, the values could be discrete numbers such as 4, 8, and 16, as could be the case for the issue width. The final result of the ANOVA technique is vector of percentages. Each percentage represents the contribution of that parameter or combination of parameters on the total variation in the output. For instance, a parameter that accounts for 90% of the variation in the execution time has a much larger impact on the execution time than a parameter that only accounts for 10% of the variation in the execution time.

A specific version of the ANOVA technique, 2^m experimental design, restricts the number of values for each parameter to 2 (m is number of parameters under test). In this case, the values for each parameter typically should represent the limits of the values for the parameter. In other words, the minimum value should represent the lower bound of "acceptable" values while the maximum value should represent the upper bound. For example, the minimum and maximum issue widths for a state-of-the-art processor could be 2-way and 8-way, respectively. The "minimum" and "maximum" issue policies for a superscalar processor could be in-order and out-of-order, respectively. However, the fundamental point in the choice of endpoint values is that the *likely* values should be between the endpoints.

Table 5 shows the final list of parameters under test and the minimum/off and maximum/on values we used for each. For the branch predictor, we chose the two-level branch predictor as the minimum value; this branch predictor could be considered as good, but not great. For the maximum value, we chose the perfect branch predictor, which is best theoretical branch predictor possible.

Parameters	"Off" Value	"On" Value
Branch Predictor	2-Level	Perfect
Number of Int ALUs	2	4
Number of ROB Entries	16	64
L1 D-Cache Size	4KB	64KB
L1 D-Cache Block Size	8 Bytes	32 Bytes
L1 D-Cache Associativity	1-Way	4-Way
L2 Cache Size	128KB	1024KB
L2 Cache Associativity	1-Way	8-Way
L2 Cache Latency	25 Cycles	5 Cycles
Memory Latency, First Block	200 Cycles	50 Cycles

Table 5: Final List of Parameters Under Test and Their "Off" and "On" Values

For the number of Integer ALUs, we choose the endpoint values to be 2 and 4. While these endpoints accurately represent the number of Integer ALUs that are present in state-of-the-art commercial processors, there are two important facts about SimpleScalar that affect how many Integer ALUs are needed. The first fact is that the version of sim-outorder that we used can start the execution of only 4 (or fewer) instructions per cycle. Therefore, even though an instruction could be ready for execution and its corresponding functional unit could be free, the instruction may not be executed if that cycle's "execution width" has been exhausted. Secondly, each SimpleScalar Integer ALU is a completely general purpose Integer ALU. Therefore, it is not necessary to route certain instructions, such as shifts, to specific Integer ALUs; any Integer ALU can execute that instruction. These two facts contribute to how many Integer ALUs are needed.

We chose the two values for the number of ROB entries to be 16 and 64. Since a 16-entry ROB is smaller than the ROBs that are in commercial processors, it meets the requirements for the minimum value. On the other hand, a 64-entry ROB is not too much larger than the ROBs in commercial processors. However, in SimpleScalar, the ROB access time is only a single cycle. Therefore, since it is somewhat unreasonable to assume that a 128-entry or larger ROB will still have a single cycle access time, we set the maximum ROB size to 64-entries.

We also used similar reasoning when choosing the L1 D-Cache size and associativity. Since the hit latency of the L1 D-cache was set to 1 cycle, we tried to choose maximum values

for those two parameters that would still allow for a single cycle hit time. Therefore, for those parameters, we chose the maximum values to be 64KB and 4-Way associativity. While these values are probably high, we optimistically assumed that they would fulfill the single cycle access time requirement. For the minimum values, we chose 4KB and direct-mapped. All four values were chosen based on our survey of commercial processors, as discussed in Section 3.

For the remaining parameters (L1 D-Cache block size, L2 Cache size, L2 Cache associativity, L2 Cache latency, and the First Block Memory Latency), we based our minimum and maximum values on a combination of our survey of commercial processors, the values that were commonly used in previous simulation-based studies, and our best estimates when no published material was available.

5 Results

Table 6 shows the benchmarks and input sets that we used in this study. With the exception of 300.twolf, all the SPEC 2000 benchmarks used a reduced input set. These reduced

Benchmark	Suite	Type	Input Set
099.go	SPEC 95	Integer	Train
124.m88ksim	SPEC 95	Integer	Train
126.gcc	SPEC 95	Integer	Test
129.compress	SPEC 95	Integer	Train
130.li	SPEC 95	Integer	Train
132.ijpeg	SPEC 95	Integer	Test
134.perl	SPEC 95	Integer	Test
164.gzip	SPEC 2000	Integer	Reduced Small
175.vpr	SPEC 2000	Integer	Reduced Medium
177.mesa	SPEC 2000	Floating-Point	Reduced Large
179.art	SPEC 2000	Floating-Point	Reduced Large
181.mcf	SPEC 2000	Integer	Reduced Medium
183.equake	SPEC 2000	Floating-Point	Reduced Large
188.ammp	SPEC 2000	Floating-Point	Reduced Medium
197.parser	SPEC 2000	Integer	Reduced Medium
255.vortex	SPEC 2000	Integer	Reduced Medium
300.twolf	SPEC 2000	Integer	Test

Table 6: Selected Benchmarks Characteristics

input sets were developed to produce characteristics that were similar to the characteristics of the reference input set [KleinOsowski00]. All benchmarks were compiled at optimization level O3 using the SimpleScalar version of the gcc compiler. All benchmarks were run to completion.

Since 134.perl and 175.vpr use two "sub-input" sets (Jumble and Primes for 134.perl and Place and Route for 175.vpr) for each input set, the results for each of these sub-input sets will be listed separately.

5.1 The Effect of Single Parameters

Figure 1 shows the effect, in percent, of each parameter on the execution time. The rightmost bar shows the average effect for each parameter across all of the benchmark programs. The effect that a parameter has is what percentage of the variation of the execution time is due to that parameter. Therefore, the more effect that a parameter has, i.e. the larger percentage accounted for by the parameter, the more sensitive the execution time is to changes in that parameter.

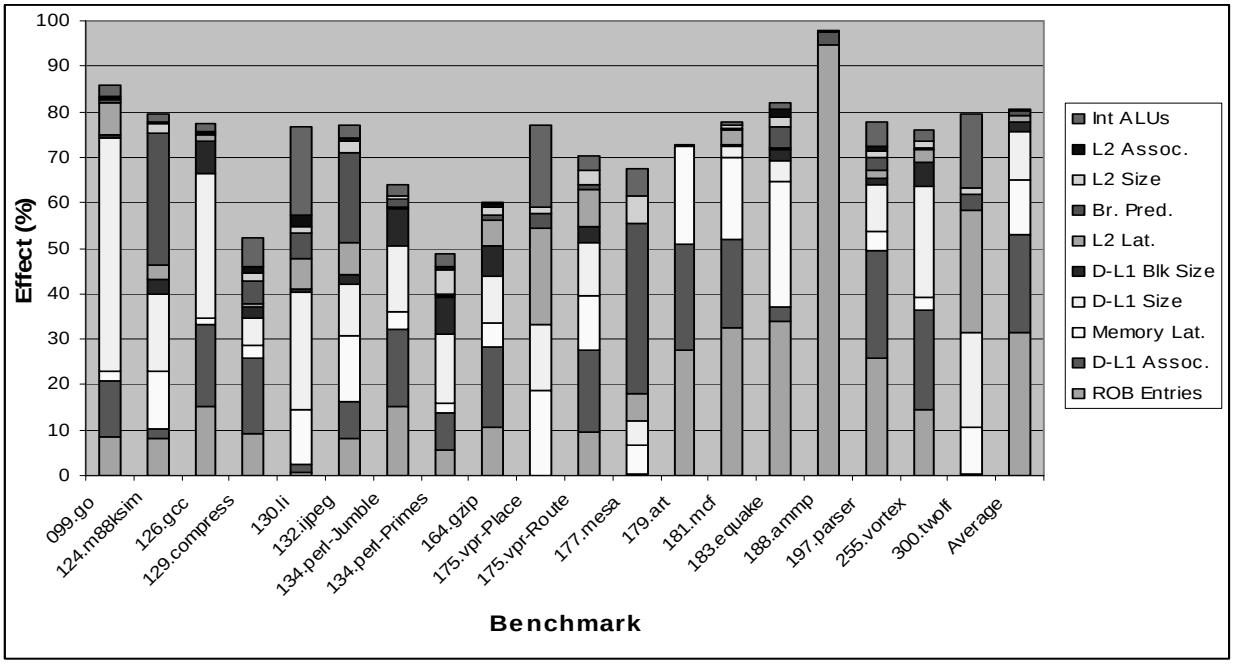

Figure 1: The Effect (Percent) of Single Parameters on the Total Variation in the Execution Time (Cycles).

Several key results are found in this figure. The first result is that four parameters account for an average of 75.71% of the all variation in the execution time. These four parameters are, in decreasing order of effect: the number of ROB entries (31.52%), the L1 D-Cache associativity (21.51%), the memory latency (11.93%), and the L1 D-Cache size (10.75%). The number in parentheses for each parameter is the weighted average effect that each parameter has across all benchmarks.

While these four parameters account for 75.71% of all the variation in the execution time, the remaining six single parameters account for only 4.99% of the variation. This result provides incontrovertible evidence that some parameters have a much greater effect on the execution time than other parameters and that those parameters have much more effect individually than when they interact. Section 5.2 discusses the effect of parameter interaction in more depth.

In two benchmarks, one of these four parameters has an enormous effect on the execution time. In 099.go, the L1 D-Cache size accounts for 51.03% of the variation observed in the execution time. In 188.ammp, the number of ROB entries accounts for a startling 94.67% of the variation in the execution time. Therefore, for these two benchmarks, the execution time is highly correlated to the actual values for each of these two parameters. As a result, one has to be exceptionally careful when choosing the L1 D-Cache size and number of ROB entries for 099.go and 188.ammp, respectively.

Part of the reason why the number of ROB entries and the memory latency are so important may be due to the fact that these parameters are used as an explicit reference value. In other words, since the number of LSQ entries is held at half of the number of ROB entries and the latency of the following blocks is held at one-fiftieth of the latency for the first block, these two relationships are partially responsible for the effect that the number of ROB entries and the memory latency produce. That is, the number of ROB entries and the memory latency of the first memory block may account for larger percentages of the variability in the execution time because those parameters also represent the effect of the number of LSQ entries and the latency of the following blocks, respectively. However, determining the exact contribution of the

number of LSQ entries and the latency of the following blocks would necessitate increasing the number of parameters by two and quadrupling the total simulation time required for this study.

The second result is the branch predictor, the number of Integer ALUs, and the L2 Cache latency have very different effects on each benchmark. For instance, the branch predictor has a large effect in three benchmarks, 124.m88ksim (28.89%), 132.ijpeg (19.64%), and 177.mesa (37.36%). On the other hand, the branch predictor has virtually no effect for seven benchmarks, 099.go (0.53%), 126.gcc (0.27%), 134.perl-Primes (0.56%), 179.art (0.02%), 181.mcf (0.26%), 188.ammp (0.02%), and 255.vortex (0.43%). However, it is very important to notice that the *average* effect of these parameters is very low (branch predictor: 1.037%, number of Integer ALUs: 0.002%, and L2 Cache latency 1.140%). For these three parameters, the actual value may have a very sizeable effect on the execution time or, more likely, it may have almost no effect. However, without performing this type of analysis, it is virtually impossible to predict the actual effect a parameter of this type will have on a specific benchmark.

The relatively weak impact of the branch predictor could be partially explained by the branch misprediction penalty. Since the SimpleScalar architecture is only a five-stage pipeline, we set the branch misprediction penalty to 3 cycles for all simulations, which is a reasonable value for this pipeline. While it is difficult to project the effect of a higher branch misprediction penalty, it is likely that the effect will not be large enough to significantly change these results, especially in the light of the fact that the SPEC benchmarks do not effectively stress the memory hierarchy. In other words, the branch predictor becomes less important when memory access accounts for a higher percentage of the execution time, which is not the case with the SPEC benchmarks. However, since this result is somewhat counterintuitive, we will study the effect of the branch predictor with a larger misprediction penalty as part of our future work.

A third result is that some parameters, the L1 D-Cache block size, the L2 Cache size, and the L2 Cache associativity, have a uniformly weak impact for all benchmarks. This small impact is partially due to the choice of input set. Since all the benchmarks either used the test, train, or a reduced input set, and since these input sets are known to have a smaller memory footprint than the reference input set, the effect of these three parameters is somewhat muted. However, using

the reference input sets would have increased the total simulation time of this study to an unmanageable level. Nevertheless, it is probably safe to say that a larger memory footprint will increase the effect of these parameters at the expense of other parameters not associated with the memory hierarchy. In spite of this qualification, this result seems to advocate, even for the SPEC benchmarks, the design approach of using a large, highly associative L1 D-Cache and a moderately sized L2 Cache with a low associativity.

Finally, the last result from Figure 1 is that only a few parameters are chiefly responsible for most of the variation in the execution time. Furthermore, with the exception of the 129.compress and 134.perl-Primes, at least 60.06% of the variation in the execution time is due to single parameters. (The effect of single parameters on these benchmarks is small since they have relatively low instruction counts.) In other words, the variation in the execution time is not primarily due to the interaction between parameters, but rather to each parameter individually. For example, the top five single parameters account for 42.18% – 97.74% of the total variation in the execution time while all ten parameters account for 48.63% – 97.75%. These ranges show that only a few parameters account for a very significant fraction of the total variability in the overall execution time.

5.2 The Effect of Parameter Interactions

Table 7 shows the interactions between parameters that had the most effect in each benchmark. An interaction between two or more parameters is the impact that those parameters have when they are both "on." The first column lists the interaction while the second column lists the benchmarks for which that interaction had the greatest effect. The third column shows the range of percentages for that interaction.

Not surprisingly, the dominant interaction is due to the two parameters that have the most effect: the number of ROB entries and the L1 D-Cache associativity. This is also the interaction that has the most impact, on average (9.60%). Furthermore, in all but one benchmark (177.mesa), at least one parameter in the interaction is a dominant parameter, e.g. the number of ROB entries, the L1 D-cache size or associativity, and the memory latency. (For 177.mesa, the

most dominant interaction has relatively little effect, only 5.98%.) Therefore, since the dominant interaction has a sizeable effect (with the exceptions of 124.m88ksim, 183.equake, and 188.ammp), one must be even more careful when choosing the values for the four dominant parameters because the influence of a dominant parameter also inflates the effect of the interaction of that parameter and another.

Interaction	Benchmark	Range (%)
# of ROB Entries & L1 D-Cache Assoc.	099.go, 126.gcc, 129.compress, 134.perl-Jumble, 164.gzip, 175.vpr-Route, 179.art, 181.mcf, 183.equake, 188.ammp, 197.parser, 255.vortex	1.53 - 11.8
# Int ALUs & Memory Latency	124.m88ksim	2.47
L1 D-Cache Size & L2 Cache Latency	130.li, 132.ijpeg, 175.vpr-Place, 300.twolf	5.45 - 15.35
L1 D-Cache Block Size & L1 D-Cache Assoc.	134.perl-Primes	7.92
L2 Cache Size & L2 Latency	177.mesa	5.98

Table 7: The Primary Interactions and Their Effect on the Total Variation in the Execution Time for Each Benchmark

6 Future Work

The first few items that we are focusing on for future work are the effect of the issue width, the instruction latencies and degree of pipelining within a functional unit, and the effect of the branch misprediction penalty. When the issue width is increased and the other processor parameters are scaled accordingly, do the same results hold? The same question holds for the instruction latencies and the degree of pipelining within a functional unit. For the branch misprediction penalty, we are interested in checking whether or not our assertion that the results will not change dramatically when the branch misprediction penalty is increased is correct.

We are also currently working on a more detailed study of the memory hierarchy since the results in this paper indicated that a few memory hierarchy parameters (along with the number of ROB entries) were generally responsible for most of the variation in the execution time.

Finally, while we think that our results are general enough to be applied to any architecture, we are interested in confirming that claim.

7 Conclusion and Recommendations

Computer architects rely heavily on simulators when trying to design a new processor architecture or when evaluating the performance of new compiler-based and microarchitectural mechanisms. However, since the simulation results can change significantly based on the values of the processor parameters that are used (i.e. independent of what feature is actually under test), it is extremely important to choose reasonable parameter values. However, it is unknown how much of an impact each processor parameter (or interaction between two or more parameters) actually has.

To focus on this problem and to take the first step in developing a methodology to determine the best parameter values, we determined the effect of what we judged to be the ten most important processor parameters. Specifically, we examined the effect of the branch predictor; the number of Integer ALUs; the number of reorder buffer (ROB) entries; the L1 D-Cache size, block size, and associativity; the L2 Cache size, associativity, and latency; and the memory latency. To determine the actual impact of each parameter or combination of parameters, we used the statistical analysis of variance technique.

Our results showed that the number of ROB entries, the L1 D-Cache size and associativity, and the memory latency accounted for 75.71% of the total variation in the execution time (cycles). In other words, these parameters were responsible for 75.71% of the change in the execution time. Surprisingly, the branch predictor accounted for only 1.04% of the variation in the execution time. However, this low percentage may be an artifact of the relatively low – but architecture-accurate – branch misprediction penalty of three cycles. Our results also show that the combination of the number of ROB entries and the L1 D-Cache associativity accounts for 9.60% of the variation in the execution time while all other combinations of parameters have very little impact.

Based on these results, we make two recommendations. First of all, we recommend that extreme care be exercised when choosing the values for these four dominant parameters. Not only must the parameter values be reasonable (i.e. in the ballpark for that specific issue width), they must also be appropriate for the architecture and be feasible to implement. For instance, one must choose a ROB size that is appropriate for the architecture's pipeline and that can be accessed in a reasonable number of cycles.

In addition to considering whether or not the parameter values accurately reflect what is appropriate for that architecture and what can actually be implemented, one must also consider the effect of the input set that is used for the benchmark when choosing the parameter values. For instance, our results showed that the parameters from the memory hierarchy accounted for 48.14% of the variation observed in the execution time. Therefore, if an input set is used that has a memory footprint that is smaller than the reference input set's memory footprint, the memory hierarchy parameters will have *less* impact than if the reference input set was used. As a result, incorrect conclusions will be drawn about the performance potential of that particular processor or microarchitectural enhancement.

Secondly, we also recommend iteratively testing the performance of the simulated processor after selecting the parameter values to ensure that the parameters that were chosen will reasonably demonstrate the performance potential of the feature under test. Without this step, one could carefully choose slightly "wrong" values for some of the key parameters, which could seriously overestimate or underestimate the performance of that enhancement and lead to incorrect conclusions. And ultimately, this defeats the purpose of performing those simulations.

References

[Bannon97] P. Bannon and Y. Saito; "The Alpha 21164PC Microprocessor"; International Computer Conference, 1997

[Black98] B. Black and J. Shen; "Calibration of Microprocessor Performance Models"; IEEE Computer, Vol. 31, No. 5, May 1998; Pages 59-65

[Burger97] D. Burger and T. Austin; "The SimpleScalar Tool Set, Version 2.0"; University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, 1997

[Christie96] D. Christie; "Developing the AMD-K5 Architecture"; IEEE Micro, Vol. 16, No. 2, March-April 1996; Pages 16-26

[Desikan01] R. Desikan, D. Burger, and S. Keckler; "Measuring Experimental Error in Microprocessor Simulation"; International Symposium on Computer Architecture, 2001

[Edmondson95] J. Edmondson, P. Rubinfeld, and R. Preston; "Superscalar Instruction Execution in the 21164 Alpha Microprocessor"; IEEE Micro, Vol. 15, No. 2, March-April 1995; Pages 33-43

[Gibson00] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich; "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop"; International Conference on Architectural Support for Programming Languages and Operating Systems, 2000

[Horel99] T. Horel and G. Lauterbach; "UltraSPARC-III: Designing Third-Generation 64-Bit Performance"; IEEE Micro, Vol. 19, No. 3, May-June 1999; Pages 73-85

[Kessler98] R. Kessler, E. McLellan, and D. Webb; "The Alpha 21264 Microprocessor Architecture"; International Conference on Computer Design, 1998

[Kessler99] R. Kessler; "The Alpha 21264 Microprocessor"; IEEE Micro, Vol. 19, No. 2, March-April 1999; Pages 24-36

[KleinOsowski00] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja; "Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research"; Workload Characterization of Emerging Computer Applications; Pages 83-100

[Kumar97] A. Kumar; "The HP PA-8000 RISC CPU"; IEEE Micro, Vol. 17, No. 2, March-April 1997; Pages 27-32

[Leiholz97] D. Leiholz and R. Razdan; "The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor"; International Computer Conference, 1997

[Lilja00] D. Lilja; “Measuring Computer Performance”; Cambridge University Press, 2000

[Matson98] M. Matson, D. Bailey, S. Bell, L. Biro, S. Butler, J. Clouser, J. Farrell, M. Gowan, D. Priore, and K. Wilcox; "Circuit Implementation of a 600 MHz Superscalar RISC Microprocessor"; International Conference on Computer Design, 1998

[Normoyle98] K. Normoyle, M. Csoppenszky, A. Tzeng, T. Johnson, C. Furman, and J. Mostoufi; "UltraSPARC-IIi: Expanding the Boundaries of a System on a Chip"; IEEE Micro, Vol. 18, No. 2, March-April 1998; Pages 14-24

[Papworth96] D. Papworth; "Tuning the Pentium Pro Microarchitecture"; IEEE Micro, Vol. 16, No. 2, March-April 1996; Pages 8-15

[Reilly99] M. Reilly; "Designing an Alpha Microprocessor"; IEEE Computer, Vol. 32, No. 7, July 1999; Pages 27-34

[Silc99] J. Silc, B. Robic, and T. Ungerer; “Processor Architecture : From Dataflow to Superscalar and Beyond”; Springer-Verlag, 1999

[Sima97] D. Sima, T. Fountain, and P. Kacsuk; “Advanced Computer Architectures, A Design Space Approach”; Addison Wesley Longman, 1997

[Sodani97] A. Sodani and G. Sohi; "Dynamic Instruction Reuse"; International Symposium on Computer Architecture, 1997

[Tremblay96] M. Tremblay and J.M. O'Connor; "UltraSparc I: A Four-Issue Processor Supporting Multimedia"; IEEE Micro, Vol. 16, No. 2, March-April 1996; Pages 42-50

[Yeager96] K. Yeager; "The MIPS R10000 Superscalar Microprocessor"; IEEE Micro, Vol. 16, No. 2, March-April 1996; Pages 28-40