Improving Computer Architecture Simulation Methodology by Adding Statistical Rigor

Joshua J. Yi, *Member*, *IEEE*, David J. Lilja, *Senior Member*, *IEEE*, and Douglas M. Hawkins

Abstract—Due to cost, time, and flexibility constraints, computer architects use simulators to explore the design space when developing new processors and to evaluate the performance of potential enhancements. However, despite this dependence on simulators, statistically rigorous simulation methodologies are typically not used in computer architecture research. A formal methodology can provide a sound basis for drawing conclusions gathered from simulation results by adding statistical rigor and, consequently, can increase the architect's 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 enhancements. Our results showed that, out of the 41 user-configurable parameters in SimpleScalar, only 10 had a significant effect on the execution time. Of those 10, the number of reorder buffer entries and the L2 cache latency were the two most significant ones, by far. Our results also showed that Instruction Precomputation—a value reuse-like microarchitectural technique—primarily improves the processor's performance by relieving integer ALU contention.

Index Terms—Performance analysis and design aids, measurement techniques, simulation output analysis.

1 Introduction

The most important tool in processor design and computer architecture research is the simulator. Using a simulator reduces the cost and time of a project by allowing the architect to quickly evaluate the performance of different processor configurations instead of fabricating a new processor for each one, a process that takes years and is extraordinarily expensive. Additionally, a simulator is much more flexible than fabricating the processor since it can accurately determine the expected performance of a processor enhancement without having to undergo all the necessary circuit-level design steps. Consequently, without simulators, designing processors would either be too expensive or would yield very poor designs.

Computer architects also use simulators to guide design decisions, determine what points to explore in the design space, and to quantify the efficacy of a processor enhancement. Consequently, since misleading simulation results can severely affect the final design of the processor or lead to erroneous conclusions, the accuracy of the simulator's results is extremely important. Therefore, to minimize the amount of error in the simulation results, computer architects need to do two things. First, they should try to minimize the amount of error inherent to the simulator (as compared to the silicon version of the processor that the simulator models). Second, they should try to reduce the amount of "error" that they introduce when running simulations. An architect may introduce additional error into the simulation results by choosing a poor set of parameter values or benchmarks, thus altering the processor's apparent performance. While current research also focuses on decreasing the processor's power consumption and improving the processor's fault tolerance, for brevity, the remainder of this paper assumes that the architect is trying only to improve the processor's performance. However, the techniques that are described in this paper are equally applicable to power consumption reduction and improving the processor's reliability.

In spite of this dependence on simulators, relatively little research has focused on decreasing the amount of error in simulation results by improving the accuracy of simulators or by improving simulation methodology. In fact, current simulation methodology is, at best, ad hoc. Therefore, to decrease the amount of error in the simulation results and also to improve the overall quality of simulation methodology, this paper advocates using rigorous, statistically-based simulation methodology.

While the downside of using such a methodology is that it may require some additional simulations, it also has the following advantages:

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 [2], [4], [6].
2. It gives more insight into what is occurring inside the processor or the actual effect of a processor enhancement.
3. It gives objective confidence to the results and provides statistical support regarding the observed behavior.

- *J.J. Yi is with the Networking and Computing Systems Group, Freescale Semiconductor, Inc., Austin, TX 78729. E-mail: joshua.yi@freescale.com.*
- *D.J. Lilja is with the Dept. of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455. E-mail: lilja@ece.umn.edu.*
- *D.M. Hawkins is with the School of Statistics, University of Minnesota, Minneapolis, MN 55455. E-mail: doug@stat.umn.edu.*

Manuscript received 28 Dec. 2003; revised 22 Apr. 2005; accepted 4 May 2005; published online 16 Sept. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0298-1203.

More specifically, this paper improves the simulation methodology used by computer architects by recommending specific procedures on how to: 1) choose the parameter values, 2) select a subset of benchmarks, and 3) analyze the effect that an enhancement has on the processor. The first two recommendations target the simulation setup phase of the simulation process, while the last recommendation targets the analysis phase. To illustrate the efficacy of the last recommendation, this paper analyzes the effect that two processor enhancements, Instruction Precomputation [21] and the Simplification and Elimination of Trivial Computations [22], have on the processor.

The contributions of this paper are as follows:

1. This paper motivates the need for methodological improvement in computer architecture research and design.
2. This paper makes specific recommendations on how to improve simulation methodology. In particular, the recommendations include how to:
   a. choose the parameter values,
   b. select a subset of benchmarks, and
   c. analyze the effect that an enhancement has on the processor.

   Collectively, these recommendations can improve the simulation methodology, decrease the total number of simulations, quickly determine the processor's performance bottlenecks, and provide analytical insights into the impact of processor enhancements.

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

2 The Plackett and Burman Design

To determine the effect that a parameter has on the processor's performance, in this paper, we used a Plackett and Burman (PB) design [14]. While we could have used one of several other statistical techniques, we chose the PB design because it required only about N simulations (where N is the number of parameters) to produce the desired level of information. The other approaches that we considered using were the "one-at-a-time" technique and the ANOVA technique [11]. However, these two techniques did not produce the desired level of information (one-at-a-time) or required too many simulations (2^N) simulations (ANOVA).

The PB design is one well-known type of a fractional multifactorial design. The base PB design requires X simulations, where X is the next multiple of four greater than N. For example, if $N = 3$, then $X = 4$; if $N = 16$, then $X = 20$. An improvement on the base PB design is the PB design with foldover [12]. This doubles the number of required simulations to $2 * X$.

A PB design with foldover can accurately quantify the effect that single parameters and two-parameter interactions have on the output value. But, it cannot quantify the effect of interactions that are composed of three or more parameters. While this may appear to be a major problem for computer architects, it is not. In general, if an interaction has a significant effect, it is likely that that interaction is significant because one of its parameters is a single parameter with a significant effect. The results from [23] show that, for a small number of parameters (10 total), the most significant interactions are the results of significant single parameters. Therefore, in computer architecture, it is likely that dominant interactions are the result of one or more dominant parameters. These conclusions are an example of what the statistical literature calls the "sparsity of effects." This phenomenon holds that, typically, only a small proportion of the parameters have a large effect on the output value and an even smaller proportion of the possible interactions are relevant. Exploiting this phenomenon, the PB design reduces the number of configurations that need to be simulated from 2^N by studying only enough configurations to quantify all effects and provide an indication of where there may be interactions.

2.1 Mechanics of the Plackett and Burman Design

For each test case in the PB design, the value of each parameter is given by the PB design matrix. For most values of X, the PB design matrix is simple to construct. Each row of the design matrix specifies if the parameter is set to its high or low value for that configuration. For a PB design with foldover, there are $2 * X$ rows, or configurations; only X configurations are needed when using a PB design without foldover. Each column specifies the values that a parameter is set to for each configuration. With or without foldover, there are always $X - 1$ columns in the design matrix. When there are more columns than parameters (i.e., $N < X - 1$), then the extra columns are simply "dummy parameters" and have no effect on the simulation results.

For most values of X, the first row of the design matrix is given in [14]. The value of each entry in the design matrix is either "$+1$" or "-1," where $+1$ corresponds to the parameter's high value and -1 corresponds to its low value. The next $X - 2$ rows are formed by performing a circular right shift on the preceding row. Finally, without foldover, the last row of the design matrix (row X) is a row of minus ones, which corresponds to the base case. The gray-shaded portion of Table 1 illustrates the construction of the PB design matrix for $X = 8$, i.e., suitable for $N = 4, 5, 6, 7$.

When using foldover, X additional rows are added to the matrix. The signs of the values in each of these additional rows are the opposite of the corresponding entries in the original matrix; the corresponding row is X rows above that row. Consequently, the last row, Row $2 * X$, is a row of plus ones. Table 1 shows the complete PB design matrix with foldover. Note that rows 9 to 16 specifically show the additional foldover rows.

The high value $(+1)$ for a parameter represents a value that is slightly higher than the range of normal values for that parameter, while the low value (-1) represents a value that is slightly lower than the range of normal values. It is important to note that the high and low values are not restricted to numerical values only. For example, in the case of branch prediction, the high and low "values" could be perfect and two-level branch prediction, respectively. It is also important to note that choosing high and low values

TABLE 1
PB Design Matrix with Foldover $(X = 8)$ for Parameters A to G

A	B	C	D	E	F	G	Execution Time
+1	+1	+1	-1	+1	-1	-1	9
-1	+1	+1	+1	-1	+1	-1	11
-1	-1	+1	+1	+1	-1	+1	20
+1	-1	-1	+1	+1	+1	-1	10
-1	+1	-1	-1	+1	+1	+1	9
+1	-1	+1	-1	-1	+1	+1	74
+1	+1	-1	+1	-1	-1	+1	7
-1	-1	-1	-1	-1	-1	-1	112
-1	-1	-1	+1	-1	+1	+1	17
+1	-1	-1	-1	+1	-1	+1	76
+1	+1	-1	-1	-1	+1	-1	6
-1	+1	+1	-1	-1	-1	+1	31
+1	-1	+1	+1	-1	-1	-1	19
-1	+1	-1	+1	+1	-1	-1	33
-1	-1	+1	-1	+1	+1	-1	6
+1	+1	+1	+1	+1	+1	+1	4
-34	-224	-96	-202	-110	-170	32	

that yield too large a range can artificially inflate the parameter's apparent effect. On the other hand, too small a range of values for a parameter means that that parameter will have very little or no effect on the output. However, having too large a range is better than having too small a range because that ensures that that parameter will have an effect on the output value. In any case, the computer architect should carefully choose the high and low values for each parameter that are just outside of the "normal" range of values.

After determining the configurations and performing the simulations, the effect of each parameter is computed by multiplying the parameter's corresponding $+1$ or -1 for that configuration by the output value (e.g., execution time) for that configuration and summing the resulting products across all configurations. For example, given the execution times in the rightmost column in Table 1, the effect of parameter A is computed as follows:

$$\text{Effect}_{\text{A}} = (1 * 9) + (-1 * 11) + (-1 * 20) + \ldots + (-1 * 33) + (-1 * 6) + (1 * 4) = -34.$$

By performing the same computation for each parameter, the results in Table 1 show that the parameters that have the most effect on the execution time are B, D, and F, in descending order of their overall impact on the execution time. Only the magnitude of the effect is important in determining relative importance; the sign of the effect is essentially meaningless.

The effect that a parameter has represents how much of the total variation in the output value is attributable to that parameter. Therefore, a parameter that has a large effect on the execution time accounts for a significant amount of variability in the execution time. For those parameters, since they have a large effect on the execution time (or else they would not cause large variations in the execution time), they represent significant performance bottlenecks (since changing the parameter from its low value to its high value results in large changes in the execution time).

After computing the magnitude of the effect for each parameter, the parameters can be ranked based on their magnitudes (1 = most important, $X - 1$ = least important). Since the execution time, in cycles, for the same processor configuration, can be very different across benchmarks, the magnitudes of the effects reflect those differences. Consequently, ranking the parameters by their magnitudes allows for comparisons between benchmarks, which would not be possible due to the large differences in the execution times.

Obviously, mapping the PB magnitudes to ranks has advantages and disadvantages. The first advantage is that it prevents the most significant parameter from dominating the results. For example, assume that the effects of all parameters were normalized to the largest effect. If the magnitude of that effect is large enough, then the remaining parameters would essentially have zero *relative* effect although their *absolute* effect may be nonnegligible. By using ranks, less significant parameters still have an effect. The second advantage of using ranks instead of using a scaled version of the effects is that it is simple to use and provides a common set of values for all benchmarks.

The obvious disadvantage of ranks is that they may distort the results due to "quantization" error. For example, assume that the PB magnitudes of the 10 most significant parameters are all virtually equal. By using ranks, instead of normalizing all effects to the magnitude of the most dominant parameter, there is a difference of nine in the ranks of the most significant parameter and the 10th most, despite their magnitudes being virtually identical.

Therefore, due to their inherent advantages and disadvantages, one needs to use ranks judiciously. (In this paper, we checked a few parameters and benchmarks to determine the effect that ranking had. Our checks showed that the ranks did not seem to significantly affect our results.)

Collectively, the ranks of all parameters form a vector of ranks, one for each benchmark. These vectors are used as the basis for improving the simulation methodology. In particular, the three subsections of Section 4 explain how these vectors can be used to improve the way in which parameter values are chosen, benchmarks are selected, and the effect of an enhancement is analyzed.

2.2 Guidelines for Plackett and Burman Design Usage for Computer Architecture Research

As described in the previous section, the PB design is able to accurately quantify the effects of all single parameters and, with foldover, selected (i.e., two-factor) interactions. Without foldover, the effects of all interactions—ranging from two-factor to N-factor—are randomly mixed into the effects of the individual, single parameters. The PB design with foldover allows the user to quantify the effects of the two-factor interactions and filters out the effects of the two-factor interactions from the effects of the single parameters. Consequently, the first key point is that the PB design is best suited for experimental designs where the effects of interactions are relatively low (i.e., most of the variability in the output value is due to single parameters) and/or primarily due to two-factor interactions. The PB design may produce misleading results when interactions composed of three or more parameters are the most significant effects, especially if each of those parameters is not a significant single parameter. Fortunately, in most cases, there is

sparsity of effects such that single parameters and two-factor interactions will be the most significant parameters and interactions; the results in this paper confirm that that is the case for computer architecture.

It is important to note that using two values only for each parameter, i.e., high and low values, only tracks the *net* effect that a parameter has. Suppose that the execution time decreases as the parameter's value is increased from its low value to a middle value, but increases back to the original execution time as the parameter's value changes from its middle value to its high value. In this situation, this parameter would appear to have no effect, but, in actuality, it has no *net* effect since the execution time increased after the parameter was set to its high value. An example of this kind of parameter could be the number of branch history table (BHT) entries. When there are very few entries, many branches may map to the same BHT entries and destructively interfere with each other. On the flip side, if there are too many entries, it may take too many branches to adequately train the BHT or eliminate the possibility of constructive interference. However, an intermediate number of entries may be the perfect balance between constructive interference and long training times, which would then result in the lowest execution time.

There are two potential solutions to minimize this problem. First, the architect can either use three values (low, medium, and high) instead of just two. Obviously, using three values would allow the architect to track the intermediate values that may reveal effects that are not linearly increasing. A second solution, instead of increasing the number of simulations by 50 percent, is to rely on the architect's intuition to realize if the effects of some parameters are likely to be monotonic or not.

It is important to reiterate that, when choosing high and low parameter values, the most important thing is to choose a pair of values that spans a large enough range of values to ensure that the parameter has an effect on the output value. In other words, it is preferable to choose too large a range rather than one that is too small. However, the competing factor is that too large a value will cause that parameter to have a disproportionately large effect on the execution time. Therefore, when choosing high and low parameter values, it is important for architects to use their intuition about that parameter's effect on the output value to select the values.

For some parameters, the high and low values may not necessarily be constant, i.e., the high value may not always be the value for that parameter that yields the best output value (e.g., lowest execution time). For example, one such parameter may be the cache replacement policy where LRU may be the high "value" for some benchmarks, but the low "value" for others, while a random replacement policy may be the high "value" for other benchmarks and the low "value" for others. Fortunately, the PB design is not affected by these kinds of parameters since each parameter is set to its high and low values for half of the configurations.

Finally, when foldover is not used, instead of simulating what corresponds to the top half of Table 1, the bottom half of the table can be substituted in its place to decrease the simulation time. From a statistical point of view, the bottom half of the PB design matrix is the logical equivalent of the top half. Therefore, substituting one half of the table for the other produces the same results. However, the benefit of using the bottom half of the table is that, in the last configuration, each parameter value set to its high value, which results in a lower cycle count than when each parameter is set to its low value, and, presumably, a lower simulation time.

TABLE 2
Benchmarks from the SPEC 2000
Benchmark Suite Used in This Paper

Benchmark	Type	Dynamic Instructions (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

3 Simulator, Benchmarks, and Processor Parameter Values

In the remainder of this paper, we illustrate the efficacy of the PB design in improving simulation methodology. The base simulator, `sim-outorder`, is from the SimpleScalar tool suite [3] and models a superscalar processor. We modified `sim-outorder` to include user configurable instruction latencies and throughputs. Table 2 lists the benchmarks and input sets that were used in this paper.

As described in Section 2, the parameter values should be chosen to be slightly too low and too high—with respect to the normal range—to allow the PB design to more accurately determine the effect of each parameter. As a result, the final values that we chose for each parameter are not values that would actually be present in commercial processors nor are they values that should be used in other simulations. We based our parameter values on those found in several commercial processors, including the Alpha 21164 [1] and 21264 [8]; the UltraSparc I [19], II [13], and III [7]; the HP PA-8000 [10]; the PowerPC 604 [18]; and the MIPS R10000 [20]. To fill in the gaps left by these papers, [15], [16], and several Web searches were also used as references. Tables 3, 4, and 5 show the final values for each of the relevant parameters in the processor core, the functional units, and the memory hierarchy, respectively.

A few parameters in these three tables are shaded in gray. For these parameters, the high and low values cannot be chosen completely independently of the other parameters due to the mechanics of the PB design. The problem occurs when one of those parameters is set to its high value while the parameter that it is related to is set to its low

TABLE 3
Processor Core Parameters and Their Plackett and Burman Values

Parameter	Low Value	High Value
Fetch Queue Entries	4	32
Branch Predictor	2-Level	Perfect
Branch Misprediction Penalty	10 Cycles	2 Cycles
RAS Entries	4	64
BTB Entries	16	512
BTB Assoc	2-Way	Fully-Assoc
Speculative Branch Update	In Commit	In Decode
Decode/Issue Width	4-Way	
ROB Entries	8	64
LSQ Entries	0.25 * ROB	1.0 * ROB
Memory Ports	1	4

value. This combination of values 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 was chosen independently of the number of ROB entries, then some of the configurations could have a 64-entry LSQ and an eight-entry ROB. Since the total number of in-flight instructions cannot exceed the number of ROB entries, the maximum number of filled LSQ entries will never exceed eight, which is just like setting the high value of the number of LSQ entries to eight. In this case, the effect of the number of LSQ entries is artificially constrained by the number of ROB entries. On the other hand, we did not base the cache access latency on the cache size, associativity, and block size for two key reasons. First, since there are two values for each parameter, there are a total of eight cache configurations and, since there is both a high and a low value for the cache access latency for each cache configuration, we could potentially have 16 different cache access latencies. For certain cache configurations, some of the "low" values may be lower than some of the "high" values (remember that the high value for cache access latency is a smaller number than the low value). In these cases, the "high" and "low" values are "flipped" around, which makes it impossible for the PB design to accurately determine the effect of the cache access latency. Second, unlike the ROB/LSQ situation, where the low value of one parameter effectively changes the high value of the other parameter, none of these three parameters has that same effect. In conclusion, the rule-of-thumb that we used to decide if one parameter depended on another was if the value of one parameter constrained a value of the other parameter, as was the case for the number of ROB and LSQ entries.

TABLE 4
Functional Unit Parameters and Their Plackett and Burman Values

Parameter	Low Value	High Value
Int ALUs	1	4
Int ALU Latency	2 Cycles	1 Cycle
Int ALU Throughput	1	
FP ALUs	1	4
FP ALU Latency	5 Cycles	1 Cycle
FP ALU Throughput	1	
Int Mult/Div Units	1	4
Int Mult Latency	15 Cycles	2 Cycles
Int Div Latency	80 Cycles	10 Cycles
Int Mult Throughput	1	
Int Div Throughput	1 / Equal to Int Div Latency	
FP Mult/Div Units	1	4
FP Mult Latency	5 Cycles	2 Cycles
FP Div Latency	35 Cycles	10 Cycles
FP Sqrt Latency	35 Cycles	15 Cycles
FP Mult Throughput	Equal to 1/ FP Mult Latency	
FP Div Throughput	Equal to 1 / FP Div Latency	
FP Sqrt Throughput	Equal to 1 / FP Sqrt Latency	

TABLE 5
Memory Hierarchy Parameters and Their Plackett and Burman Values

Parameter	Low Value	High Value
L1 I-Cache Size	4 KB	128 KB
L1 I-Cache Assoc	1-Way	8-Way
L1 I-Cache Block Size	16 Bytes	64 Bytes
L1 I-Cache Repl Policy	Least Recently Used	
L1 I-Cache Latency	4 Cycles	1 Cycle
L1 D-Cache Size	4 KB	128 KB
L1 D-Cache Assoc	1-Way	8-Way
L1 D-Cache Block Size	16 Bytes	64 Bytes
L1 D-Cache Repl Policy	Least Recently Used	
L1 D-Cache Latency	4 Cycles	1 Cycle
L2 Cache Size	256 KB	8192 KB
L2 Cache Assoc	1-Way	8-Way
L2 Cache Block Size	64 Bytes	256 Bytes
L2 Cache Repl Policy	Least Recently Used	
L2 Cache Latency	20 Cycles	5 Cycles
Mem Latency, First	200 Cycles	50 Cycles
Mem Latency, Next	0.02*Mem Latency, First	
Mem Bandwidth	4 Bytes	32 Bytes
I-TLB Size	32 Entries	256 Entries
I-TLB Page Size	4 KB	4096 KB
I-TLB Assoc	2-Way	Fully Assoc
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 Assoc	2-Way	Fully-Assoc
D-TLB Latency	Same as I-TLB Latency	

All parameter values were based on a four-way issue processor. While the issue width is a very important parameter, we fixed the issue width at four for two reasons. First, we fixed it to avoid having a set of high and low values for each issue width since almost all of the parameters are related to the issue width. Having two sets of high and low values could dramatically affect the results. Second, we fixed issue width to eliminate the guesswork needed to determine the normal range of parameter values for a higher issue width processor since there is very little documentation available for processors with an issue width greater than four.

4 PB Design Results for the Simulation Setup and Analysis

In computer architecture research, the simulation process is the sequence of steps that architects must perform to run their simulations and to analyze their simulation results. Most academic computer architects start with a publicly available simulator, such as SimpleScalar, and then add their own code to the simulator to model their enhancement. This paper divides the simulation process into six major steps. They are:

1. simulator implementation and validation,
2. processor enhancement implementation and verification,
3. processor and memory hierarchy parameter value selection,
4. benchmark and input set selection,
5. simulation, and
6. analysis of an enhancement's effect.

This paper focuses on improving the simulation methodology of the third step, processor parameter selection, the fourth step, benchmark selection, and the last step, the analysis of an enhancement's effect. This paper excludes the first step (simulator implementation and validation) for two reasons. First, since most computer architects do not implement their own simulator, but rather use publicly available simulators as their base simulator, focusing on this step benefits only a small number of computer architects. Second, as is described in Section 5, there have been a few papers that have focused on improving the accuracy of simulators. For similar reasons, this paper does not focus on the second (processor enhancement implementation and verification) and fifth (simulation) steps.

4.1 Processor Parameter Selection

Choosing the processor parameter values for simulation is the third step of the simulation process. Choosing a "good" set of values is very important since improperly choosing the value of even a single parameter can significantly affect the simulated speedup of a processor enhancement. For example, simply increasing the reorder buffer (ROB) size can change the apparent *speedup* of value reuse [23] from approximately 20 percent to approximately 30 percent.

However, choosing a good set of parameters is extremely difficult since many of the important parameters may interact, thereby compounding the error of selecting a single poor value. Determining which parameters interact requires performing a sensitivity analysis on all of the parameters simultaneously or choosing a select few parameters for detailed study. The problem with the former approach is that simulating all possible combinations is virtually impossible due to time limitations. The problem with the latter approach is that, in studying only a few parameters, the other parameters have to have constant values. Therefore, if one of the constant parameters significantly interacts with one of the free parameters, then the results of the sensitivity analysis will be distorted.

A related problem is efficient design space exploration. Due to the large number of variables (processor, memory hierarchy, and system parameters; compiler options; etc.), the number of points in the design space is extremely large, which makes it virtually impossible to simulate all of them. However, reducing the number of points in the design space to a tractable level is difficult since it is difficult to separate the less significant parameters from ones that are slightly more significant.

A PB design solves both of these problems by quantifying the effect of all single parameters and two-factor interactions. To determine which single parameters are the most significant ones, for each parameter, the ranks of that parameter across all benchmarks are first averaged together and then the averages are sorted in ascending order. Consequently, the parameter with the lowest average rank corresponds to, across all benchmarks, the parameter that has the most effect on the variation in the execution time. Then, by examining the average rank for each parameter, the computer architect can determine which parameters have the most effect on the execution time and can then carefully choose values for those parameters.

In the same way that mapping the PB magnitudes to ranks has its advantages and disadvantages, averaging the ranks across benchmarks has its own advantages and disadvantages. Averaging has the twin advantages of simplicity and equal benchmark weighting. However, one of its disadvantages is that it assigns the same weight to each benchmark, where, in some cases, some benchmarks should have a larger weight (e.g., benchmarks that are more frequently run). In this paper, we opt for simplicity to more clearly focus on the technique rather than the efficacy of specific mechanics. Nevertheless, it is important to realize that averaging the ranks across benchmarks may not be appropriate for each architect's situation.

More formally, this paper recommends using the following steps as a guide when choosing processor and memory hierarchy parameter values:

1. Determine the most significant processor parameters using a PB design.
    a. Choose high and low values for each of the parameters.
    b. Run and analyze the PB simulations to determine the critical parameters.
2. Iteratively perform sensitivity analyses for each critical parameter using the ANOVA design.
3. Choose final values for the significant parameters based on the results of the sensitivity analyses.
4. Choose the final values for the remaining parameters based on commercial processor values or some other appropriate source.

Of these four steps, the most important step is the first step. In this step, the computer architect uses a PB design to determine the most significant parameters. The second step is optional, depending on the results of the first step. If the first step shows that there are relatively few significant parameters, then second step is not necessary. Finally, in the third and fourth steps, the computer architect chooses the final values for all parameters based on the results of the first two steps and based on commercial parameter values. When choosing the values for each parameter, it is important to choose values that produce a balanced

TABLE 6
Plackett and Burman Design Results for All Processor Parameters, Ranked by Significance and Sorted by the Average Rank

Parameter	gzip	vpr-Place	vpr-Route	gcc	mesa	art	mcf	equake	ammp	parser	vortex	bzip2	twolf	Ave
ROB Entries	1	4	1	4	3	2	2	3	6	1	4	1	4	**2.8**
L2 Cache Latency	4	2	4	2	2	4	4	2	13	3	2	8	2	**4.0**
Branch Predictor	2	5	3	5	5	27	11	6	4	4	16	7	5	**7.7**
Int ALUs	3	7	5	8	4	29	8	9	19	6	9	2	9	**9.1**
L1 D-Cache Latency	7	6	7	7	12	8	14	5	40	7	5	6	6	**10.0**
L1 I-Cache Size	6	1	12	1	1	12	37	1	36	8	1	16	1	**10.2**
L2 Cache Size	9	35	2	6	21	1	1	7	2	2	6	3	43	**10.6**
L1 I-Cache Block Size	16	3	20	3	16	10	32	4	10	11	3	22	3	**11.8**
Memory Latency First	36	25	6	9	23	3	3	8	1	5	8	5	28	**12.3**
LSQ Entries	12	14	9	10	13	39	10	10	17	9	7	4	10	**12.6**
Speculative Branch Update	8	17	23	28	7	16	39	12	8	20	22	20	17	**18.2**
D-TLB Size	20	28	11	23	29	13	12	11	25	14	25	11	24	**18.9**
L1 D-Cache Size	18	8	10	12	39	18	9	36	32	21	12	31	7	**19.5**
L1 I-Cache Associativity	5	40	15	29	8	34	23	28	16	17	15	9	21	**20.0**
FP Multiply Latency	31	12	22	11	19	24	15	23	24	29	14	23	19	**20.5**
Memory Bandwidth	37	36	13	14	43	6	6	29	3	12	19	12	38	**20.6**
Int ALU Latencies	15	15	18	13	41	22	33	14	30	16	41	10	16	**21.8**
BTB Entries	10	24	19	20	9	42	31	20	22	19	20	17	34	**22.1**
L1 D-Cache Block Size	17	29	34	22	15	9	24	19	28	13	32	28	26	**22.8**
Int Divide Latency	29	10	26	16	24	32	41	32	20	10	10	43	8	**23.2**
Int Mult/Div	14	20	29	31	10	23	27	24	33	36	18	26	15	**23.5**
L2 Cache Associativity	23	19	14	19	32	28	5	39	37	18	42	21	12	**23.8**
I-TLB Latency	33	18	24	18	37	30	30	16	21	32	11	29	18	**24.4**
Instruction Fetch Queue Entries	43	13	27	30	26	20	18	37	9	25	23	34	14	**24.5**
Branch Misprediction Penalty	11	23	42	21	6	43	20	34	11	22	39	37	23	**25.5**
FP ALUs	34	11	31	15	34	17	40	22	26	37	13	42	13	**25.8**
FP Divide Latency	22	9	35	17	30	21	38	15	43	38	17	39	11	**25.8**
I-TLB Page Size	42	39	8	37	36	40	7	17	12	26	28	14	39	**26.5**
L1 D-Cache Associativity	13	38	17	34	18	41	34	33	14	15	35	15	42	**26.8**
I-TLB Associativity	24	27	37	25	17	31	42	13	29	30	21	33	22	**27.0**
L2 Cache Block Size	25	43	16	38	31	7	35	27	7	35	38	13	40	**27.3**
BTB Associativity	21	21	36	32	11	33	17	31	34	43	27	35	25	**28.2**
D-TLB Associativity	40	32	25	26	22	35	26	26	18	33	26	30	35	**28.8**
FP ALU Latencies	32	16	38	41	38	11	22	30	23	27	30	40	29	**29.0**
Memory Ports	39	31	41	24	27	15	16	41	5	42	29	41	27	**29.1**
I-TLB Size	35	34	28	35	20	37	19	18	31	34	34	27	31	**29.5**
Dummy Factor #2	27	42	21	39	35	14	13	35	41	28	43	18	30	**29.7**
FP Mult/Div	41	22	43	40	40	19	28	38	27	31	31	19	20	**30.7**
Int Multiply Latency	30	41	39	36	14	26	29	21	15	41	37	32	41	**30.9**
FP Square Root Latency	38	30	40	33	33	5	25	42	42	24	24	38	37	**31.6**
L1 I-Cache Latency	26	26	32	42	28	38	21	40	38	40	36	25	33	**32.7**
Return Address Stack Entries	28	33	33	27	42	25	36	25	39	39	33	36	32	**32.9**
Dummy Factor #1	19	37	30	43	25	36	43	43	35	23	40	24	36	**33.4**

processor configuration. In other words, the architect should choose parameter values that are large enough to minimize the effects of the biggest performance bottlenecks, while not selecting values that are too large, i.e., that will be underutilized, dissipate a disproportionate amount of power compared to its performance benefit, and consume an unwarranted amount of area. Therefore, while the PB design can help an architect identify the most sensitive parameters, the architect needs to carefully choose the final parameter values to ensure that the processor is not unbalanced.

When using this approach to reduce the design space, it is only necessary to do the first step—determine the most significant parameters—since this step reduces the maximum number of parameters in the design space to only the most significant ones. Then, the architect only needs to focus on points in the design space that are touched by these parameters.

Table 6 shows the results of a PB design with foldover ($X = 44$) for a superscalar processor with the parameter values shown in Tables 3, 4, and 5. This table shows several key results. First, based on the average ranks, there are 10 significant parameters. This conclusion is drawn by examining the large difference between the average rank of the 10th parameter, LSQ size, which has an average rank of 12.6, and the average rank of the 11th parameter, Speculative Branch Update, which has an average rank of 18.2. Furthermore, while the ranks of the top 10 parameters for each benchmark are completely different, two parameters, the number of ROB Entries and L2 Cache Latency are significant across all of the benchmarks since those two parameters invariably have one of the lowest ranks for every benchmark. Stating it differently, this means that the number of ROB Entries and the L2 Cache Latency are the two biggest performance bottlenecks in the processor across all of the benchmarks tested in this paper. Therefore, of all the user-configurable simulator parameters, the architect needs to be especially careful when choosing parameter values for the number of ROB entries and the L2 Cache Latency.

Although the number of ROB entries and the L2 Cache Latency are the two parameters that have the two lowest average ranks, the L1 I-Cache Size is the most significant performance bottleneck (i.e., has a rank of 1) in six of the 13 benchmarks. By contrast, the number of ROB entries is the most significant performance bottleneck for only four of the 13 benchmarks and the L2 Cache is never the most significant performance bottleneck. However, the L1 I-Cache Size is not one of the five most significant performance

bottlenecks because its rank is 12 or greater for five benchmarks. As a result, when averaging the ranks, the average rank of the L1 I-Cache Size places it as the sixth most significant performance bottleneck. (To determine if averaging the ranks undervalues the significance of the L1 I-Cache Size as a performance bottleneck, we averaged the results in three other ways: computing the weighted average of the cycles, a nonweighted average of the cycles, and an average of the IPCs. The results show that, for all three averaging approaches, the L1 I-Cache Size is never the most significant performance bottleneck, but rather the fifth, fifth, and second most significant performance bottleneck, respectively.) On the other hand, with the exception of two benchmarks, the number of ROB entries and the L2 Cache Latency are always one of the four most significant performance bottlenecks for every benchmark. These results clearly show that the average rank may not reflect the significance of a parameter for groups of individual benchmarks (e.g., L1 I-Cache Size) and is best used only to gain a big-picture view of the results.

Second, the effect that each benchmark has on the processor can be clearly seen. The "effect" that a benchmark has on the processor can be defined as the performance bottlenecks that are present in the processor when running that program. For example, for a compute intensive benchmark, the number of functional units will probably be a performance bottleneck for that processor. On the other hand, for a memory intensive benchmark, the sizes of the L1 D-Cache and the L2 Cache may be the performance bottlenecks.

In this case, for *mesa*, since the ranks for the L1 I-Cache size, associativity, and block size are lower than or similar to the ranks for the L1 D-Cache size, associativity, and block size, respectively, the performance of the instruction cache is more of a limiting factor than the performance of the data cache. The miss rates for the L1 I-Cache and the L1 D-Cache validate this result. When using a 32-byte cache block, the miss rates of the L1 I-Cache are similar to or higher than the miss rates of the L1 D-Cache. Therefore, it is not surprising to see that the L1 I-Cache parameters are generally more significant.

Third, several parameters have surprisingly low ranks in some benchmarks. For example, the FP square root latency in *art* has a rank of five. Since *art* does not have a significant number of FP square root instructions, its rank does not appear to be consistent with its intuitive significance. However, what the rank does not show is that the magnitude of the effect for this parameter is much smaller than magnitudes of the effects for the four most significant parameters. In other words, while ranking the parameters for each benchmark provides a basis for comparison across benchmarks, it cannot be used as the sole arbiter in concluding the significance of a parameter's impact since the rank is not proportional to the magnitude of the effect. For several benchmarks, the L1 I-Cache latency has a very high rank. One reason that the L1 I-Cache latency may be relatively insignificant is because its high and low values span a range of values that may potentially be too small.

Finally, Table 6 shows that the L1 D-Cache parameters (size, associativity, block size, and latency) are not as significant as one would expect. The lowest ranks for the L1 D-Cache size, associativity, block size, and latency are seven (*twolf*), 13 (*gzip*), nine (*art*), and five (*vortex*), respectively. Given the amount of effort that the computer architecture community has put into improving memory performance, one would expect that the L1 D-Cache parameters would have much lower ranks. Therefore, the key question is: Why are the L1 D-Cache parameters not more significant?

One reason that the L1 D-Cache parameters are not more significant is that the memory hierarchy of `sim-outorder` tends to overestimate the memory performance since it does not model memory contention. In addition, `sim-outorder` has a shorter-than-normal pipeline, does not partition the execution core, does not replay traps, and has fewer pipeline flushes. The net effect of these factors is that the average IPC "error" of SimpleScalar for eight selected SPEC 2000 benchmarks is 36.7 percent [4]. Another reason is that the reduced input sets [9] that were used in this paper do not realistically stress the memory hierarchy. Given the unrealistic memory behavior in SimpleScalar and the smaller-than-expected memory footprint, it is not too surprising to see that the L1 D-Cache parameters are not as significant as expected.

4.2 Benchmark Selection

Just as a poorly chosen set of parameter values can drastically affect the performance results, a poorly chosen set of benchmarks may not accurately depict the true performance of the processor or an enhancement. Improperly choosing benchmarks and input sets may affect the results, the conclusions that are drawn, or both. If a computer architect chooses a set of benchmarks that does not accurately reflect the applications that the proposed processor enhancement targets, then the apparent speedup due to that enhancement may be misleading enough to affect the conclusion that the architect forms. Although the results of those simulations are not wrong, they could still be misleading.

However, how does the architect know if two benchmarks are similar or dissimilar? One option is to rely on existing classifications, such as integer versus floating-point, computationally-bound versus memory-bound, or by application type. The problem with this approach is that two benchmarks that are classified differently may have the same characteristics, such as having the same performance bottlenecks in the processor. On the other hand, two benchmarks that are classified to be in the same group may have very dissimilar characteristics. Therefore, simply relying on existing classifications without verifying the similarity of benchmarks within and across classification groups may still result in a poor choice of benchmarks.

The solution proposed in this paper approaches this problem from a different direction. Instead of classifying benchmarks based on their intrinsic characteristics, benchmarks are classified based on what effect they have on the processor. Different benchmarks have different sets of performance bottlenecks. Therefore, two benchmarks that have a similar effect on the processor have most of the same performance bottlenecks and, consequently, should be grouped together. Since the results of the PB design show which parameters are the most important (or, in other

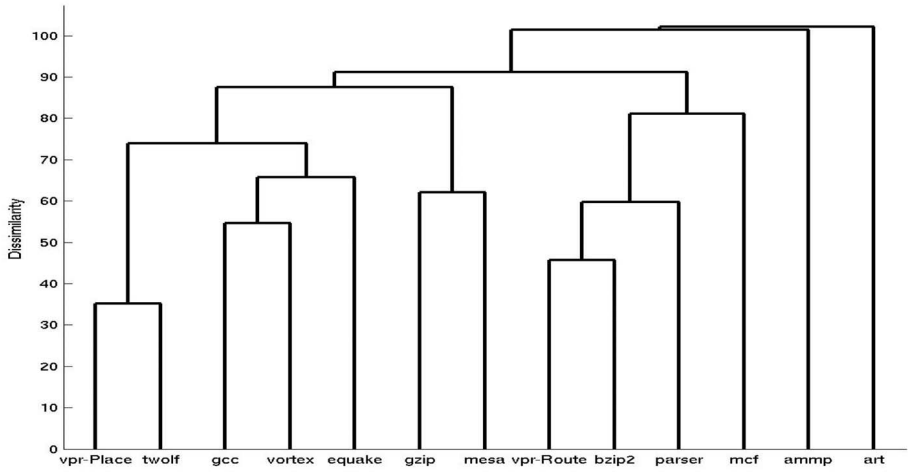


Fig. 1. Cluster analysis results (i.e., dendrogram) for the large MinneSPEC input set.

words, are the biggest performance bottlenecks), comparing the PB design results of two benchmarks indicates how similar the two benchmarks are, in terms of their performance bottlenecks. Benchmarks that are similar are put into the same group. After grouping all the benchmarks into different groups, selecting the final set of the benchmarks is easy since the architect only needs to select either one benchmark from each group or all benchmarks from a single group.

Starting with the results of the PB design, the first step in determining whether two benchmarks have similar effects on the processor is to calculate the Euclidean distance between all possible pair-wise combinations of benchmarks. Since the PB design results for each benchmark is simply a vector of ranks, where each value in the vector corresponds to the rank for that parameter, the formula for computing the Euclidean distance is simply:

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

In this formula, n is the number of parameters while $X = [x_1, x_2, \ldots, x_{n-1}, x_n]$ and $Y = [y_1, y_2, \ldots, y_{n-1}, y_n]$ are the vector of ranks that represent benchmarks X and Y, respectively.

For example, the Euclidean distance between *gzip* and *vpr-Place*, using the ranks from Table 6, is as follows:

$$\text{Distance} = [(1-4)^2 + (4-2)^2 + \ldots + (28-33)^2 + (19-37)^2]^{1/2} = [8,058]^{1/2} = 89.8.$$

In the second step, the benchmarks were clustered together based on their Euclidean distances. And, in the third step, the final clustering tree is plotted. Fig. 1 shows the output of the cluster analysis for the ranks given in Table 6.

In Fig. 1, the benchmarks are arranged along the x-axis while the y-axis represents the level of dissimilarity—the level of dissimilarity is simply the Euclidean distance—between any two benchmarks (or group of benchmarks). Whenever two benchmarks are connected by a horizontal line, that means at that level of dissimilarity and higher, those two benchmarks are considered to be similar. For example, since *vpr-Place* and *twolf* are connected together at a dissimilarity of 35.19, for dissimilarities (or Euclidean distances) less than 35.19, those two benchmarks are categorized into separate groups. However, when the level of dissimilarity exceeds 35.19, they are categorized into the same group. All benchmarks in the same group are considered to be similar.

The first step in selecting a final group of benchmarks to simulate using the dendrogram is to draw a horizontal line at a dissimilarity of 0. Then, the horizontal line should be moved up until the number of vertical lines that it intersects matches the maximum number of benchmarks that can be simulated. The number of intersecting vertical lines represents the number of groups that the benchmarks have been classified into. At that level of dissimilarity, all benchmarks within the same group are considered to be similar, while any benchmark in another group is considered to be dissimilar. The final step in the benchmark selection process is to select one benchmark from each group to form the final set of benchmarks.

The middle column of Table 7 shows the benchmarks in each of the eight groups, while the rightmost column shows the benchmark that could be selected from each group to form the final set. The final set of benchmarks consists of

TABLE 7
Example of Benchmark Selection, Choosing Eight Benchmarks from 13

Group	Benchmarks	Final Set
I	*gzip*, *mesa*	*gzip*
II	*vpr-Place*, *twolf*	*vpr-Place*
III	*vpr-Route*, *parser*, *bzip2*	*vpr-Route*
IV	*gcc*, *vortex*	*gcc*
V	*art*	*art*
VI	*mcf*	*mcf*
VII	*equake*	*equake*
VIII	*ammp*	*ammp*

five integer benchmarks (*gzip*, *vpr-Place*, *vpr-Route*, *gcc*, and *mcf*) and three floating-point benchmarks (*art*, *equake*, and *ammp*). In addition, two of the benchmarks (*art* and *mcf*) have very high cache miss rates (over 20 percent for a 32 KB, two-way associative cache) while the other six have comparatively low miss rates (less than 5 percent for a 32 KB, 2-way cache). Therefore, the final set of benchmarks consists of benchmarks that would come from different groups when categorizing benchmarks using existing methods (integer versus floating-point, computation-bound versus memory-bound, etc.).

Finally, it is important to note that it may be useful to consider other factors when selecting a benchmark from each group. In this example, one reason to choose *gzip* instead of *mesa* from Group I is because *gzip* has a much lower instruction count although those two benchmarks are statistically similar. Similarly, one reason to choose *vpr-Route* over *parser* and *bzip2* from Group III is to match the choice of *vpr-Place* from Group II.

It is important to note that the classification in Table 7 represents only one possible outcome of classifying these benchmarks. It is also important to realize that key metrics, such as IPC and miss rates, could be different within a group. However, since the purpose of this section was to introduce an alternative method of classifying benchmarks (based on their effect on the processor), it is left to the user to group the benchmarks and to decide which benchmarks to select based on this method of classification and, potentially, other metrics. Finally, it is important to realize that the classification of the benchmarks in Fig. 1 and Table 7 are predicated on the efficacy of the mapping of PB magnitudes to ranks. For this paper, we used ranks only because it offered a simple and clean way of normalizing the results of all benchmarks to a common numerical basis. However, due to possible mapping errors for some benchmarks, the classification results may change. Nevertheless, ranks are an appropriate choice for this paper since it allows us to focus on this specific approach to benchmark classification, namely, classification by performance bottlenecks.

4.3 Analysis of Processor Enhancements

In many computer architecture papers, analyzing the effect of a processor enhancement involves examining only individual metrics (e.g., speedup, miss rate, etc.). While these metrics may provide some insight into the effect of the enhancement on key hardware structures, identifying all of the important metrics and trying to piece them back together to form the big picture as to how the enhancement actually affects the processor is extremely difficult, if not impossible. Additionally, while these approaches give the architect a high-level picture of the enhancement's effect, it shows only the net effect.

For example, suppose that a processor enhancement yields a speedup of 25 percent. Also, suppose that two parameters (A and B) are the primary performance bottlenecks in the processor. One case is that the enhancement relieves both bottlenecks by about the same amount. Therefore, the bottlenecks due to both parameters still exist, albeit to a lesser degree. However, another case is that the enhancement relieves the bottleneck due to parameter A, but exacerbates the bottleneck due to parameter B. While both cases could result in the same speedup, the two cases arrive at that speedup by different ways. Therefore, understanding what effect the enhancement has on the performance bottlenecks is a crucial step in trying to improve the performance of the enhancement. High-level metrics such as speedup only show what the enhancement did, but not how it got there. Since the "how" affects the "what," it is important to determine the effect that an enhancement has to a greater depth than just with high-level metrics.

Therefore, as a complement to the high-level metrics, this paper proposes using the PB design to quantify the effect of an enhancement. The results of a PB design can be used to measure the significance of all parameters with and without the enhancement. Since the significance of a parameter is an indication of how much of a performance bottleneck that parameter is, a change in the significance of a parameter means that that parameter is more or less of a performance bottleneck with that enhancement.

To determine what effect an enhancement has on each parameter, we compute the difference in the average ranks of each parameter. Consequently, any parameter that experiences a large change in its average rank after an enhancement is applied has become more of a bottleneck ($\text{Before} - \text{After} > 0$) or less of a bottleneck ($\text{Before} - \text{After} < 0$).

To illustrate how a PB design can be used in this way, this technique was used to analyze the effects that instruction precomputation and simplifying and eliminating trivial computations had on the processor. Table 8 presents the PB design results for instruction precomputation, while Table 9 does the same for simplifying and eliminating trivial computations.

4.3.1 Instruction Precomputation

Instruction precomputation [21] is similar to value reuse [17], with the key difference that instruction precomputation uses profiling to statically identify the highest frequency redundant computations instead of identifying them at runtime. Also, in instruction precomputation, the redundant computations are loaded into the precomputation table (PT) before the program begins and are never updated. By contrast, value reuse continually updates the value reuse table.

Table 8 shows the results for instruction precomputation when using two different input sets for profiling and for execution and while using a 128-entry PT. Table 8 represents the "after" case, while Table 6 represents the "before" case, that is, the unenhanced processor.

Comparing these two tables yields two conclusions about the effect that instruction precomputation has 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. While instruction precomputation changes the relative ordering of the significant parameters, with respect to each other, it does not change which parameters have the greatest significance.

Second, of the significant parameters, the parameter that has the biggest change in its overall effect (defined as the biggest change in its average rank) is the number of integer ALUs; instruction precomputation changes its average rank

TABLE 8
Plackett and Burman Design Results for All Processor Parameters when Using Instruction Precomputation, Ranked by Significance and Sorted by the Average Rank

Parameter	gzip	vpr-Place	vpr-Route	gcc	mesa	art	mcf	equake	ammp	parser	vortex	bzip2	twolf	Ave
ROB Entries	1	4	1	4	3	2	2	3	6	1	4	1	4	**2.8**
L2 Cache Latency	4	2	4	2	2	4	4	2	13	3	2	8	2	**4.0**
Branch Predictor	2	5	3	5	5	28	11	8	4	4	16	7	5	**7.9**
L1 D-Cache Latency	7	6	5	7	11	8	14	5	40	7	5	4	6	**9.6**
L1 I-Cache Size	5	1	12	1	1	12	38	1	36	8	1	15	1	**10.2**
Int ALUs	6	8	8	9	8	29	9	13	20	6	9	3	9	**10.5**
L2 Cache Size	9	35	2	6	22	1	1	6	2	2	6	2	43	**10.5**
L1 I-Cache Block Size	15	3	20	3	14	10	32	4	10	11	3	20	3	**11.4**
Memory Latency First	35	25	6	8	18	3	3	7	1	5	7	6	27	**11.6**
LSQ Entries	13	14	9	10	15	40	10	9	17	9	8	5	10	**13.0**
D-TLB Size	21	28	11	24	25	13	12	10	25	14	25	10	24	**18.6**
Speculative Branch Update	8	20	25	29	7	16	39	11	8	20	21	22	19	**18.8**
L1 I-Cache Associativity	3	41	15	28	6	34	23	28	16	17	11	9	21	**19.4**
L1 D-Cache Size	18	7	10	12	42	19	8	35	32	21	13	32	7	**19.7**
FP Multiply Latency	31	12	22	11	19	24	15	22	24	28	14	24	18	**20.3**
Memory Bandwidth	33	36	13	14	43	6	6	31	3	12	20	11	38	**20.5**
BTB Entries	10	23	19	20	9	41	31	20	22	19	19	16	34	**21.8**
Int ALU Latencies	16	15	18	13	40	22	33	14	31	16	41	12	16	**22.1**
L1 D-Cache Block Size	17	30	34	22	16	9	24	19	26	13	33	25	26	**22.6**
Int Divide Latency	30	10	26	17	24	33	40	33	19	10	10	41	8	**23.2**
L2 Cache Associativity	23	19	14	19	33	27	5	39	37	18	42	21	12	**23.8**
Int Mult/Div	14	21	30	31	12	23	27	23	33	37	18	27	15	**23.9**
I-TLB Latency	32	17	24	18	34	30	30	16	21	33	12	29	17	**24.1**
Instruction Fetch Queue Entries	43	13	27	30	23	20	19	37	9	25	23	34	14	**24.4**
Branch Misprediction Penalty	11	24	41	21	4	43	20	32	11	22	39	35	23	**25.1**
FP Divide Latency	20	9	36	16	28	21	37	15	43	38	17	38	11	**25.3**
FP ALUs	34	11	31	15	38	17	41	24	27	36	15	43	13	**26.5**
I-TLB Page Size	42	38	7	38	39	39	7	17	12	26	28	14	39	**26.6**
L1 D-Cache Associativity	12	39	17	35	17	42	34	34	14	15	36	17	42	**27.2**
L2 Cache Block Size	25	43	16	37	31	7	35	27	7	35	38	13	40	**27.2**
I-TLB Associativity	26	27	38	25	20	31	42	12	29	30	22	33	22	**27.5**
BTB Associativity	22	18	35	32	10	32	17	30	34	43	27	36	25	**27.8**
D-TLB Associativity	40	32	23	26	27	35	25	26	18	32	26	28	35	**28.7**
Memory Ports	39	31	39	23	26	15	16	40	5	42	30	40	29	**28.8**
FP ALU Latencies	37	16	37	41	37	11	21	29	23	27	29	42	28	**29.1**
I-TLB Size	36	34	28	34	21	37	18	18	30	34	34	30	32	**29.7**
Dummy Factor #2	28	42	21	39	32	14	13	36	42	29	43	18	30	**29.8**
Int Multiply Latency	29	40	42	36	13	26	29	21	15	41	35	31	41	**30.7**
FP Mult/Div	41	22	43	40	41	18	28	38	28	31	31	19	20	**30.8**
FP Square Root Latency	38	29	40	33	35	5	26	43	41	24	24	39	37	**31.8**
Return Address Stack Entries	27	33	33	27	36	25	36	25	39	40	32	37	31	**32.4**
L1 I-Cache Latency	24	26	32	42	29	38	22	41	38	39	37	26	33	**32.8**
Dummy Factor #1	19	37	29	43	30	36	43	42	35	23	40	23	36	**33.5**

from 9.1 in the base processor to 10.5 with instruction precomputation. This result is intuitively reasonable since most of the instructions that instruction precomputation eliminates would have executed on the integer ALUs. Therefore, using instruction precomputation decreases the impact of the number of integer ALUs on the processor's performance.

Although these results show that instruction precomputation improves the processor's performance by reducing functional unit contention, it also improves the processor's performance by decreasing the execution latency of redundant computations. However, since the base SimpleScalar processor has a short pipeline, this latter effect appears to be relatively small.

4.3.2 The Simplification and Elimination of Trivial Computations

A trivial computation is one where the output is zero, one, 0xffffffff, or a shifted version of one of the inputs. To exploit these trivial computations, hardware is added to the processor to first check the opcode and input operands to determine whether each instruction is trivial and, if so, whether it can be simplified or eliminated. When the trivial computation can be simplified, the instruction is converted to another type of instruction that produces the same result, but with a lower execution latency. When the trivial computation can be eliminated, the trivial computation hardware "computes" its result and removes the instruction from the pipeline.

Table 9 shows the results for simplifying and eliminating trivial computations [22] and it represents the "after" case, while Table 6 represents the "before" case.

The results in Table 9 show that simplifying and eliminating trivial computations has a similar effect on all processor parameters. That is, the performance bottlenecks in the base processor do not get substantially better or worse when hardware to exploit trivial computations is added to the processor. There are two reasons to support this conclusion. First, the order of the 10 most significant parameters is the same as the base processor. Since their average ranks are nearly identical, with respect to the base case, this enhancement has a very similar effect on the most significant processor parameters.

Second, there is relatively little difference between the average ranks for the other parameters. The maximum difference between the average ranks for a parameter with and without adding the trivial computation exploitation hardware is 1.3. Although this difference rivals the change in the average rank for the number of integer ALUs when instruction precomputation is added to the base processor, this difference is less meaningful because it is a smaller

TABLE 9
Plackett and Burman Design Results for All Processor Parameters when Simplifying and Eliminating Trivial Computations, Ranked by Significance and Sorted by the Average Rank

Parameter	gzip	vpr-Place	vpr-Route	gcc	mesa	art	mcf	equake	ammp	parser	vortex	bzip2	twolf	Ave
ROB Entries	1	4	1	4	3	2	2	3	6	1	4	1	4	**2.8**
L2 Cache Latency	4	2	4	2	2	4	4	2	13	3	2	8	2	**4.0**
Branch Predictor	2	5	3	5	5	27	11	8	4	4	16	7	5	**7.8**
Int ALUs	6	8	7	8	6	29	9	6	19	7	9	2	9	**9.6**
L1 D-Cache Latency	7	6	5	7	12	8	14	5	40	6	5	6	6	**9.8**
L1 I-Cache Size	5	1	12	1	1	11	38	1	36	8	1	16	1	**10.2**
L2 Cache Size	9	40	2	6	21	1	1	7	2	2	6	3	43	**11.0**
L1 I-Cache Block Size	16	3	20	3	15	10	32	4	10	11	3	21	3	**11.6**
Memory Latency First	36	28	6	9	20	3	3	9	1	5	7	5	27	**12.2**
LSQ Entries	12	16	9	10	14	43	10	10	17	10	8	4	10	**13.3**
Speculative Branch Update	8	18	26	29	7	16	39	13	8	20	21	20	18	**18.7**
L1 D-Cache Size	18	7	11	12	43	18	8	25	31	21	12	33	7	**18.9**
D-TLB Size	20	27	10	23	30	13	12	12	25	14	27	11	25	**19.2**
FP Multiply Latency	31	11	22	11	19	24	15	17	24	26	14	24	17	**19.6**
Memory Bandwidth	37	41	13	14	39	6	6	22	3	12	20	12	39	**20.3**
L1 I-Cache Associativity	3	38	16	28	8	34	23	43	16	16	13	9	21	**20.6**
BTB Entries	10	22	18	20	9	42	30	15	22	18	19	17	34	**21.2**
Int Divide Latency	29	10	24	17	25	33	40	21	20	9	10	43	8	**22.2**
L1 D-Cache Block Size	17	29	31	22	16	9	24	14	27	13	34	28	26	**22.3**
Int ALU Latencies	15	14	19	13	42	21	33	18	30	17	43	10	16	**22.4**
L2 Cache Associativity	23	15	14	19	33	28	5	38	37	19	39	22	13	**23.5**
Branch Misprediction Penalty	11	23	43	21	4	40	20	19	11	22	42	36	23	**24.2**
Int Mult/Div	14	24	30	31	11	23	27	34	33	37	17	26	14	**24.7**
FP Divide Latency	21	9	34	16	26	22	37	11	43	38	18	38	11	**24.9**
Instruction Fetch Queue Entries	43	13	27	30	27	20	18	40	9	25	23	35	15	**25.0**
I-TLB Latency	32	20	23	18	37	30	31	24	21	33	15	27	19	**25.4**
L1 D-Cache Associativity	13	39	15	34	18	41	34	23	14	15	37	15	41	**26.1**
FP ALUs	34	12	33	15	31	17	41	32	26	36	11	41	12	**26.2**
BTB Associativity	22	19	37	32	10	32	17	20	34	43	26	34	24	**26.9**
I-TLB Page Size	42	35	8	38	36	39	7	26	12	27	29	14	38	**27.0**
I-TLB Associativity	25	25	36	25	17	31	42	16	29	29	22	32	22	**27.0**
L2 Cache Block Size	26	42	17	37	32	7	35	41	7	35	36	13	40	**28.3**
Memory Ports	40	30	41	24	28	14	16	29	5	42	31	40	29	**28.4**
D-TLB Associativity	39	33	25	26	22	35	25	37	18	32	25	29	36	**29.4**
FP ALU Latencies	33	21	38	41	38	12	21	28	23	30	30	42	28	**29.6**
Dummy Factor #2	28	36	21	39	34	15	13	42	41	28	40	18	32	**29.8**
FP Mult/Div	41	17	42	40	40	19	28	39	28	31	32	19	20	**30.5**
I-TLB Size	35	31	28	35	23	36	19	27	32	34	33	30	33	**30.5**
FP Square Root Latency	38	32	39	33	35	5	26	33	42	24	24	39	37	**31.3**
Int Multiply Latency	30	43	40	36	13	26	29	31	15	41	38	31	42	**31.9**
Dummy Factor #1	19	37	29	43	24	37	43	30	35	23	41	23	35	**32.2**
L1 I-Cache Latency	24	26	35	42	29	38	22	36	38	39	35	25	31	**32.3**
Return Address Stack Entries	27	34	32	27	41	25	36	35	39	40	28	37	30	**33.2**

percentage of the average rank for that parameter. In other words, since those parameters are insignificant to begin with, large changes in their average rank do not imply that the enhancement has a large effect on that parameter.

4.3.3 Limitations of the Plackett and Burman Design for Enhancement Analysis

While this approach has many strengths—including comparing how all performance bottlenecks migrate due to the enhancement—it has at least two important limitations. First, if the enhancement directly changes the value of the input parameters, then this approach cannot be used since the high and low values for the parameter have effectively changed, which then makes it impossible to accurately calculate the PB magnitude for that parameter.

Second, this analysis cannot determine the performance bottlenecks—and, hence, their subsequent migration—that are not input parameters. An example of a noninput parameter might be the processor's ISA. While a poor ISA could lead to significant performance degradation, it is very difficult to make the ISA into an input parameter since it could require changes to the architecture as well, which then introduces additional parameters. In any case, since the PB design only measures that effect that each parameter has on the variability in the output value, the only performance bottlenecks are due to input parameters, which means that the migration of noninput parameter performance bottlenecks cannot be examined by this analysis.

Finally, as described in the previous sections, using ranks and averaging ranks trades simplicity for the potential of quantization error. Nevertheless, using and averaging ranks is acceptable in this paper since it allows us to focus on this specific performance analysis technique.

4.3.4 Summary

In conclusion, this method of analyzing simulation results has a few advantages over commonly used approaches that only look at a single metric. First, the exact effect that an enhancement has on the parameters can be determined. This information is especially useful in finding parameters that would seem to be unaffected by an enhancement, but are, in actuality, significantly affected. This information also can point the user to areas in the processor that may require a more detailed analysis. Second, the user can determine the most significant parameters of the enhancement and how its ranks compare to the ranks of the parameters. This comparison allows the user to make design decisions as to how to maximize the performance while minimizing the enhancement's cost. Finally, using this method 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.

5 Related Work

The related work in this section is divided into two categories: simulator validation, and benchmark and input set characterization.

5.1 Simulator Validation

Black and Shen [2] iteratively improved the accuracy of their performance model by comparing the cycle count of their simulator, which targeted a specific architecture, against the cycle count of 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.

Desikan et al. [4] measured the amount of error, as compared to the Alpha 21264 processor, that was present in an Alpha version of the SimpleScalar simulator. Their results showed that the simulators that model a generic machine (i.e., nonspecific architecture) generally report higher IPCs than simulators that are validated against a real machine. This result is not particularly surprising since it is likely that unvalidated, generic-architecture simulators will tend to underestimate the complexity of the implementing certain microarchitectural features.

Gibson et al. [6] described the types of errors that were present in the FLASH simulator when compared to the custom-built FLASH multiprocessor system. Their results showed that most simulators can accurately predict the architectural trends if all of the important components have been accurately modeled and that the margin of error (the percentage difference in the execution time) of some simulators was more than 30 percent, which is higher than the speedups that are often reported for specific architectural enhancements.

5.2 Benchmark and Input Set Characterization

To address the problem of potentially selecting a poor set of benchmarks, Eeckhout et al. [5] used statistical data analysis techniques to determine the statistical similarity of benchmark and input set pairs. To quantify the similarity, they used metrics such as the instruction mix, the branch prediction accuracy, the data and instruction cache miss rates, the number of instructions in a basic block, and the maximum amount of parallelism inherent to the benchmark. After characterizing each benchmark with the aforementioned metrics, they used statistical approaches such as principal component analysis and cluster analysis to actually cluster the benchmarks and input set pairs together.

6 Conclusion

Computer architects heavily rely on simulators when designing processor architectures or when evaluating the performance of processor enhancements. 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 sets of benchmark programs. Furthermore, without a formalized methodology, computer architects may not glean as much information as possible from their simulation results. By adding statistical rigor to their methodology, computer architects can have more confidence in their 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 seeks to improve the performance analysis phase of the simulation process. The first method focuses on how the processor parameter values are chosen, but can also be used to efficiently and accurately reduce the design space. In particular, 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.

The second method focuses on benchmark selection. Our method groups benchmarks together if they have a similar effect on the processor. Two benchmarks have similar effects on the processor if their 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 performance analysis in the postsimulation 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.

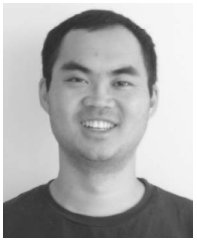
Acknowledgments

The authors would like to thank Chris Hescott, Baris Kazar, Keith Osowski, Mike Tobin, and Keqiang Wu for their helpful comments on previous drafts of this work. A preliminary version of this work was presented at the Ninth Annual International Symposium on High-Performance Computer Architecture [24]. This work was supported in part by US National Science Foundation grants CCR-9900605 and EIA-9971666, IBM Corporation, and the Minnesota Supercomputing Institute.

References

[1] P. Bannon and Y. Saito, "The Alpha 21164PC Microprocessor," *Proc. IEEE Int'l Computer Conf.*, 1997.
[2] B. Black and J. Shen, "Calibration of Microprocessor Performance Models," *Computer*, vol. 31, no. 5, pp. 59-65, May 1998.
[3] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report #1342, Computer Sciences Dept., Univ. of Wisconsin-Madison, 1997.
[4] R. Desikan, D. Burger, and S. Keckler, "Measuring Experimental Error in Microprocessor Simulation," *Proc. Int'l Symp. Computer Architecture*, 2001.

[5] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Workload Design: Selecting Representative Program-Input Pairs," *Proc. Int'l Conf. Parallel Architectures and Compilations Techniques,* 2002.
[6] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2000.
[7] T. Horel and G. Lauterbach, "UltraSPARC-III: Designing Third-Generation 64-Bit Performance," *IEEE Micro,* vol. 19, no. 3, pp. 73-85, May-June 1999.
[8] R. Kessler, E. McLellan, and D. Webb, "The Alpha 21264 Microprocessor Architecture," *Proc. Int'l Conf. Computer Design,* 1998.
[9] A. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters,* vol. 1, 2002.
[10] A. Kumar, "The HP PA-8000 RISC CPU," *IEEE Micro,* vol. 17, no. 2, pp. 27-32, Mar.-Apr. 1997.
[11] D. Lilja, *Measuring Computer Performance.* Cambridge Univ. Press, 2000.
[12] D. Montgomery, *Design and Analysis of Experiments,* third ed. Wiley, 1991.
[13] 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, pp. 14-24, Mar.-Apr. 1998.
[14] R. Plackett and J. Burman, "The Design of Optimum Multifactorial Experiments," *Biometrika,* vol. 33, no. 4, pp. 305-325, June 1946.
[15] J. Silc, B. Robic, and T. Ungerer, *Processor Architecture: From Dataflow to Superscalar and Beyond.* Springer-Verlag, 1999.
[16] D. Sima, T. Fountain, and P. Kacsuk, *Advanced Computer Architectures, A Design Space Approach.* Addison Wesley Longman, 1997.
[17] A. Sodani and G. Sohi, "Dynamic Instruction Reuse," *Proc. Int'l Symp. Computer Architecture,* 1997.
[18] S. Song, M. Denman, and J. Chang, "The PowerPC 604 RISC Microprocessor," *IEEE Micro,* vol. 14, no. 5, pp. 8-17, Oct. 1994.
[19] M. Tremblay and J.M. O'Connor, "UltraSparc I: A Four-Issue Processor Supporting Multimedia," *IEEE Micro,* vol. 16, no. 2, pp. 42-50, Apr. 1996.
[20] K. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro,* vol. 16, no. 2, pp. 28-40, Apr. 1996.
[21] J. Yi, R. Sendag, and D. Lilja, "Increasing Instruction-Level Parallelism with Instruction Precomputation," *Proc. Int'l Euro-Par Conf. Parallel Processing,* 2002.
[22] J. Yi and D. Lilja, "Improving Processor Performance by Simplifying and Bypassing Trivial Computations," *Proc. Int'l Conf. Computer Design,* 2002.
[23] J. Yi and D. Lilja, "Effects of Processor Parameter Selection on Simulation Results," Minnesota Supercomputer Inst. Report 2002/146, 2002.
[24] J. Yi, D. Lilja, and D. Hawkins, "A Statistically Rigorous Approach for Improving Simulation Methodology," *Proc. Int'l Conf. High-Performance Computer Architecture,* Feb. 2003.



Joshua J. Yi received the PhD, MS, and BS degrees, all in electrical engineering, from the University of Minnesota in Minneapolis. He is currently a performance analyst at Freescale Semiconductor, Inc. in Austin, Texas. His research interests include high-performance computer architecture, simulation, benchmarking, low power design, and reliable computing. He is a member of the IEEE and the IEEE Computer Society.

David J. Lilja received the PhD and MS degrees, both in electrical engineering, from the University of Illinois at Urbana-Champaign and the BS degree in computer engineering from Iowa State University. He is currently a professor of electrical and computer engineering and a fellow of the Minnesota Supercomputing Institute at the University of Minnesota in Minneapolis. He also serves as a member of the graduate faculties in computer science and scientific computation. He has been a visiting senior engineer with the Hardware Performance Analysis Group at IBM in Rochester, Minnesota, and a visiting professor at the University of Western Australia in Perth, supported by a Fulbright award. Previously, he worked as a research assistant at the Center for Supercomputing Research and Development at the University of Illinois and as a development engineer at Tandem Computers Inc. (now a division of Hewlett-Packard) in Cupertino, California. He has chaired and served on the program committees of numerous conferences, was a distinguished visitor of the IEEE Computer Society, is a member of the ACM and a senior member of the IEEE and the IEEE Computer Society, and is a registered professional engineer in electrical engineering in Minnesota and California. His primary research interests are in high-performance computer architecture, parallel computing, hardware-software interactions, nano-computing, and performance analysis.

Douglas M. Hawkins studied at Witwatersrand University in Johannesburg, South Africa. After a spell as chairman of statistics there and another running his own statistical consultancy companies, he joined the School of Statistics at the University of Minnesota. His research interests are varied, including statistical tools for quality improvement, data diagnostics, data mining, and multivariate methods.