A Statistically Rigorous Approach for Improving Simulation Methodology

Joshua J. Yi[1], David J. Lilja[1], and Douglas M. Hawkins[2]
[1]Department of Electrical and Computer Engineering
[1]Minnesota Supercomputing Institute
[2]School of Statistics
University of Minnesota – Twin Cities
{jjyi,lilja}@ece.umn.edu
doug@stat.umn.edu

Abstract

Due to cost, time, and flexibility constraints, simulators are often used to explore the design space when developing a new processor architecture, as well as when evaluating the performance of new compiler-based and microarchitectural performance-enhancement mechanisms. However, despite this continued dependence on simulators, statistically rigorous simulation methodologies are not typically used for computer architecture research. Without a formal methodology, however, it is possible to underemphasize the effect of parameter interactions when choosing processor parameter values for simulations or when performing sensitivity analyses. A formal methodology can provide a sound basis for drawing conclusions gathered from simulation results by adding statistical rigor and, consequently, can increase confidence in the simulation results. This paper demonstrates the application of a rigorous statistical technique to the setup and analysis phases of the simulation process. Specifically, we apply a Plackett and Burman design to: 1) identify key processor parameters, 2) classify benchmarks based on how they affect the processor, and 3) analyze the effect of processor performance enhancements. Our technique expands on previous work by applying a statistical method to improve the simulation methodology instead of applying a statistical model to estimate the performance of the processor.

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 quickly determine the expected performance improvement of a new compiler-based or microarchitectural mechanism.

Despite this dependence on simulators, computer architects often approach the simulation process in an ad-hoc manner. For example, when performing a sensitivity analysis, the architect will hold most of the processor parameters constant while varying the values of a select group. However, there are several questions that must be addressed regarding the simulation setup. For instance, which parameters should be held constant and which should be varied? Do any of the constant parameters interact with the variable ones? What is the magnitude of the effects of those interactions? How do the magnitudes of the interactions compare with the magnitudes of the effects of the individual parameters? What values should be used for the constant parameters? Should they be set to very large values or to middle-of-the-range values? How much effect, if any, do the specific values chosen for the constant parameters have on the results? Can they significantly alter the apparent effect of the variable parameters? What range of values should be used to test the effects of the variable parameters?

Since it is impossible to separate out the effect of the interactions and constant parameters after performing the simulations when varying only a single parameter at a time, the architect must answer these questions before starting the simulations. Due to the sheer computational cost, however, it is virtually impossible to simulate the effect of all parameters and their interactions simultaneously. This type of situation illustrates the need for a statistically-based methodology that addresses these types of questions.

While using such a methodology may require the overhead of some additional simulations, it has the following advantages as compared to the current ad-hoc method:

1) It **decreases the number of errors** that are present in the simulation process and helps the computer architect **detect errors more quickly**. Errors include, but are not limited to, simulator modeling errors, user implementation errors, and simulation setup errors [Black98, Cain02, Desikan01, Gibson00, Glamm00].
2) It **gives more insight into what is occurring** inside the processor or the actual effect that a compiler-based or microarchitectural enhancement has on the processor.
3) It **gives objective confidence to** the results and **provides statistical support** regarding the observed behavior.

While the first and third advantages are self-explanatory, it is not obvious from the second advantage how a statistically rigorous simulation methodology can help improve the quality of the analysis. Since simulators are complex, it is very difficult to fully understand the effect that a design change or an enhancement may have on the processor. As a result, in most cases, architects resort to using high-level metrics, such as speedup, to understand the "big-picture" effects. However, analyzing a processor from a statistical point-of-view can help the architect quantify the effects that all components have on the performance and on other important design metrics, such as the power consumption, and so on.

Therefore, as a first step in developing a formal methodology, this paper makes specific suggestions on how to improve the simulation setup process and the analysis of results. The suggestions include methods for identifying the key processor parameters, an approach for classifying benchmarks based on how they affect the processor, and analyzing the effects of processor enhancements.

The contributions of this paper are as follows:

1) This paper demonstrates the need for methodological improvement in computer architecture research and the efficacy of a particular statistical method to accomplish that.
2) This paper makes specific recommendations on how to improve the simulation methodology. In particular, the recommendations include how to: A. choose the processor parameter values, B. classify benchmarks, and C. analyze the effect that an enhancement has on the processor. Collectively, these recommendations can improve simulation methodology, decrease the total number of simulations, quickly determine the processor's performance bottlenecks (i.e. key processor parameters), and provide analytical insights into the impact of processor enhancements.
3) This paper, by way of illustrating the second contribution, determines the most important machine parameters in the commonly used SimpleScalar superscalar simulator [Burger97].

The remainder of this paper is organized as follows: Section 2 describes the statistical method that is used in this paper, the Plackett and Burman (PB) design. Sections 3 and 4 describe the experimental setup and the results, respectively, while Section 5 discusses some related work. Section 6 then concludes.

2 Fractional Multifactorial Analysis Using a Plackett and Burman Design

2.1 Computational Cost vs. Level of Detail for Applicable Statistical Methods

The value of a parameter that remains constant throughout a group of simulation experiments can significantly affect the results of a sensitivity analysis by interacting with one of the variable parameters. Varying just one parameter at a time, while leaving others constant, is called a 'one-at-a-time' experimental design. These designs are bad and should be avoided because they are vulnerable to masking important effects that exist due to interactions between the parameters. Furthermore, a constant parameter can be set to such an extreme value that it dominates the results and overshadows the effect of the parameters under test. For instance, setting the size of a buffer too small can cause this unit to become the performance bottleneck, thereby masking the actual performance effect provided by some new architectural enhancement.

To avoid these problems, the user should vary all parameters simultaneously. However, the problem with this solution is that it is extremely computationally expensive. For example, if the user were trying to measure the effect of 40 parameters, each of which can assume only two values, simulating all the possible combinations would require 2^{40}, or more than 1 *trillion*, simulations. Since this number of simulations is prohibitively high, simulating all possible combinations of parameter values is not a feasible solution. For N parameters, the first technique of fixing all parameters save one requires N+1 simulations. On the other hand, simulating all possible combinations of N parameters that can assume b unique values requires b^N simulations. That latter case is referred to as a full multifactorial design or analysis. An example of this is the analysis of variance (ANOVA) technique [Lilja00].

'Saturated designs' are recipes that vary all N parameters simultaneously over a total of N+1 simulations. They provide the logically minimal number of simulations required to estimate the effect of each of the N parameters. The Plackett-Burman (PB) design [Plackett46] is a well-established approach of this type. An improvement on the basic PB design is the 'foldover' PB design [Montgomery91]. This requires 2(N+1) runs. With this experimental design, the user can determine the effect of all of the main parameters and selected interactions. PB designs exist only in sizes that are multiples of 4. Thus a foldover PB design requires 2X simulations, where X is the next multiple of four that is greater than N. Table 1 summarizes the key aspects of each of the previously mentioned three methods.

As is summarized in Table 1, the one-at-a-time single parameter design, as exemplified by a set of single parameter sensitivity analyses, requires half as many simulations as the fractional multifactorial design. However, the level of detail that is obtained from these simulations is much lower than the other experimental designs. It provides information on each parameter at only a single level of all other parameters (rather than the averaging obtained with a simultaneous design) and is badly affected by interactions, which it has no power to detect. Furthermore, since each parameter's effect is estimated by the difference between a single simulation and the base case, the information it gives is far less precise than that given by a multifactorial approach in which each parameter is at the high level in half the simulations and at the low level in the other half.

Table 1: Key Aspects of Three Simulation Designs, N = Number of Parameters, Each of Which can Assume Two Values

Design	Example	Simulations	Level of Detail
One Parameter at-a-time	Simple Sensitivity Analysis	N+1	Single Parameter
Fractional	Plackett and Burman	~2N	All Parameters, Selected Interactions
Full Multifactorial	ANOVA	2^N	All Parameters, All Interactions

On the other hand, the user is able to glean the most information from a full multifactorial design. The effect of all parameters and all interactions can be explicitly quantified using such a design. However, the computational cost is enormous unless N is tiny. As a result, while this type of design could be extremely useful for a small number of parameters, with a large number of parameters, the computational cost is prohibitively expensive.

2.2 The Plackett and Burman Fractional Experimental Design

From a cost versus level of detail obtained point-of-view, using a fractional multifactorial design, such as a Plackett and Burman design, is by far the best choice. This type of design only requires approximately 2N simulations, but it quantifies the effects of all of the parameters and specific interactions selected by the experimenter.

The downside of the PB design, however, is that it cannot quantify the effects of all the interactions. Therefore, the possibility exists for unknown, but significant, interactions to alter the apparent effect of any of the parameters. Fortunately for computer architects, this situation probably does not occur for processor parameters, as was shown for a set of benchmarks from the SPEC 2000 benchmark suite [Yi02-2]. When there was a significant interaction, it was the result of two significant individual parameters, although, in contrast to the main effects, the effect of the interactions was relatively small. As a result, using a fractional multifactorial design to analyze the effects of processor parameters does not affect or compromise the results. For even more precision, the fractional multifactorial design can be used to determine the most significant parameters and then a full multifactorial design can be used to determine the effects of those most significant parameters and their interactions.

The PB design determines the effect of each parameter by varying several parameters in each configuration and then comparing the results of all configurations with respect to a base case. The PB design matrix gives the parameters' configuration for each test case. For most values of X, a PB design is fairly simple to construct. For these values of X, the first row of the design matrix, a series of plus and minus ones, is given in [Plackett46]. The design matrix is X rows by X – 1 columns (recall that X is the next multiple of 4 larger than the N, the number of parameters being varied). The next X – 2 rows are formed by performing a circular right shift on the preceding row. The last line of the design matrix, line X, is a row of minus ones. Table 2 illustrates the construction of the PB design matrix for X=8, a design appropriate for investigating 7 (or fewer) parameters

Table 2: Plackett and Burman Design Matrix for X = 8 (Up to 7 Parameters)

+1	+1	+1	-1	+1	-1	-1
-1	+1	+1	+1	-1	+1	-1
-1	-1	+1	+1	+1	-1	+1
+1	-1	-1	+1	+1	+1	-1
-1	+1	-1	-1	+1	+1	+1
+1	-1	+1	-1	-1	+1	+1
+1	+1	-1	+1	-1	-1	+1
-1	-1	-1	-1	-1	-1	-1

The rows of the design matrix correspond to different simulation configurations. The columns of the design matrix correspond to the values that each of the parameters will have. In the event that there are more columns than parameters (i.e. N < X – 1), then the additional columns are simply "dummy parameters" and consequently have no effect on the simulation results. A "+1" in the table corresponds the "high" value for a parameter. The high value represents a value that is higher than the range of normal values. Conversely, a "-1" corresponds to the "low" value for a parameter, a value that is lower than the range of normal values. It is important to note that the high and low values are not restricted to only numerical values. In the case of branch prediction, for instance, the high value could correspond to perfect branch prediction while the low value could correspond to some other specific branch predictor. In the case of compiler optimizations, the high value could correspond to loop unrolling, for instance, while the low value would correspond to no loop unrolling.

It is extremely important to note that choosing high and low values that represent too large a range for numerically-valued parameters can significantly affect the results by inflating the effect of that parameter. Furthermore, the opposite situation of too small a range has the opposite effect. Therefore, it is necessary to exercise some caution when choosing each value. Ideally, the high and low values for each parameter should be just outside of the "normal" range of values. If not, then the effect of that parameter could be over or under-inflated by an amount proportional to the difference between the "ideal" and actual values for that parameter.

The most basic PB design requires only X runs. However, with this basic design, the user is unable to determine the effects of any interactions. Therefore to protect the results from the effects of some of the most important interactions, the user can use the *foldover* concept. Using foldover, X additional rows are added to the matrix. The signs in each entry in the additional rows are the opposite of the corresponding entries in the original matrix. Table 3 shows the resulting design matrix after using foldover for the design shown in Table 2. Note that the original design matrix is shaded with gray in Table 3.

Table 3: Plackett and Burman Design Matrix for X=8 with Foldover

+1	+1	+1	-1	+1	-1	-1
-1	+1	+1	+1	-1	+1	-1
-1	-1	+1	+1	+1	-1	+1
+1	-1	-1	+1	+1	+1	-1
-1	+1	-1	-1	+1	+1	+1
+1	-1	+1	-1	-1	+1	+1
+1	+1	-1	+1	-1	-1	+1
-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	+1	-1	+1	+1
+1	-1	-1	-1	+1	-1	+1
+1	+1	-1	-1	-1	+1	-1
-1	+1	+1	-1	-1	-1	+1
+1	-1	+1	+1	-1	-1	-1
-1	+1	-1	+1	+1	-1	-1
-1	-1	+1	-1	+1	+1	-1
+1	+1	+1	+1	+1	+1	+1

After generating the configuration files and performing the simulations, the effect of each parameter is computed by multiplying the result for each configuration by the value of the entry for that configuration and parameter combination. Then the results of all those multiplications are summed together to determine the overall effect for that parameter. Table 4 illustrates the mechanics of this process using the design matrix shown in Table 2 for parameters A – G.

Table 4: Example Analysis Using a Plackett and Burman Design Without Foldover for X=8

	A	B	C	D	E	F	G	Result
	+1	+1	+1	-1	+1	-1	-1	1
	-1	+1	+1	+1	-1	+1	-1	9
	-1	-1	+1	+1	+1	-1	+1	74
	+1	-1	-1	+1	+1	+1	-1	28
	-1	+1	-1	-1	+1	+1	+1	3
	+1	-1	+1	-1	-1	+1	+1	6
	+1	+1	-1	+1	-1	-1	+1	112
	-1	-1	-1	-1	-1	-1	-1	84
Effect	-23	-67	-137	129	-105	-225	73	

The effect of parameter A in this table is computed as follows:

$$\text{Effect}_A = (1 * 1) + (-1 * 9) + (-1 * 74) + (1 * 28) + (-1 * 3) + (1 * 6) + (1 * 112) + (-1 * 84) = -23$$

These results show that the parameters with the most effect are F, C, and D, in order of their overall impact on performance. Only the magnitude of the effect is important; the sign of the effect is meaningless.

To summarize, this section compares three different statistical techniques in terms of the number of simulations versus level of detail obtained from these simulations as it applies to simulation-based computer architecture research. We find that a PB design finely balances the number of simulations required with the level of detail that can be obtained.

3 Simulator, Benchmarks, and Processor Parameter Values

In the remainder of the paper, we demonstrate how the Plackett and Burman experimental design approach can be used to appropriately select parameters for a series of simulation experiments, to classify and select benchmark programs for the simulations, and to provide insights into the performance impact of a specific microarchitectural enhancement. The base simulator was from the SimpleScalar tool suite [Burger97] and is an execution-driven simulator that models a state-of-the-art superscalar processor. The simulator was modified to include user configurable instruction latencies and throughputs.

The benchmarks that were used in this study, shown in Table 5, were selected from the SPEC 2000 benchmark suite. These benchmarks were chosen because they were the only ones that had MinneSPEC [KleinOsowski02] large reduced input sets available at the time. Since vpr uses two "sub-input" sets, Place and Route, the results for each are listed separately. All benchmarks were compiled at optimization level O3 using the SimpleScalar version of the gcc compiler and were run to completion.

Table 5: Selected Benchmarks from the SPEC 2000 Benchmark Suite Used in This Study

Benchmark	Type	Instructions Simulated (M)
gzip	Integer	1364.2
vpr-Place	Integer	1521.7
vpr-Route	Integer	881.1
gcc	Integer	4040.7
mesa	Floating-Point	1217.9
art	Floating-Point	2181.1
mcf	Integer	601.2
equake	Floating-Point	713.7
ammp	Floating-Point	1228.1
parser	Integer	2721.6
vortex	Integer	1050.2
bzip2	Integer	2467.7
twolf	Integer	764.6

As described in the Section 2, the choice of values for the processor parameters used in the simulations should be chosen to be values that are slightly too low and too high to allow the PB experimental design to work most efficiently. As a result, the final values that we chose for each parameter are not values that would be actually present in commercial processors nor are they supposed to represent a potential value. Rather, the values were ***deliberately chosen to be values that were slightly higher and lower than the range of "reasonable" values***. Choosing values in this way allows the PB design to more accurately determine the effect of each parameter on the processor's performance.

We based our range of "reasonable" values on the parameter values found in several commercial processors. Our list of commercial processors included the Alpha 21164 [Bannon97, Edmondson95] and 21264 [Kessler98, Kessler99, Leiholz97, Matson98]; the UltraSparc I [Tremblay96], II [Normoyle98], and III [Horel99]; HP PA-8000 [Kumar97]; the PowerPC 604 [Song94]; and the MIPS R10000 [Yeager96]. To fill in the gaps left by the aforementioned papers, [Silc99, Sima97] and several web searches were also used as references.

Based on the range of reasonable values, we chose a low and high value for each parameter. Tables 6, 7, and 8 show the final values for each of the relevant parameters for the processor core, the functional units, and the memory hierarchy, respectively.

Table 6: Processor Core Parameters and Their Plackett and Burman Values

Processor Core Parameter	Low/Off Value	High/On Value
Instruction Fetch Queue (IFQ) Entries	4	32
Branch Predictor	2-Level	Perfect
Branch Predictor Misprediction Penalty	10 Cycles	2 Cycles
Return Address Stack (RAS) Entries	4	64
Branch Target Buffer (BTB) Entries	16	512
Branch Target Buffer (BTB) Associativity	2-Way	Fully-Associative
Speculative Branch Update	In Commit	In Decode
Decode, Issue, and Commit Width	4-Way	
Reorder Buffer (ROB) Entries	8	64
Load-Store Queue (LSQ) Entries	0.25 * ROB	1.0 * ROB
Memory Ports	1	4

Table 7: Functional Units Parameters and Their Plackett and Burman Values

Functional Unit Parameter	Low/Off Value	High/On Value
Integer ALUs	1	4
Integer ALU Latencies	2 Cycles	1 Cycle
Integer ALU Throughputs	1	
Floating-Point ALUs	1	4
Floating-Point ALU Latencies	5 Cycles	1 Cycle
Floating-Point ALU Throughputs	1	
Integer Mult/Div Units	1	4
Integer Multiply Latency	15 Cycles	2 Cycles
Integer Divide Latency	80 Cycles	10 Cycles
Integer Multiply Throughput	1	
Integer Divide Throughput	Equal to the Integer Divide Latency	
Floating-Point Mult/Div Units	1	4
Floating-Point Multiply Latency	5 Cycles	2 Cycles
Floating-Point Divide Latency	35 Cycles	10 Cycles
Floating-Point Square Root Latency	35 Cycles	15 Cycles
Floating-Point Multiply Throughput	Equal to the Floating-Point Multiply Latency	
Floating-Point Divide Throughput	Equal to the Floating-Point Divide Latency	
Floating-Point Square Root Throughput	Equal to the Floating-Point Square Root Latency	

Several parameters across all three tables are shaded in gray. For these parameters, the low and high values cannot be chosen completely independently of the other parameters due to the mechanics of a PB design. The problem occurs when one of the shaded parameters is set to its high or low value and the parameter it is related to is set to the opposite value. In those configurations, the combination of values for those parameters leads to a situation that either does not make sense or would not actually occur in a real processor. For example, if the number of LSQ entries were chosen independently of the number of ROB entries, some of the configurations would have an 8-entry reorder buffer and a 64-entry LSQ. Since the total number of in-flight instructions cannot exceed the number of reorder buffer entries, the maximum number of filled LSQ entries could never exceed 8. Therefore, to avoid the above and other similar situations, the specific values used in the simulations for all gray-shaded parameters are based on their related parameter.

All parameter values were based on a 4-way issue processor. While the issue width is a very important parameter, we fixed the issue width at 4 for several reasons. The first reason is the same as the reason given above for the parameters shaded in gray. If the issue width were set to its low value while the number of functional units were set to their high values, then some of the functional units would never be used since simulator allows only 4 new instructions to start executing per cycle. Second, several 4-way issue commercial processors exist and these processors are fairly well documented. Therefore, to obtain a good range of values for each parameter, we chose the issue width to reflect the issue width of the processors with good documentation. However, fixing the issue width to a constant value does not affect the conclusions drawn from these simulations in any way. It only removes the issue width as one of variable parameters.

Table 8: Memory Hierarchy Parameters and Their Plackett and Burman Values

Memory Hierarchy Parameter	Low/Off Value	High/Off Value
L1 I-Cache Size	4 KB	128 KB
L1 I-Cache Associativity	1-Way	8-Way
L1 I-Cache Block Size	16 Bytes	64 Bytes
L1 I-Cache Replacement Policy	Least Recently Used (LRU)	
L1 I-Cache Latency	4 Cycles	1 Cycle
L1 D-Cache Size	4 KB	128 KB
L1 D-Cache Associativity	1-Way	8-Way
L1 D-Cache Block Size	16 Bytes	64 Bytes
L1 D-Cache Replacement Policy	Least Recently Used (LRU)	
L1 D-Cache Latency	4 Cycles	1 Cycle
L2 Cache Size	256 KB	8192 KB
L2 Cache Associativity	1-Way	8-Way
L2 Cache Block Size	64 Bytes	256 Bytes
L2 Cache Replacement Policy	Least Recently Used (LRU)	
L2 Cache Latency	20 Cycles	5 Cycles
Memory Latency, First Block	200 Cycles	50 Cycles
Memory Latency, Following Blocks	0.02 * Memory Latency, First Block	
Memory Bandwidth	4 Bytes	32 Bytes
I-TLB Size	32 Entries	256 Entries
I-TLB Page Size	4 KB	4096 KB
I-TLB Associativity	2-Way	Fully-Associative
I-TLB Latency	80 Cycles	30 Cycles
D-TLB Size	32 Entries	256 Entries
D-TLB Page Size	Same as I-TLB Page Size	
D-TLB Associativity	2-Way	Fully-Associative
D-TLB Latency	Same as I-TLB Latency	

Finally, it is important to mention that this particular simulator was used instead of the validated Alpha 21264 simulator [Desikan01] for three reasons. The first reason is that this is a methodology study and not an architecture or performance-only study. Consequently, since the simulation results serve only to illustrate certain key points, the choice of a specific simulator does not affect the point that is being made. The second reason is that the Alpha simulator contains many parameters that are specific to the Alpha architecture while the basic SimpleScalar simulator models a generic superscalar processor. Therefore, to avoid the risk of producing results that are particular to the Alpha 21264 processor, we decided to use the generic processor. The third reason is that the SimpleScalar simulator is itself a widely used simulator. Therefore, using this simulator has the extra benefit of producing results that are beneficial to the SimpleScalar community.

4 Plackett and Burman Design Results for the Simulation Setup and Analysis

The three most basic stages of the simulation process in computer architecture research are simulation setup, the simulation itself, and data analysis. Note that the first stage occurs after determining the initial set of testcases to examine and after modifying the simulator and/or compiler. Therefore, in the first stage, the user needs to determine the values of the different user-configurable processor parameters and to select the benchmarks that will be simulated. In the third stage, the user analyzes the results that were gathered during the simulation stage. Then, depending on the results, the user may wish to repeat the process with variations to the specific mechanism or enhancement being tested.

The section focuses on improving the methodology in the first and third stages of the simulation process. To improve the methodology of the first stage, we describe a statistically rigorous method of choosing the values of the processor parameters. In addition, to improve the benchmark selection process, we describe a method of classifying benchmarks based on grouping those benchmarks that have similar effects on the processor. To improve the methodology of the third stage, we describe a statistically rigorous method of analyzing the effect that an enhancement has on the base processor. For each method, we briefly describe the problem or pitfalls that could result if that particular method were not employed. Furthermore, to show the effectiveness, usefulness, and mechanics of each method, a short example is given. It is important to note that each example contains general results that can be considered a contribution to the art.

4.1 Pre-Simulation Methodology: Processor Parameter Selection

Improperly choosing a single parameter can significantly affect the simulated speedup of a processor enhancement that is being tested. For instance, simply increasing the reorder buffer size can change the speedup of a value reuse mechanism from approximately 20% to approximately 30% [Yi02-2]. As a result, a few poorly chosen parameters can severely overestimate or underestimate the potential speedup of the enhancement. However, choosing a "good" set of parameters is extremely difficult since many of the important parameters may interact, thus compounding the error associated with choosing only a single poor value for one of the parameters. Determining which parameters interact requires performing a sensitivity analysis on all the parameters simultaneously or choosing a select few parameters for detailed study. The problem with the former approach is that trying to simulate all possible combinations is an extremely difficult problem. The problem with the latter approach is that in studying only a few parameters, other parameters have to be set to constants. However, if one of those constant parameters is significant and interacts heavily with some of the free parameters, then the results of the sensitivity analysis will be distorted. Fortunately, this problem can be solved by using a PB design to identify the significant parameters.

Table 9: Plackett and Burman Design Results for All Processor Parameters; Ranked by Significance and Sorted by the Sum of Ranks

Parameter	gzip	vpr-Place	vpr-Route	gcc	mesa	art	mcf	equake	ammp	parser	Vortex	bzip2	twolf	Sum
Reorder Buffer Entries	1	4	1	4	3	2	2	3	6	1	4	1	4	36
L2 Cache Latency	4	2	4	2	2	4	4	2	13	3	2	8	2	52
BPred Type	2	5	3	5	5	27	11	6	4	4	16	7	5	100
Int ALUs	3	7	5	8	4	29	8	9	19	6	9	2	9	118
L1 D-Cache Latency	7	6	7	7	12	8	14	5	40	7	5	6	6	130
L1 I-Cache Size	6	1	12	1	1	12	37	1	36	8	1	16	1	133
L2 Cache Size	9	35	2	6	21	1	1	7	2	2	6	3	43	138
L1 I-Cache Block Size	16	3	20	3	16	10	32	4	10	11	3	22	3	153
Memory Latency First	36	25	6	9	23	3	3	8	1	5	8	5	28	160
LSQ Entries	12	14	9	10	13	39	10	10	17	9	7	4	10	164
Speculative Branch Update	8	17	23	28	7	16	39	12	8	20	22	20	17	237
D-TLB Size	20	28	11	23	29	13	12	11	25	14	25	11	24	246
L1 D-Cache Size	18	8	10	12	39	18	9	36	32	21	12	31	7	253
L1 I-Cache Associativity	5	40	15	29	8	34	23	28	16	17	15	9	21	260
FP Multiply Latency	31	12	22	11	19	24	15	23	24	29	14	23	19	266
Memory Bandwidth	37	36	13	14	43	6	6	29	3	12	19	12	38	268
Int ALU Latencies	15	15	18	13	41	22	33	14	30	16	41	10	16	284
BTB Entries	10	24	19	20	9	42	31	20	22	19	20	17	34	287
L1 D-Cache Block Size	17	29	34	22	15	9	24	19	28	13	32	28	26	296
Int Divide Latency	29	10	26	16	24	32	41	32	20	10	10	43	8	301
Int Mult/Div	14	20	29	31	10	23	27	24	33	36	18	26	15	306
L2 Cache Associativity	23	19	14	19	32	28	5	39	37	18	42	21	12	309
I-TLB Latency	33	18	24	18	37	30	30	16	21	32	11	29	18	317
Instruction Fetch Queue Entries	43	13	27	30	26	20	18	37	9	25	23	34	14	319
BPred Misprediction Penalty	11	23	42	21	6	43	20	34	11	22	39	37	23	332
FP ALUs	34	11	31	15	34	17	40	22	26	37	13	42	13	335
FP Divide Latency	22	9	35	17	30	21	38	15	43	38	17	39	11	335
I-TLB Page Size	42	39	8	37	36	40	7	17	12	26	28	14	39	345
L1 D-Cache Associativity	13	38	17	34	18	41	34	33	14	15	35	15	42	349
I-TLB Associativity	24	27	37	25	17	31	42	13	29	30	21	33	22	351
L2 Cache Block Size	25	43	16	38	31	7	35	27	7	35	38	13	40	355
BTB Associativity	21	21	36	32	11	33	17	31	34	43	27	35	25	366
D-TLB Associativity	40	32	25	26	22	35	26	26	18	33	26	30	35	374
FP ALU Latencies	32	16	38	41	38	11	22	30	23	27	30	40	29	377
Memory Ports	39	31	41	24	27	15	16	41	5	42	29	41	27	378
I-TLB Size	35	34	28	35	20	37	19	18	31	34	34	27	31	383
Dummy Factor #2	27	42	21	39	35	14	13	35	41	28	43	18	30	386
FP Mult/Div	41	22	43	40	40	19	28	38	27	31	31	19	20	399
Int Multiply Latency	30	41	39	36	14	26	29	21	15	41	37	32	41	402
FP Square Root Latency	38	30	40	33	33	5	25	42	42	24	24	38	37	411
L1 I-Cache Latency	26	26	32	42	28	38	21	40	38	40	36	25	33	425
Return Address Stack Entries	28	33	33	27	42	25	36	25	39	39	33	36	32	428
Dummy Factor #1	19	37	30	43	25	36	43	43	35	23	40	24	36	434

Table 9 shows the results of the PB design for the base superscalar processor with the parameter values shown in Tables 6-8. To generate the results in Table 9, an X = 44 foldover PB design was used. After simulating all 88 (2X) configurations, the PB design results were calculated. Then the parameters for each benchmark were assigned a rank based on the significance of the parameters (1 = most important). Then the ranks of each parameter were summed across all benchmarks and the resulting sums sorted in ascending order. Summing the ranks across benchmarks reveals the most significant parameters averaged across all of the benchmarks. The parameters with the lowest sums represent the parameters that have the most effect across all benchmarks.

Several key results can be drawn from this table. First, we see that only the first ten parameters are significant across all benchmarks. The conclusion can be drawn by examining the large difference between the sum of the ranks of the tenth parameter (LSQ size) and the sum of the ranks of the eleventh parameter (Speculative Branch Update). Furthermore, we see that, while the ranks of the top ten parameters for each benchmark are completely different, only two parameters (Reorder Buffer Entries and L2 Cache Latency) are significant across all benchmarks since those two parameters are almost always one of the most important parameters for every benchmark.

Second, the effect of each benchmark on the processor can be clearly seen. For instance, since the ranks for the I-Cache size, associativity, and block size are lower than or similar to the ranks for the D-Cache size, associativity, and block size for *mesa*, we conclude that *mesa* stresses the instruction cache much more than the data cache. Furthermore, the results show that *mesa's* performance is highly dependent on the branch predictor and its related parameters (misprediction penalty, BTB entries and associativity, and the speculative branch update) since those parameters have relatively low ranks. Since each benchmark has a unique "fingerprint" as to how it stresses the different parameters of the processor, a computer architect could use this information to classify benchmarks based on their effect on the processor, as we demonstrate in the next section.

Finally, several parameters have surprisingly high rankings in some benchmarks. For example, the FP square root latency in *art* has a rank of 5. Since *art* does not have a significant number of FP square root instructions, the rank of 5 does not appear to be consistent with the significance of that parameter. In this particular case, the parameter's rank only indicates that it is the fifth most important parameter. However, what the rank does not reveal is that this parameter is completely overshadowed when compared to any of the four most significant parameters (i.e. ranks 1-4). Therefore, this situation shows that the rank alone cannot be used to measure the significance of a parameter's impact on performance.

After determining the key parameters, the task of choosing the actual parameter values is greatly simplified since only the values for the key parameters need to be chosen with extreme caution; the others can be chosen with less caution. To actually choose the "appropriate" value for each of the key parameters, we recommend performing iterative sets of sensitivity analyses so that the exact interaction between key parameters can be accounted for when choosing the final parameter values. To summarize, we recommend the following steps when choosing processor parameter values:

1) Determine the critical processor parameters using a Plackett and Burman design.
   a) Choose low and high values for each of the parameters.
   b) Run and analyze the PB simulations to determine the key parameters.
2) Choose reasonable values for the non-critical parameters based on commercial processor values, or some other appropriate source.
3) Perform a sensitivity analysis over reasonable ranges for each critical parameter using the ANOVA technique.
4) Choose final values for the critical parameters based on the sensitivity analysis results.

4.2 Pre-Simulation Methodology: Benchmark Selection

Just as a poorly chosen set of processor parameters can drastically affect the performance results, a poorly chosen set of benchmarks from a benchmark suite may not accurately depict the true potential of the processor or the enhancement being tested. For instance, if the set of benchmarks was extremely memory-intensive, then an optimization to the memory hierarchy, such as data prefetching, will overestimate the performance of that optimization across all benchmarks. However, if in an effort to avoid the above situation the user decides to simulate all the benchmarks from the benchmark suite, then the user may be wasting time by simulating benchmarks that are too similar in their impact on the processor. In this case, the obvious drawback is that the user may be simulating "redundant" benchmarks at the expense of a more complete exploration of the design space. Therefore, for accuracy and efficiency reasons, it is important for the user to simulate a set of benchmarks that are distinct, but that are representative of the population of potential benchmarks.

However, determining which benchmarks in a benchmark suite are similar is a difficult task since benchmarks can be classified in many different ways (by application, by relative use of integer or floating-point operations, by processing time versus memory usage, etc.). Therefore, as an alternative classification that may be more relevant to computer architects, we propose that the benchmarks could be classified by their effect on the processor. Under this method of classification, if two benchmarks stress the same components of the processors to similar degrees, then the user could classify those two benchmarks as being similar.

To determine the similarity of benchmarks, we treated the rank of each parameter that we varied in this study as an element of a vector. Accordingly, each benchmark's vector represents the ranks of all parameters. To determine how similar the two benchmarks were, we simply computed the Euclidean distance between the two vectors as follows:

$$\text{Distance} = [(x_1-y_1)^2 + (x_2-y_2)^2 + \ldots + (x_{n-1}-y_{n-1})^2 + (x_n-y_n)^2]^{½}$$

In this computation, n is the number of parameters that can be varied and $X = [x_1, x_2, \ldots, x_{n-1}, x_n]$ and $Y = [y_1, y_2, \ldots, y_{n-1}, y_n]$ are the vectors that represent benchmark X and Y. Each element of X and Y corresponds to the rank of that parameter.

For example, the distance between *gzip* and *vpr-Place*, using the ranks from Table 9, is as follows:

$$\text{Distance} = [(6-1)^2 + (5-40)^2 + (16-3)^2 + \ldots + (40-32)^2 + (19-37)^2 + (27-42)^2]^{½} = [8058]^{½} = 89.8$$

Accordingly then, the distance between the two vectors is a measure of the similarity of the impact of the simulated parameters for the two benchmarks. Obviously, the smaller the distance between vectors, the greater the similarity between the two benchmarks, in terms of how those two benchmarks affect the processor. Benchmarks can be defined to be similar if their distance is below some user-defined threshold. Therefore, by comparing each benchmark against all the other benchmarks in a benchmark suite, the user can determine which benchmarks can be defined as similar.

To illustrate this process, Table 10 shows the result of comparing each benchmark against all the other benchmarks, including itself, using the ranks from Table 9.

Table 10: Distance Between Benchmark Vectors, Based on Parameter Ranks

	gzip	vpr Place	vpr Route	gcc	mesa	art	mcf	equake	ammp	parser	vortex	bzip2	twolf
gzip	0.0	89.8	81.1	81.9	**62.0**	113.5	109.6	79.5	111.7	73.6	92.0	78.1	85.5
vpr-Place	89.8	0.0	98.9	63.7	94.0	102.8	110.9	84.7	118.1	89.7	68.5	111.4	**35.2**
vpr-Route	81.1	98.9	0.0	71.7	98.5	100.4	75.5	73.3	91.7	**56.4**	79.2	**45.7**	96.6
gcc	81.9	63.7	71.7	0.0	90.9	92.6	94.5	63.6	98.5	65.0	**54.6**	88.8	67.3
mesa	**62.0**	94.0	98.5	90.9	0.0	120.9	109.9	81.8	100.2	88.9	87.8	94.1	91.7
art	113.5	102.8	100.4	92.6	120.9	0.0	98.6	96.3	105.2	94.4	92.7	102.5	105.2
mcf	109.6	110.9	75.5	94.5	109.9	98.6	0.0	104.9	94.8	87.6	101.3	80.0	111.1
equake	79.5	84.7	73.3	63.6	81.8	96.3	104.9	0.0	98.4	77.1	67.8	76.1	86.5
ammp	111.7	118.1	91.7	98.5	100.2	105.2	94.8	98.4	0.0	91.1	98.8	92.7	120.0
parser	73.6	89.7	**56.4**	65.0	88.9	94.4	87.6	77.1	91.1	0.0	77.4	**62.9**	89.7
vortex	92.0	68.5	79.2	**54.6**	87.8	92.7	101.3	67.8	98.8	77.4	0.0	94.8	73.1
bzip2	78.1	111.4	**45.7**	88.8	94.1	102.5	80.0	76.1	92.7	**62.9**	94.8	0.0	107.9
twolf	85.5	**35.2**	96.6	67.3	91.7	105.2	111.1	86.5	120.0	89.7	73.1	107.9	0.0

The diagonal of the table is all zeros because it represents the cases where a benchmark is compared against itself. For this example, if the similarity threshold was set to 63.2 (the square root of 4000), all pairs of benchmarks that had a distance less than 63.2 could be considered similar. The entries shown in bold correspond to benchmark pairs whose distances are less than the threshold. Each row in the following table, Table 11, shows the groups of benchmarks that have a similar effect on the processor, as determined by the above similarity distances.

Table 11: Benchmarks Grouped by Their Effect on the Processor

gzip, mesa
vpr-Place, twolf
vpr-Route, parser, bzip2
gcc, vortex
art
mcf
equake
ammp

The *gzip* and *mesa* benchmarks are defined to be similar because the distance between those two benchmarks, 62.0, is less than our arbitrary threshold value of 63.2. On the other hand *gzip* and *vpr-Place* are defined as dissimilar because their distance of 89.8 exceeds the threshold.

It is important to note that the benchmarks in Table 11 represent only one method of classifying benchmarks with one specific threshold value. It is also important to realize that key metrics, such as IPC and miss rates, could be quite different for benchmarks within the same set. The primary purpose of this section was to introduce an alternative method of classifying benchmarks that simultaneously considers the average impact of all processor parameters. Therefore, it is left to the experimenter to set the threshold value, to group the benchmarks accordingly, and to decide which benchmarks to select based on this method of classification and potentially, other metrics (e.g., IPC, miss rates, etc.).

4.3 Post-Simulation Methodology: Analysis of Processor Enhancements

For many computer architecture studies, analyzing the simulation results due to a processor enhancement involves examining the individual statistics (e.g. speedup, miss rate, functional unit utilization, etc.) without trying to determine the whole-picture effect of the enhancement on the processor. While the individual statistics may provide insight into the effect of the enhancement on key hardware structures, identifying all of the important individual statistics and trying to piece them together to form a higher-level view of the overall effect on the processor is daunting. Therefore, as a method of improving the methodology in the analysis of processor enhancements, we describe a method that simultaneously considers all components of the processor to give the user a higher-level view that is very easy to understand.

Our proposed method involves using a PB design to analyze the effect that a benchmark has on the processor parameters before and after the application of the enhancement. Therefore, by using this method, the user can determine the effect of the enhancement on the most significant parameters and also determine which of the enhancement's parameters are significant and how those parameters compare in significance to the processor's parameters without the enhancement.

The analysis of an enhancement uses the same steps as were used to determine the processor parameter values in Section 4.1. Namely, a PB design is used to determine the effect that each parameter has on the performance. The parameters are then ranked in descending order of significance. Then, to account for the significance of all benchmarks, the ranks for each parameter are summed together. The parameters with the lowest sum-of-ranks are the most significant parameters. Therefore, by comparing the sum-of-ranks for each parameter before and after the application of the enhancement, the user can determine how the enhancement affects the processor. For example, if the L1 D-Cache size and associativity sharply drop in significance due to an enhancement, it is reasonable to conclude that that particular enhancement does a good job of improving memory performance. However, this particular enhancement may also cause a sharp rise in the significance of the memory ports and the number of LSQ entries, for instance. Therefore, it would be reasonable to conclude that this particular enhancement improves the memory performance at the cost of increased pressure on the memory ports and the LSQ.

To illustrate the mechanics and effectiveness of this method, we analyze the effect that the instruction precomputation mechanism [Yi02-1] has on the processor. Instruction precomputation is similar to value reuse [Sodani97] in that it dynamically removes redundant computations from the pipeline by using a cached output value instead of computing the result. The key difference between instruction precomputation and value reuse is that instruction precomputation uses the compiler to statically identify the highest frequency redundant computations instead of using hardware. In instruction precomputation, the redundant computations are loaded into the on-chip precomputation table before the program begins execution and are never updated. By contrast, value reuse dynamically updates the on-chip value reuse table with the most current computations.

Table 12 shows the effect of instruction precomputation with a 128-entry precomputation table on the processor. While Table 12 represents the "after" case, Table 9 represents the "before" case, that is, the unenhanced processor.

By comparing these two tables, we can draw two conclusions about the effect of instruction precomputation on the processor. First of all, the same parameters that were significant for the base processor are also significant for the processor with instruction precomputation. Instruction precomputation changes the relative ordering of the significant parameters, with respect to each other, but does not change which parameters have the greatest significance. Secondly, of the significant parameters, the parameter that has the biggest change in its overall effect (defined as the biggest change in the sum of ranks) is the number of integer ALUs. This result is intuitively reasonable since most of the instructions that instruction precomputation eliminates would have executed on the integer ALUs. Therefore, by using instruction precomputation, the impact of the number of integer ALUs on the processor's performance decreases in significance.

In conclusion, applying this method to analyze simulation results has a few advantages over commonly-used approaches that simply look at a single metric, such as execution time. First of all, the exact effect that the enhancement has on the processor parameters can be determined, as was shown in the preceding example. This is especially useful in finding parameters that would seem to be unaffected by an enhancement, but actually turn out to be impacted by the enhancement. This result also can quickly point the experimenter to the areas of the processor that may require a more detailed analysis.

Second, by analyzing the impact of the enhancement on all of the parameters, the experimenter can determine which of those parameters is the most significant and how that parameter compares with the unenhanced processor's important parameters. This comparison can allow the experimenter to make design decisions as to how to maximize the performance and minimize the cost of the enhancement. Finally, using this method to analyze simulation results gives the analysis a statistically solid foundation that improves the overall quality of the analysis, in addition to improving the confidence in the final results and conclusions.

Table 12: Plackett and Burman Design Results for All Processor Parameters When Using Instruction Precomputation; Ranked by Significance and Sorted by the Sum of Ranks

Factor	gzip	vpr-Place	vpr-Route	gcc	mesa	art	mcf	equake	ammp	parser	vortex	bzip2	twolf	Sum
RUU Entries	1	4	1	4	3	2	2	3	6	1	4	1	4	36
L2 Cache Latency	4	2	4	2	2	4	4	2	13	3	2	8	2	52
BPred Type	2	5	3	5	5	28	11	8	4	4	16	7	5	103
L1 D-Cache Latency	7	6	5	7	11	8	14	5	40	7	5	4	6	125
L1 I-Cache Size	5	1	12	1	1	12	38	1	36	8	1	15	1	132
Int ALUs	6	8	8	9	8	29	9	13	20	6	9	3	9	137
L2 Cache Size	9	35	2	6	22	1	1	6	2	2	6	2	43	137
L1 I-Cache Block Size	15	3	20	3	14	10	32	4	10	11	3	20	3	148
Memory Latency First	35	25	6	8	18	3	3	7	1	5	7	6	27	151
LSQ Entries	13	14	9	10	15	40	10	9	17	9	8	5	10	169
D-TLB Size	21	28	11	24	25	13	12	10	25	14	25	10	24	242
Speculative Branch Update	8	20	25	29	7	16	39	11	8	20	21	22	19	245
L1 I-Cache Associativity	3	41	15	28	6	34	23	28	16	17	11	9	21	252
L1 D-Cache Size	18	7	10	12	42	19	8	35	32	21	13	32	7	256
FP Multiply Latency	31	12	22	11	19	24	15	22	24	28	14	24	18	264
Memory Bandwidth	33	36	13	14	43	6	6	31	3	12	20	11	38	266
BTB Entries	10	23	19	20	9	41	31	20	22	19	19	16	34	283
Int ALU Latencies	16	15	18	13	40	22	33	14	31	16	41	12	16	287
L1 D-Cache Block Size	17	30	34	22	16	9	24	19	26	13	33	25	26	294
Int Divide Latency	30	10	26	17	24	33	40	33	19	10	10	41	8	301
L2 Cache Associativity	23	19	14	19	33	27	5	39	37	18	42	21	12	309
Int Mult/Div	14	21	30	31	12	23	27	23	33	37	18	27	15	311
I-TLB Latency	32	17	24	18	34	30	30	16	21	33	12	29	17	313
Instruction Fetch Queue Entries	43	13	27	30	23	20	19	37	9	25	23	34	14	317
BPred Misprediction Penalty	11	24	41	21	4	43	20	32	11	22	39	35	23	326
FP Divide Latency	20	9	36	16	28	21	37	15	43	38	17	38	11	329
FP ALUs	34	11	31	15	38	17	41	24	27	36	15	43	13	345
I-TLB Page Size	42	38	7	38	39	39	7	17	12	26	28	14	39	346
L1 D-Cache Associativity	12	39	17	35	17	42	34	34	14	15	36	17	42	354
L2 Cache Block Size	25	43	16	37	31	7	35	27	7	35	38	13	40	354
I-TLB Associativity	26	27	38	25	20	31	42	12	29	30	22	33	22	357
BTB Associativity	22	18	35	32	10	32	17	30	34	43	27	36	25	361
D-TLB Associativity	40	32	23	26	27	35	25	26	18	32	26	28	35	373
Memory Ports	39	31	39	23	26	15	16	40	5	42	30	40	29	375
FP ALU Latencies	37	16	37	41	37	11	21	29	23	27	29	42	28	378
I-TLB Size	36	34	28	34	21	37	18	18	30	34	34	30	32	386
Dummy Factor #2	28	42	21	39	32	14	13	36	42	29	43	18	30	387
Int Multiply Latency	29	40	42	36	13	26	29	21	15	41	35	31	41	399
FP Mult/Div	41	22	43	40	41	18	28	38	28	31	31	19	20	400
FP Square Root Latency	38	29	40	33	35	5	26	43	41	24	24	39	37	414
Return Address Stack Entries	27	33	33	27	36	25	36	25	39	40	32	37	31	421
L1 I-Cache Latency	24	26	32	42	29	38	22	41	38	39	37	26	33	427
Dummy Factor #1	19	37	29	43	30	36	43	42	35	23	40	23	36	436

5 Related Work

While there are several previous studies that are related to this work, we were unable to find any previous work that directly focused on simulation methodology. Most of the related work focuses on simulator validation, while other related work focused on modeling the performance of processors through statistical methods or on improving the accuracy and precision of simulation results. Some previous work has performed sensitivity analyses of key processor parameters and has described a method for classifying benchmarks. The work presented in this paper complements this previous work by providing a basis for solid statistical analysis.

5.1 Simulator Validation, Processor Modeling, and Improving Simulator Accuracy

The authors of several papers have detailed their experiences with simulator validation. Black and Shen [Black98] described 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. Their results show that modeling, specification, and abstraction errors 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. Their work showed the need for extensive, iterative validation before the results from a performance model can be trusted.

Desikan *et al* [Desikan01] measured the amount of error that was present in an Alpha version of the SimpleScalar simulator. They defined the amount of error to be the difference in the simulated execution time and the execution time of the processor itself. 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.

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 simulated execution time reported by the FLASH simulator against the actual execution time of the FLASH processor. 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 if all of the important components have been accurately modeled. Furthermore, they 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 for specific architectural enhancements.

Finally, Glamm and Lilja [Glamm00] introduced a method of verifying the functional correctness of a simulated ISA. Their method of validation involved simultaneously executing a program's instructions on a simulator and on the targeted machine. After each instruction, the simulated processor's state was compared to the real machine's processor state. Any difference between the states indicated the presence of an error in the simulated ISA.

A few papers also have described statistical methods for reducing the complexity of a simulator and, thereby, the resulting simulation time. Noonburg and Shen [Noonburg97] described a method that uses a trace of a program along with probabilistic models to estimate the performance of the processor given a particular processor configuration. Their simulator consists of two main pieces, the processor model and the statistical model. The processor model accounts for the microarchitecture and the timing aspects by modeling the key components, such as buffers, pipelines, etc. The statistical model then uses probabilities to account for how long the instructions are in a particular state. By combining the two models and using the program trace as an input, they were able to achieve estimates of performance ranging from 1% to 10% of the processor's actual performance (as measured by the IPC) in a fraction of the time.

Oskin *et al* [Oskin00] described the HLS simulator that uses statistical profiles and symbolic execution to estimate processor performance. The statistical profile stores the basic block sizes, distribution, and the number of instructions separating the different instruction types (integer, floating-point, etc.). The symbolic code transforms the instruction stream into a control-flow graph of blocks. Within each block are the resource requirements that are necessary to execute the instructions that are represented by that block. The HLS simulator was reported to estimate to within 10% the performance of the SimpleScalar simulator.

Cain *et al* [Cain02] measured the effect of the operating system and the effects of I/O on simulator accuracy. To accomplish this task, they integrated the SimOS-PPC operating system with SimMP, a multiprocessor simulator. Their results showed that not including an operating system could introduce errors as high as 100% in the simulated performance. Furthermore, their results showed the potential for error due to I/O if the additional memory traffic is not properly taken into account. Overall, their results showed the need to integrate an operating system into the simulator for increased simulator accuracy and precision.

5.2 Analysis of the Effect of Key Processor Parameters

Skadron *et al* [Skadron99] performed an in-depth study of the trade-offs between the instruction-window size, branch prediction accuracy, and the L1 caches. Their paper performed a set of detailed sensitivity analyses that examined the IPC for different instruction-window sizes, data and instruction cache sizes, and different branch prediction accuracies using the integer benchmarks of the SPEC 95 benchmark suite. Their base simulator was a heavily modified version of the SimpleScalar simulator. While their results were very detailed and had several meaningful conclusions, they did not determine the important parameters and interactions before they performed their sensitivity analyses. Therefore, the conclusions that were drawn from these results cannot be taken completely at face value without determining if any of the constant parameters or interactions are significant enough to affect the results.

5.3 Benchmark Classification

Giladi and Ahituv [Giladi95] identified the “redundant” benchmarks in the SPEC 92 benchmark suite. A redundant benchmark was defined to be one that can be removed from the benchmark suite without significantly affecting the resulting SPEC number for that benchmark suite. The SPEC number theoretically measures the performance of a computer system across a wide range of programs. The SPEC number is generated by normalizing each benchmark’s execution time to a baseline system and then a geometric mean formula is used to average all the normalized times. Their results show that 13 of the 20 benchmarks in the SPEC 92 suite were redundant. This method of determining redundant benchmarks is significantly different from our proposed method for at least two reasons. First of all, this method is completely based on approximating the SPEC number. Secondly, since the SPEC number is calculated by using the benchmark’s execution and by normalizing the execution times to a baseline system, there is no direct connection to the effect that each benchmark has on the processor. However, our method focuses exclusively on the benchmark’s effect on the processor.

6 Conclusion

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, due to a lack of a formalized methodology, most current methods approach simulation methodology in an ad-hoc fashion. As a result, unnecessary errors arise, such as using poorly chosen processor parameter values or using poorly chosen sets of benchmark programs. Furthermore, without a more formalized methodology, computer architecture researchers may not glean as much information as possible from their extensive simulation results. Finally, by adding statistical rigor to our methodology, we as a community can have more confidence in our simulation results.

As a first step in developing a formalized simulation methodology, this paper describes three methods of improving the simulation methodology in computer architecture research. The first two methods seek to improve the simulation setup while the third method seeks to improve the analysis of the results. The first method focuses on how the processor parameter values are chosen. In particular, instead of using the current ad-hoc methods of choosing processor parameters, this method advocates using a Plackett and Burman (PB) design to determine the most important parameters. The values for these key parameters need to be chosen with care since the specific value chosen can seriously affect the performance results. Less caution needs to be exercised when choosing values for the remaining parameters.

The second method to improve simulation methodology focuses on benchmark selection. Our proposed method of benchmark selection groups the benchmarks based on the effect that each benchmark has on the processor. Two benchmarks have similar effects on the processor if their sets of processor parameters have similar ranks. As with the processor parameter selection, a PB design is used to determine the effect that a benchmark has on the processor.

Finally, the last method focuses on improving the analysis of the simulation results in the post-simulation phase. This method uses a PB design to rank the parameters before and after an enhancement is added to the processor. By comparing the before and after ranks, the effect that the enhancement has on the processor can be readily determined.

In conclusion, there is plenty of room for improvement with the current simulation methodology. Adopting some or all of the methods described in this paper can significantly improve the quality of, and confidence in, simulation results.

Acknowledgements

This work was supported in part by National Science Foundation grants CCR-9900605 and EIA-9971666, the IBM Corporation, Compaq's Alpha Development Group, and the Minnesota Supercomputing Institute.

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.
[Cain02]	H. Cain, K. Lepak, B. Schwartz, and M. Lipasti; "Precise and Accurate Processor Simulation "; Workshop on Computer Architecture Evaluation using Commercial Workloads, 2002.
[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.

[Giladi95]	R. Giladi and N. Ahituv; "SPEC as a Performance Evaluation Measure"; IEEE Computer, Vol. 28, No. 8, August 1995; Pages 33-42.
[Glamm00]	R. Glamm and D. Lilja; “Automatic Verification of Instruction Set Simulation Using Synchronized State Comparison”; Annual Simulation Symposium, 2001.
[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.
[KleinOsowski02]	A. KleinOsowski and D.J. Lilja; "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research."; Vol. 1, June 2002.
[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.
[Montgomery91]	D. C. Montgomery; “Design and Analysis of Experiments” (Third Edition), Wiley 1991.
[Noonburg97]	D. Noonburg and J. Shen; "A Framework for Statistical Modeling of Superscalar Processor Performance "; International Symposium on High Performance Computer Architecture, 1997.
[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.
[Oskin00]	M. Oskin, F. Chong, and M. Farrens; "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs"; International Symposium on Computer Architecture, 2000.
[Plackett46]	R. Plackett and J. Burman; "The Design of Optimum Multifactorial Experiments"; Biometrika, Vol. 33, Issue 4, June 1956; Pages 305-325.
[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.
[Skadron99]	K. Skadron, P. Ahuja, M. Martonosi, and D. Clark; "Branch Prediction, Instruction-Window Size, and Cache Size: Performance Trade-Offs and Simulation Techniques"; IEEE Transactions on Computers, Vol. 48, No. 11, November 1999; Pages 1260-1281.
[Sodani97]	A. Sodani and G. Sohi; "Dynamic Instruction Reuse"; International Symposium on Computer Architecture, 1997.
[Song94]	S. Song, M. Denman, and J. Chang; "The PowerPC 604 RISC Microprocessor"; IEEE Micro, Vol. 14, No. 5, October 1994; Pages 8-17.
[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.
[Yi02-1]	J. Yi, R. Sendag, and D. Lilja; "Increasing Instruction-Level Parallelism with Instruction Precomputation"; Euro-Par 2002.
[Yi02-2]	J. Yi and D. Lilja; " Effects of Processor Parameter Selection on Simulation Results"; MSI tech report #(pending).