A Statistically Rigorous Approach for Improving Simulation Methodology

Joshua J. Yi, David J. Lilja
Department of Electrical and Computer Engineering
University of Minnesota – Twin Cities
{jjyi, lilja}@ece.umn.edu

Douglas M. Hawkins
School of Statistics
University of Minnesota – Twin Cities
doug@stat.umn.edu

Abstract

Due to cost, time, and flexibility constraints, simulators are often used to explore the design space when developing new processor architectures, as well as when evaluating the performance of new processor enhancements. However, despite this dependence on simulators, statistically rigorous simulation methodologies are not typically 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 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. Additionally, they allow the architect to quickly determine the expected performance improvement of a new processor enhancement.

Despite this dependence on simulators, 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? What values should be used for those parameters? Do any of the constant parameters interact with the variable ones? What is the magnitude of the effects for those interactions? How much impact do the specific values for the constant parameters have? 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, the architect must answer these questions before starting the simulations. Due to the sheer computational cost, however, it is virtually impossible to simulate all possible combinations of parameters. This situation illustrates the need for a statistically-based methodology to answer these types of questions.

While using such a methodology 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, 5, 8, 10].
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.

While the first and third advantages are self-explanatory, it is not obvious from the second advantage how a statistically-based methodology could improve the quality of the analysis. However, 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, architects use high-level metrics, such as speedup, to understand the "big-picture" effects. However, analyzing the 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 (e.g. power consumption, etc.).

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, for classifying benchmarks based on how they affect the processor, and for analyzing the effect of a processor enhancement.

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 the simulation methodology, decrease the total number of simulations, quickly determine the processor's bottlenecks, 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 [3].

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. Plackett and Burman Designs

In this paper, we used a Plackett and Burman (PB) design [23] to determine the effect that a parameter has on the processor's performance. 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 [17]. However, these two techniques did not produce the desired level of information (one-at-a-time) or required 2^N simulations (ANOVA). A detailed comparison of these three techniques can be found in [33].

Saturated designs, such as the PB design, are recipes that vary all N parameters simultaneously over N+1 simulations. They provide the logically minimal number of simulations required to estimate the effect of each of the N parameters. An improvement on the basic PB design is the "foldover" PB design [19]. This requires approximately 2N simulations. With this experimental design, the user can determine the effects of all of the main parameters and selected interactions. Since PB designs exist only in sizes that are multiples of 4, the base PB design requires X simulations, where X is the next multiple of four that is greater than N (including if N is a multiple of 4), while the foldover PB design requires 2X simulations.

However, the downside of the PB design is that it cannot quantify the effects of all of the interactions. Therefore, it is possible that unknown, but significant, interactions may alter the apparent effect of any of the parameters. Fortunately for computer architects, this situation probably does not occur for processor parameters. The results in [32] show that if an interaction between parameters was significant, it was significant only because each of its constituent parameters was individually significant. Additionally, the effects of the most significant interactions were relatively small compared to the effects of the most significant parameters. As a result, using a PB design with foldover to analyze the effects of the processor parameters does not compromise the results.

The parameters' configuration for each simulation run is given by the PB design matrix which, for most values of X, is simple to construct. The rows of the design matrix correspond to different configurations while the columns correspond to the parameters' values in each configuration. 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 these values of X, the first row of the design matrix is given in [23]. The next $X - 2$ rows are formed by performing a circular right shift on the preceding row. The last line of the design matrix is a row of minus ones. The gray-shaded portion of Table 1 illustrates the construction of the PB design matrix for X=8, a design appropriate for investigating 7 (or fewer) parameters. When using foldover, X additional rows are added to the matrix. The signs in each entry of the additional rows are the opposite of the corresponding entries in the original matrix. Table 1 shows the complete PB design matrix with foldover. Note that rows 10-17 specifically show the additional foldover rows.

A "+1", or high value, for a parameter represents a value that is higher than the range of normal values for that parameter while a "-1", or low value, represents 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. For example, in the case of branch prediction, the high and low values could be perfect and 2-level branch prediction, respectively. It is

also important to note that choosing high and low values that yield too large a range can artificially inflate the parameter's apparent effect; too small a range has the opposite result. Therefore, the user should 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.

Table 1. Plackett and Burman design matrix with foldover (X=8).

A	B	C	D	E	F	G	Exec. Time
+1	+1	+1	-1	+1	-1	-1	9
-1	+1	+1	+1	-1	+1	-1	11
-1	-1	+1	+1	+1	-1	+1	2
+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	7
-1	-1	-1	-1	-1	-1	-1	4
-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	112
191	19	111	-13	79	55	239	

After determining the configurations and performing the simulations, the effect of each parameter is computed by the multiplying the parameter's PB value for a configuration by the result (e.g. execution time) for that configuration and summing the resulting products across all configurations. For example, the effect of parameter A from Table 1 is computed as follows:

$$Eff_A = (1 * 9) + (-1 * 11) + \ldots + (-1 * 6) + (1 * 112) = 191$$

For the example in Table 1, the parameters with the most effect are G, A, and C, in order of their overall impact on performance. Only the magnitude of the effect is important; the sign of the effect is meaningless.

3. Simulator, Benchmarks, and Parameters

In the remainder of the paper, we show how a PB design can be used to select parameter values and benchmark programs for simulations, and how it can provide insights into the impact of a processor enhancement. 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.

The benchmarks that were used in this paper, shown in Table 2, were selected from the SPEC 2000 benchmark suite. We chose these benchmarks because they were the only ones that had MinneSPEC [14] reduced input sets available at the time. All benchmarks were compiled at optimization level O3 using the SimpleScalar version of the gcc compiler and were run to completion.

Table 2. Selected benchmarks from the SPEC 2000 benchmark suite used in this paper.

Benchmark	Type	Dynamic Instr. (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

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 MPred Penalty	10 Cycles	2 Cycles
RAS Entries	4	64
BTB Entries	16	512
BTB Assoc	2-Way	Fully-Assoc
Spec 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

As described in the 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 be actually present in commercial processors nor are they values that should be used in the simulations. Rather, the values were ***deliberately chosen to be slightly higher and lower than the range of "normal" values***. We based our parameter

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 Throughputs	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	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 FP Mult Latency	
FP Div Throughput	Equal to FP Div Latency	
FP Sqrt Throughput	Equal to 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	

values on those found in several commercial processors. Our list of commercial processors included the Alpha 21164 [1, 6] and 21264 [12, 13, 16, 18]; the UltraSparc I [29], II [21], and III [11]; the HP PA-8000 [15]; the PowerPC 604 [28]; and the MIPS R10000 [30]. To fill in the gaps left by these papers, [24, 25] 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 couple of 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 the PB design. The problem occurs when one of those parameters is set to its high value while the parameter it is related to is set to its low value. That 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 8-entry ROB. But since the total number of in-flight instructions cannot exceed the number of reorder buffer entries, the maximum number of filled LSQ entries will never exceed 8. Therefore, to avoid those types of situations, the values 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 four for two reasons. First of all, we fixed the issue width 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 at four to eliminate the guesswork needed to determine the normal range of parameter values for a higher issue width processor. Also, there was a lot of documentation available for several 4-way issue commercial processors. Note that fixing the issue width does not affect the conclusions drawn from these simulations, especially since this is a methodology paper. Fixing the issue width merely removes it as a variable parameter.

Finally, we used sim-outorder instead of the validated Alpha 21264 simulator [5] for three reasons. First, since this is a methodology paper, the specific simulator used does not affect the final conclusions since the simulation results serve only to illustrate certain key points. Second, since the Alpha simulator has many Alpha-architecture specific parameters, we used sim-outorder to avoid producing results that were particular to the 21264. Third, since sim-outorder is a popular simulator, 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 principal phases of the simulation process in computer architecture research are setup, simulation, and data analysis. The first phase occurs after the user determines the initial set of testcases and modifies the simulator and/or compiler. As a result, in the first phase, the user determines the values of the processor parameters and selects the benchmarks that will be simulated. In the third phase, the user analyzes the results that were gathered during the simulation phase. Then, depending on the results, the process may be repeated.

This section focuses on improving the methodology of the first and third phases. To improve the methodology of the first phase, we describe a statistically rigorous method of choosing the processor parameter values. In addition, to improve the benchmark selection process, we describe a method of classifying benchmarks based on grouping together benchmarks that have similar effects on the processor. To improve the methodology of the third phase, we describe a 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 illustrate the efficacy, utility, 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. Processor Parameter Selection

Improperly choosing the value of a single parameter can significantly affect the simulated speedup of a processor enhancement. For instance, simply increasing the reorder buffer size can change the *speedup* of value reuse [27] from approximately 20% to approximately 30%. 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 a virtual impossibility. And 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. Fortunately, this problem can be solved by using a PB design to identify the significant parameters.

Table 6 shows the results of a PB design with foldover (X=44) for the base superscalar processor with the parameter values shown in Tables 3-5. 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 parameter (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 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. This conclusion is 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, two parameters (ROB Entries and L2 Cache Latency) are significant across all of the benchmarks since those two are almost always one of the most important parameters for every benchmark. This means that the Reorder Buffer and the L2 Cache latency are the two biggest bottlenecks in the processor.

Second, the effect that each benchmark has on the processor can be clearly seen. For instance, 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, for *mesa*, we conclude that *mesa* stresses the instruction cache more than the data cache. Table 6 also shows 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.

Finally, several parameters have surprisingly low 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, 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. Therefore, this example shows that while the rank is convenient to use, it cannot be used as the sole arbiter in concluding the significance of a parameter's impact.

After determining the critical parameters, the task of choosing the final parameter values is simplified since only the values for the critical parameters need to be chosen carefully. We recommend performing iterative sets of sensitivity analyses so that the exact interaction between critical parameters can be accounted for when choosing the final values of the critical parameters. To summarize,

Table 6. 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
ROB 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
Branch Predictor	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
Mem 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 Assoc	5	40	15	29	8	34	23	28	16	17	15	9	21	260
FP Mult 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 Latency	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 Div Latency	29	10	26	16	24	32	41	32	20	10	10	43	8	301
Int Mult/Div Units	14	20	29	31	10	23	27	24	33	36	18	26	15	306
L2 Cache Assoc	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
Fetch Queue Entries	43	13	27	30	26	20	18	37	9	25	23	34	14	319
Branch MPred 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 Div 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 Assoc	13	38	17	34	18	41	34	33	14	15	35	15	42	349
I-TLB Assoc	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 Assoc	21	21	36	32	11	33	17	31	34	43	27	35	25	366
D-TLB Assoc	40	32	25	26	22	35	26	26	18	33	26	30	35	374
FP ALU Latency	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 Parameter #2	27	42	21	39	35	14	13	35	41	28	43	18	30	386
FP Mult/Div Units	41	22	43	40	40	19	28	38	27	31	31	19	20	399
Int Mult Latency	30	41	39	36	14	26	29	21	15	41	37	32	41	402
FP Sqrt 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
RAS Entries	28	33	33	27	42	25	36	25	39	39	33	36	32	428
Dummy Parameter #1	19	37	30	43	25	36	43	43	35	23	40	24	36	434

we recommend the following steps when choosing the final simulation 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 critical parameters.
2) Iteratively perform sensitivity analyses for each critical parameter using the ANOVA technique.
3) Choose final values for the critical parameters based on the results of the sensitivity analyses.
4) Choose the final values for the non-critical parameters based on commercial processor values, or some other appropriate source.

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 enhancement. For instance, if the set of benchmarks was extremely memory-intensive, then an optimization to the memory hierarchy (e.g. prefetching) will overestimate the performance of that optimization across the range of benchmarks. However, if the user simply simulates all of the benchmarks from a benchmark suite, then he/she sacrifices a more complete exploration of the design space by simulating redundant benchmarks. Therefore, for accuracy and efficiency reasons, it is important for the user to simulate a set of benchmarks that are distinct, but that are

Table 7. Euclidean distance between benchmark vectors, based on parameter ranks in Table 6.

	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

representative of the range of benchmarks.

However, determining if two benchmarks are similar is difficult because 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 benchmarks be classified by their effect on the processor. Under this method of classification, two benchmarks are similar if they stress the same components of the processor to similar degrees.

To calculate the similarity between two benchmarks, we treat the rank of each parameter as an element of a vector. Therefore, a benchmark's vector represents the ranks of all parameters. To determine how similar two benchmarks were, we computed the Euclidean distance between the two vectors as follows:

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

where n is the number of parameters 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 ranks of the parameters in benchmarks X and Y, respectively. The distance between the two vectors measures how similarly the two benchmarks affect the processor. Obviously, the smaller the distance between the vectors, the greater the similarity. For example, the distance between *gzip* and *vpr-Place*, using the ranks from Table 6, is as follows:

$$Dist = [(1-4)^2 + (4-2)^2 + \ldots + (28-33)^2 + (19-37)^2]^{1/2}$$
$$= [8058]^{1/2} = 89.8$$

Benchmarks are similar if their distance is below some *user-defined* similarity threshold. Therefore, by calculating the similarity between all pairs of benchmarks in a benchmark suite and using the similarity threshold, the user can ***subjectively*** decide which benchmarks are similar. Table 7 shows the result of comparing all pairs of benchmarks using the ranks from Table 6.

In this example, if the user-defined similarity threshold was arbitrarily set to 63.2 (i.e. the square root of 4000), any pair of benchmarks that had a distance less than 63.2 could be considered similar. The bold entries in Table 7 correspond to benchmark pairs whose distances are less than the similarity threshold. Table 8 groups benchmarks that are similar together in the same row.

Table 8. Benchmarks grouped by their effect on the processor for an arbitrary similarity threshold of 63.2.

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

It is important to note that the classification in Table 8 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 very different within a group. However, since the purpose of this sub-section was to introduce an alternative method of classifying benchmarks (based on their effect on the processor), it is left to the user to set the similarity threshold, to group the benchmarks, and to decide which benchmarks to select based on this method of classification and potentially, other metrics.

4.3. Analysis of Processor Enhancements

In many computer architecture papers, analyzing the

Table 9. Plackett and Burman design results for all processor parameters when using Instruction Precomputation; 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
ROB 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
Branch Predictor	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
Mem 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 Assoc	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 Mult 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 Latency	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 Div Latency	30	10	26	17	24	33	40	33	19	10	10	41	8	301
L2 Cache Assoc	23	19	14	19	33	27	5	39	37	18	42	21	12	309
Int Mult/Div Units	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
Fetch Queue Entries	43	13	27	30	23	20	19	37	9	25	23	34	14	317
Branch MPred Penalty	11	24	41	21	4	43	20	32	11	22	39	35	23	326
FP Div 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 Assoc	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 Assoc	26	27	38	25	20	31	42	12	29	30	22	33	22	357
BTB Assoc	22	18	35	32	10	32	17	30	34	43	27	36	25	361
D-TLB Assoc	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 Latency	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 Parameter #2	28	42	21	39	32	14	13	36	42	29	43	18	30	387
Int Mult Latency	29	40	42	36	13	26	29	21	15	41	35	31	41	399
FP Mult/Div Units	41	22	43	40	41	18	28	38	28	31	31	19	20	400
FP Sqrt Latency	38	29	40	33	35	5	26	43	41	24	24	39	37	414
RAS 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 Parameter #1	19	37	29	43	30	36	43	42	35	23	40	23	36	436

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. Therefore, to improve the data analysis methodology, we describe a method that simultaneously considers the effect of an enhancement on all of the processor's parameters, thereby analyzing the enhancement's effect at a higher-level.

Our proposed method uses the PB design to analyze the effect on the processor's parameters before and after the application of the enhancement. By using this method, the user can determine the enhancement's effect on the processor's parameters and/or determine the significance of the enhancement's parameters.

By comparing the sum-of-ranks for each parameter before and after the application of the enhancement, the user can determine how the enhancement actually affects the processor. For example, if the L1 D-Cache size and associativity sharply drop in significance due to an enhancement (i.e. those two parameters are less of a bottleneck with the enhancement than without it), it is reasonable to conclude that that particular enhancement does a good job of improving memory performance. However, the particular enhancement also may cause a sharp rise in the significance of the memory ports and the number of LSQ entries. Therefore, it also 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 efficacy, utility, and mechanics of this method, we analyze the effect that instruction precomputation [31] has on the processor. Instruction precomputation is similar to value reuse [27] 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 the two techniques is that instruction precomputation uses profiling to statically identify the highest frequency redundant computations instead of identifying them at run-time. In instruction precomputation, the redundant computations are loaded into the precomputation table before the program begins and are never updated. By contrast, value reuse continually updates the value reuse table with the most current computations.

Table 9 shows the effect of instruction precomputation with a 128-entry precomputation table on the processor. While Table 9 represents the "after" case, Table 6 represents the "before" case.

A comparison of the 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. Instruction precomputation changes only the relative ordering of the significant parameters, 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 (i.e. biggest change in the sum of ranks) is the number of integer ALUs. This result is expected since most of the instructions that instruction precomputation eliminates would have executed on an integer ALU. In other words, by using instruction precomputation, the impact of the number of integer ALUs on the processor's performance decreases in significance.

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 enhancement parameters and how their ranks compare to ranks of the processor 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

While there are several studies that are related to this work, we did not find any that directly focused on simulation methodology. Most of the related work focused on either simulator validation, decreasing the simulation time by modeling the processor's performance with statistical methods, or improving the accuracy and precision of simulation results. Other previous work performed sensitivity analyses of key processor parameters or described a method for classifying benchmarks. This paper builds upon previous work by adding statistical rigor to the simulation setup and analysis phases.

Simulator Validation, Processor Modeling, and Improving Simulator Accuracy – The authors of several papers described their simulator validation experiences. Black and Shen [2] described a method of validation that iteratively improves the cycle count accuracy of the performance model, as compared to the actual processor. Their results show that errors were still present in their simulation model, even after a long period of debugging, and that those errors could be revealed only after comparing the performance model to the actual processor. Desikan *et al* [5] measured the difference in the execution times between the Alpha version of the SimpleScalar simulator and the Alpha 21264. They found that simulators that model a generic machine (such as SimpleScalar) generally reported higher IPCs than simulators that were validated against a real machine. On the other hand, unvalidated simulators that targeted a specific machine usually **underestimated** the performance. Gibson *et al* [8] described the types of errors that were present in the FLASH simulator when compared to the implemented FLASH processor. Their results showed that most simulators can accurately predict the architectural trends if all of the important components have been accurately modeled. Their results also 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 architectural enhancements.

Finally, Glamm and Lilja [10] verified the functional correctness of a simulated ISA by comparing the simulated and actual processor states after each instruction. A difference in the states revealed the presence of an error.

A few papers described statistical methods for reducing the complexity of a simulator and, thereby, the resulting simulation time. Noonburg and Shen [20] described a method that uses a program trace along with probabilistic models to estimate the performance of the processor given a particular processor configuration. By using probabilities to account for how long the instructions are in a particular state, 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. The HLS simulator [22] uses statistical profiles and symbolic execution to estimate processor performance. The statistical profile stores program statistics (basic block sizes, etc.) while the symbolic code transforms the instruction stream into a control-flow graph of blocks that contain the necessary resource requirements to execute that block. The HLS simulator estimates the execution time to within 10% of SimpleScalar's execution time.

Cain *et al* [4] measured the effects of the operating system and I/O on the simulator's accuracy. They integrated SimOS-PPC 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. Generally, their results demonstrated the need to integrate an operating system into the simulator for increased accuracy and precision.

Effect of Key Processor Parameters – Skadron *et al* [26] analyzed the trade-offs between the instruction-window size, branch prediction accuracy, and the sizes of 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. 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. As a result, without first quantifying the effect of the most significant interactions, the conclusions that were drawn from these results cannot be taken completely at face value.

Benchmark Classification – Eeckhout *et al* [7] used statistical data analysis techniques such as principal component analysis and cluster analysis 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. The key difference between their method of grouping benchmarks and our method is that their method is predicated on defining a set of metrics that encompasses all of the key factors that affect the performance. The deficiency of this approach is that it assumes that all significant metrics have been incorporated into the statistical design without the benefit of simulations. However, since it is possible for two unrelated processor parameters to interact, picking metrics to identify the effect of either parameter does not necessarily cover the effect of their interaction. Our approach, on the other hand, does not make that assumption; instead, all parameters are weighted equally. Finally, our method can seamlessly classify benchmarks based on other metrics, such as the power consumption, for instance, while their method requires a redefinition of the metrics.

Giladi and Ahituv [9] identified the redundant benchmarks in the SPEC 92 benchmark suite. A redundant benchmark is one that, if removed, does not significantly change the SPEC number for that benchmark suite. Their results show that 13 of the 20 benchmarks in the SPEC 92 suite were redundant. Their method of determining redundant benchmarks is significantly different from our method for at least two reasons. First of all, their method is completely based on approximating the SPEC number. Secondly, since the SPEC number is calculated by normalizing the execution times to a base system, there is no direct connection to the effect that each benchmark has on the processor. By contrast, our method focuses exclusively on the benchmark's effect on the processor.

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. Finally, 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 data analysis. The first method focuses on how the processor parameter values are chosen. 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 proposed method groups benchmarks together if they have a similar effect on the processor. Two benchmarks have similar effects on the processor if their 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 data analysis 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

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

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

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

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

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

[6] 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.

[7] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Workload Design: Selecting Representative Program-Input Pairs"; International Conference on Parallel Architectures and Compilations Techniques, 2002.

[8] 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.

[9] R. Giladi and N. Ahituv, "SPEC as a Performance Evaluation Measure", IEEE Computer, Vol. 28, No. 8, August 1995, Pages 33-42.

[10] R. Glamm and D. Lilja, "Automatic Verification of Instruction Set Simulation Using Synchronized State Comparison", Annual Simulation Symposium, 2001.

[11] 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.

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

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

[14] A. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research", Vol. 1, June 2002.

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

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

[17] D. Lilja, "Measuring Computer Performance", Cambridge University Press, 2000.

[18] 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.

[19] D. C. Montgomery, "Design and Analysis of Experiments", Third Edition, Wiley 1991.

[20] D. Noonburg and J. Shen, "A Framework for Statistical Modeling of Superscalar Processor Performance ", International Symposium on High Performance Computer Architecture, 1997.

[21] 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.

[22] M. Oskin, F. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs", International Symposium on Computer Architecture, 2000.

[23] R. Plackett and J. Burman, "The Design of Optimum Multifactorial Experiments", Biometrika, Vol. 33, Issue 4, June 1956, Pages 305-325.

[24] J. Silc, B. Robic, and T. Ungerer, "Processor Architecture: From Dataflow to Superscalar and Beyond", Springer-Verlag, 1999.

[25] D. Sima, T. Fountain, and P. Kacsuk, "Advanced Computer Architectures, A Design Space Approach", Addison Wesley Longman, 1997.

[26] 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.

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

[28] S. Song, M. Denman, and J. Chang, "The PowerPC 604 RISC Microprocessor", IEEE Micro, Vol. 14, No. 5, October 1994, Pages 8-17.

[29] 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.

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

[31] J. Yi, R. Sendag, and D. Lilja, "Increasing Instruction-Level Parallelism with Instruction Precomputation", Euro-Par 2002.

[32] J. Yi and D. Lilja, "Effects of Processor Parameter Selection on Simulation Results", MSI Report 2002/146, 2002.

[33] J. Yi, D. Lilja, and D. Hawkins, "A Statistically Rigorous Approach for Improving Simulation Methodology", ARCTiC Technical Report 02-07, 2002.