HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs

Mark Oskin, Frederic T. Chong and Matthew Farrens
Department of Computer Science
University of California at Davis

Abstract

As microprocessors continue to evolve, many optimizations reach a point of diminishing returns. We introduce HLS, a hybrid processor simulator which uses statistical models and symbolic execution to evaluate design alternatives. This simulation methodology allows for quick and accurate contour maps to be generated of the performance space spanned by design parameters. We validate the accuracy of HLS through correlation with existing cycle-by-cycle simulation techniques and current generation hardware. We demonstrate the power of HLS by exploring design spaces defined by two parameters: code properties and value prediction. These examples motivate how HLS can be used to set design goals and individual component performance targets.

1 Introduction

In this paper, we introduce a new methodology of study for microprocessor design. This methodology involves statistical profiling of benchmarks using a conventional simulator, followed by the execution of a hybrid simulator that combines statistical models and symbolic execution. Using this simulation methodology, it is possible to explore changes in architectures and compilers that would be either impractical or impossible using conventional simulation techniques.

We demonstrate that a statistical model of instruction and data streams, coupled with a structural simulation of instruction issue and functional units, can produce results that are within 5-7% of cycle-by-cycle simulation. Using this methodology, we are able to generate statistical contour maps of microprocessor design spaces. Many of these maps verify our intuitions. More significantly, they allow us to more firmly relate previously decoupled parameters, including: instruction fetch mechanisms, branch prediction, code generation, and value prediction.

To demonstrate the power of this statistical simulation model, we present a study that relates program code characteristics, such as basic block size and dynamic dependence distance, to various machine parameters. In addition, we explore value prediction.

In the next section, we describe the HLS simulator. Next in Section 3 we validate HLS against conventional simulation techniques. In Section 4, the HLS simulator is used to explore various architectural parameters. Then in Section 5, we discuss our results and summarize some of the potential pitfalls of this simulation technique. In Section 6, we discuss related work in this field. Finally, future work is discussed in Section 7, and Section 8 presents the conclusions.

2 HLS: A Statistical Simulator

HLS is a hybrid simulator which uses statistical profiles of applications to model instruction and data streams. HLS takes as input a statistical profile of an application, dynamically generates a code base from the profile, and symbolically executes this statistical code on a superscalar microprocessor core. The use of statistical profiles greatly enhances flexibility and speed of simulation. For example, we can smoothly vary dynamic instruction distance or value predictability. This flexibility is only possible with a synthetic, rather than actual, code stream. Furthermore, HLS executes a statistical sample of instructions rather than an entire program, which dramatically decreases simulation time and enables a broader design space exploration which is not practical with conventional simulators. In this section, we describe the HLS simulator, focusing on the statistical profiles, method of simulated execution, and validation with conventional simulation techniques.

2.1 Architecture

The key to the HLS approach lies in its mixture of statistical models and structural simulation. This mixture can be seen in Figure 1, where components of the simulator which use statistical models are shaded in gray. HLS does not simulate the precise order of instructions or memory accesses in a particular program. Rather, it uses a statistical profile of an application to generate a synthetic instruction stream. Caches are also modeled as a statistical distribution.

Once the instruction stream is generated, HLS symbolically issues and executes instructions much as a conventional simulator does. The structural model of the processor closely follows that of the SimpleScalar tool set [BA97], a widely used processor simulator. This structure, however, is general and configurable enough to allow us to model and validate against a MIPS R10K processor in Section 3.2.

The overall system consists of a superscalar microprocessor, split L1 caches, a unified L2 cache, and a main memory. The processor supports out-of-order issue, dispatch and completion. It has five major pipeline stages: instruction fetch, dispatch, schedule, execute, and complete. The similarity to SimpleScalar is not a coincidence: the SimpleScalar tools are used to gather the statistical profile needed by HLS. We will also compare results from SimpleScalar and HLS to validate the hybrid approach. The simulator is fully programmable in terms of queue sizes and inter-pipeline stage bandwidth; however, the baseline archi-

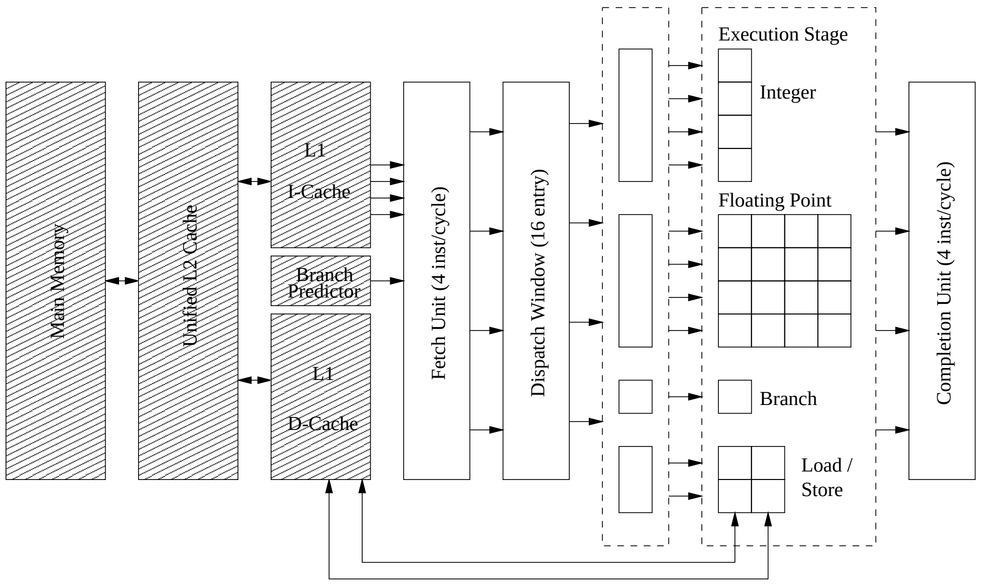


Figure 1: Simulated Architecture

Parameter	Value
Instruction fetch bandwidth	4 inst.
Instruction dispatch bandwidth	4 inst.
Dispatch window size	16 inst.
Integer functional units	4
Floating point functional units	4
Load/Store functional units	2
Branch units	1
Pipeline stages (integer)	1
Pipeline stages (floating point)	4
Pipeline stages (load/store)	2
Pipeline stages (branch)	1
L1 I-cache access time (hit)	1 cycle
L1 D-cache access time (hit)	1 cycle
L2 cache access time (hit)	6 cycles
Main memory access time (latency+transfer)	34 cycles
Fetch unit stall penalty for branch mis-predict	3 cycles
Fetch unit stall penalty for value mis-predict	3 cycles

Table 1: Simulated Architecture configuration

tecture was chosen to match the baseline SimpleScalar architecture. The various configuration parameters are summarized in Table 1.

2.2 Statistical profiles

In order to use the HLS simulator, an input profile of a real application must first be generated. Once the profile is generated, it is interpreted, a synthetic code sample is constructed, and this code is executed by the HLS simulator. Since HLS is probability-based, the process of execution is usually repeated several times in order to reduce the standard deviation of the result. This overall process flow is depicted in Figure 2.

Statistical data collection of actual benchmarks is performed in the following manner:

- The program code is compiled and a conventional binary is produced. This binary is targeted for the SimpleScalar tool suite.
- The binary is run on a modified SimpleScalar simulator. The statistical profile consists of the basic block size and distribution and a histogram of the dynamic instruction distance between instructions for each major instruction type (integer, floating-point, load, store, and branch). This dynamic instruction distance forms a critical aspect of the statistical profile and will be discussed further in this section.
- The binary is also run on a standard SimpleScalar simulator. Here, statistics about cache behavior and branch prediction accuracy are collected.

These steps were performed on the SPECint95 benchmark suite. A summary of the results (less dynamic dependence information) is presented in Table 2. Note that the average basic block size is around 5 instructions, and the wide variability in each of these averages (as indicated by the high standard deviation). This variability was modeled in the simulator.

Across most benchmarks, we found a relatively high branch predictability of 86-91%. This figure is a combination of both correctly predicted branches using the 2-level bimodal branch predictor and those that were statically determined (such as jumps). The exception is the go benchmark which has very poor predictor performance.

Table 2 shows that two distinctive L1 I-cache behaviors are occurring. The compress, ijpeg, and li benchmarks have extremely good L1 I-cache hit rates, while gcc, m88ksim, perl and go have high but not extremely high I-cache hit rates.

Dynamic instruction distance (DID), or the distance between an instruction and the instructions that are dependent upon it within the dynamic instruction stream, is a significant statistical component in the performance of an application. Intuitively, longer DID permits more overlap of instructions within the execution core of the processor. In Section 4, we demonstrate this intuitive result, but note

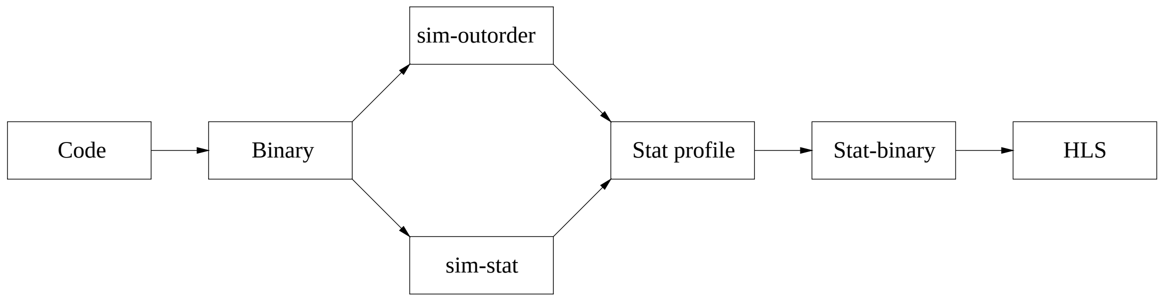


Figure 2: Simulation process

Value	perl	compress	gcc	go	ijpeg	li	m88ksim	vortex
Basic block size (μ)	5.21	4.69	4.93	5.96	6.26	4.39	6.25	5.78
Basic block size (σ)	3.63	4.91	4.57	5.16	12.65	3.04	5.33	5.54
Integer Instructions	30%	42%	38%	51%	53%	34%	54%	31%
FP Instructions	1%	<1%	0%	0%	0%	0%	0%	0%
Load Instructions	31%	22%	27%	23%	22%	26%	21%	26%
Store Instructions	18%	12%	15%	8%	10%	17%	9%	25%
Branch Instructions	19%	21%	20%	17%	16%	23%	16%	18%
Branch Predictability	91.4%	86.3%	87.6%	81.8%	90.0%	87.9%	91.8%	97.4%
L1 I-cache hit rate	96.4%	99.9%	93.7%	95.0%	99.1%	99.9%	94.1%	90.2%
L1 D-cache hit rate	99.9%	84.6%	97.8%	96.6%	99.1%	98.4%	99.6%	97.8%
L2 cache hit rate	99.9%	99.1%	97.3%	96.9%	94.5%	99.8%	99.1%	97.6%

Table 2: Statistical baseline parameters for SPECint95 (ref input, first 1 billion instructions)

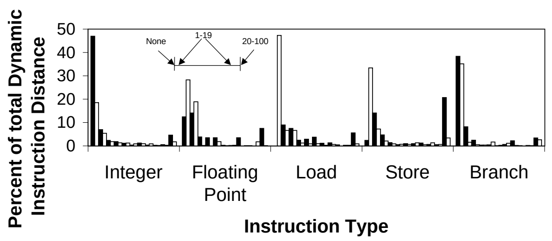


Figure 3: Dynamic instruction distances (perl)

the limitations on performance of solely extending the DID. Here we note that overlap of instructions or instruction level parallelism (ILP) is a contributor to performance. In order to correctly model a superscalar microprocessor, HLS must use DID information from real programs.

The DID information for the perl SPECint95 benchmark is presented in Figure 3. (Although the DID is program dependent, space considerations permit us to only illustrate the consolidated DID information for a single benchmark.) While a parameterized model can be formulated to characterize the DID for a specific application, we found DID to be a critical factor in the accuracy of HLS. Hence, we chose to directly extract a histogram from SimpleScalar of DID information. Note that for each instruction type, two histograms of DID are utilized (one for each possible dependence).

2.3 Statistical Code Generation

Once the statistical profile is generated, the first step in the simulation process is to generate the symbolic code sample. This symbolic code consists of instructions contained in basic blocks that are linked together into a static program flow-control graph, very much like conventional code. The difference is in the instructions themselves. Instead of containing actual arguments, each "instruction" contains the following set of statistical parameters:

- *Functional unit requirements* are described by a single parameter that specifies which functional unit is required inside the execution core of the processor to complete the instruction. This requirement is statically assigned to each instruction at code generation time and the distribution of functional unit requirements follows the breakdown of instruction types as shown in Table 2. This is done on a program-by-program basis.
- *L1I-p, L1D-p, L2I-p, L2D-p:* The L1 I-cache, L1 D-cache and L2 cache behavior is classified into four normal distributions with a mean centered around the hit rate gathered from direct program simulation using SimpleScalar. Although the simulator permits the L2 cache to be split, for this study L2I-p and L2D-p are set equal. This corresponds to the baseline SimpleScalar configuration.
- *Dynamic instruction distances:* These parameters are determined from the histogram profile obtained from simulation with SimpleScalar. They determine which instructions the current instruction is dependent upon,

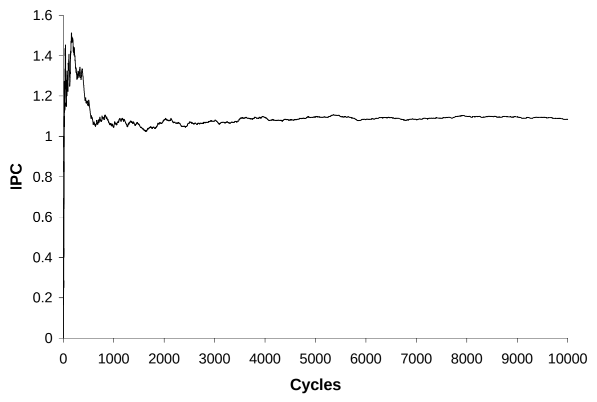


Figure 4: Simulation IPC convergence time (cycles)

in a dynamic sense. These are not linked statically, but rather only the distance is stored. As instructions are fetched, the dependencies are finally resolved at execution-time. Care is taken in the code generator to prevent the DID from pointing to a store or a branch, which would make an instruction dependent upon another instruction that in real code would not ordinarily form the basis for a dependence.

Finally, each basic block is of a size determined by the normal distribution of code sizes gathered from simulation with SimpleScalar. Each basic block terminates with a branch, and the predictability of that branch is assigned by the code generator. The predictability is assigned by the branch prediction accuracy gathered from direct simulation. The various branch prediction accuracies and basic block size parameters for each application are shown in Table 2.

2.4 Execution

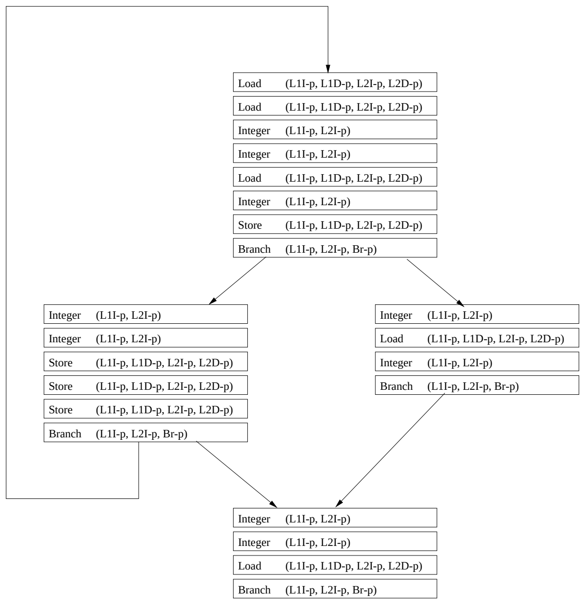
Execution of the HLS simulator is similar to a conventional simulator. The instruction fetch stage interacts with the branch predictor and I-cache system to fetch "instructions" and send them to the dispatch stage. The dispatch stage interprets these instructions and sends them to the reservation stations. Once all input dependencies have been resolved, the instruction moves from the reservation unit to an available functional unit pipeline. After executing in a functional unit, the result is "posted" to the completion unit if a result bus is available.

The observed IPC from HLS converges quickly during execution. Figure 4 depicts the IPC within HLS as instructions flow through the processor core for the first ten-thousand simulated machine cycles. Note that IPC rapidly settles down after the first one-thousand cycles, and after six-thousand cycles remains relatively constant.

The structural resources within the statistical simulator are sized to match the structural resources within the sim-outorder SimpleScalar simulator. This provides the basis for the validation of HLS using SimpleScalar, as well as the use of SimpleScalar as a source of program statistical profiles.

3 Validation and Limitations

Once a statistical profile is generated, that profile is executed several times and an average instructions per cycle



Load (L1I-p, L1D-p, L2I-p, L2D-p)
Load (L1I-p, L1D-p, L2I-p, L2D-p)
Integer (L1I-p, L2I-p)
Integer (L1I-p, L2I-p)
Load (L1I-p, L1D-p, L2I-p, L2D-p)
Integer (L1I-p, L2I-p)
Store (L1I-p, L1D-p, L2I-p, L2D-p)
Branch (L1I-p, L2I-p, Br-p)
Integer (L1I-p, L2I-p)
Integer (L1I-p, L2I-p)
Store (L1I-p, L1D-p, L2I-p, L2D-p)
Store (L1I-p, L1D-p, L2I-p, L2D-p)
Store (L1I-p, L1D-p, L2I-p, L2D-p)
Branch (L1I-p, L2I-p, Br-p)
Integer (L1I-p, L2I-p)
Load (L1I-p, L1D-p, L2I-p, L2D-p)
Integer (L1I-p, L2I-p)
Branch (L1I-p, L2I-p, Br-p)
Integer (L1I-p, L2I-p)
Integer (L1I-p, L2I-p)
Load (L1I-p, L1D-p, L2I-p, L2D-p)
Branch (L1I-p, L2I-p, Br-p)

Figure 5: Simulated code example

Benchmark	Simple-scale IPC	HLS IPC	HLS IPC σ	Error
perl	1.10	1.12	0.03	2.3%
compress	1.70	1.76	0.05	3.2%
gcc	0.89	0.93	0.04	4.4%
go	0.87	0.86	0.02	0.1%
ijpeg	1.67	1.74	0.04	3.7%
li	1.40	1.38	0.07	1.8%
m88ksim	1.32	1.30	0.04	1.7%
vortex	0.87	0.83	0.02	4.8%

Table 3: Correlation between SimpleScalar and HLS Simulators for SPECint95 (test input)

Benchmark	Simple-scale IPC	HLS IPC	HLS IPC σ	Error
perl	1.27	1.32	0.05	4.2%
compress	1.18	1.25	0.06	5.5%
gcc	0.92	0.96	0.03	3.9%
go	0.94	1.01	0.04	6.8%
ijpeg	1.67	1.73	0.06	3.9%
li	1.62	1.50	0.06	7.2%
m88ksim	1.16	1.14	0.03	1.5%
vortex	0.87	0.83	0.03	5.1%

Table 4: Correlation between SimpleScalar and HLS Simulators for SPECint95 (ref input)

Optimization Level	Simple-scalar IPC	HLS IPC	HLS IPC σ	Error
none	1.23	1.35	0.09	9.9%
-O	1.09	1.11	0.04	2.3%
-O2	1.16	1.18	0.03	2.0%
-O3	1.33	1.33	0.05	0.2%
Full opt.	1.40	1.38	0.07	1.8%

Table 5: Correlation between SimpleScalar and HLS Simulators for xlisp compiled with various optimization levels

Benchmark	R10K IPC	HLS IPC	HLS IPC σ	Error
perl	1.01	1.09	0.05	7.8%
compress	0.70	0.69	0.04	2.6%
gcc	0.93	0.96	0.05	3.8%
go	0.99	0.98	0.06	0.9%
ijpeg	1.45	1.40	0.09	4.0%
li	0.85	0.90	0.07	6.0%
m88ksim	1.15	1.15	0.08	0.1%
vortex	0.83	0.82	0.06	1.0%

Table 6: Correlation between MIPS R10k and HLS Simulators for SPECint95 (ref input)

(IPC) is calculated. The IPC is then used to gauge the relative performance differences between two experiments. It is important, however, to validate this simulation technique and ensure that the IPC reported by HLS is relevant. Furthermore, the statistical model is not perfect. Clearly, we cannot utilize the model to predict with perfect accuracy the performance of a benchmark over all possible ranges of processor configurations and component performances. It is important to find where the model works and where it does not.

3.1 Execution Correlations

Tables 3 and 4 list the experimental results from executing the SPECint95 benchmarks on both the test and reference inputs in SimpleScalar versus the statistical simulator. Across the board, we note that the error (the difference between the two simulation techniques) is less than 4.8% and 7.2% respectively. This is for benchmarks with significantly different cache behaviors and code profiles. This agreement with such small error across eight different benchmark applications and two different input sets is encouraging and a substantial correlation point for the hybrid statistical-symbolic execution model of study.

3.2 Hardware Correlation

While HLS was designed to model the same architecture as SimpleScalar, it can be configured to model a MIPS R10K processor. We validated against a 250 MHz MIPS R10K processor in an SGI Octane system running IRIX V6.5. To obtain the statistical profile required for HLS, we gathered the DID from SimpleScalar, and cache and branch predictor behavior using the "Perfex" performance monitoring tool available on the IRIX operating system [ZLTI96].

The correlations of HLS with this processor are shown in Table 6. Note that HLS correlates to within 7.7%, with an average error of 3.2% across all benchmarks.

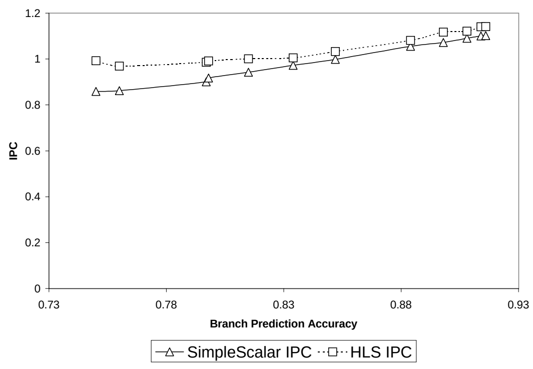


Figure 6: Correlation between SimpleScalar and HLS for Branch Prediction Accuracy.

There is also one other statistical difference of note between modeling SimpleScalar and modeling the R10K. Cache hit rates for SimpleScalar can be modeled accurately in HLS with a uniform distribution. Hit rates for the R10K, however, require a more complex model which uses a normal distribution with a variance corresponding to the R10K. We suspect that this is necessary to account for machine and operating system effects, such as servicing program I/O and context switching, which alter cache behavior.

3.3 Single-value Correlations

Although several validation experiments are possible, in this paper we focus on those that form the first-order effects on performance: cache and branch prediction behavior. We will show that the HLS simulator behaves similarly to SimpleScalar under varying branch prediction accuracy, L1 I-cache hit rate, L1 D-cache hit rate, and compiler optimization levels.

We will also show that there are ranges of statistical parameters where the HLS simulator is not accurate. Although the HLS simulator precisely models the structure of a superscalar microprocessor, several sources of statistical error can be introduced. It is important to identify where the HLS simulator will have unacceptable error, so that it is not used inappropriately.

Figure 6 plots IPC versus branch prediction accuracy as reported from both SimpleScalar and HLS running the perl SPECint95 benchmark application. SimpleScalar branch prediction accuracy was varied by modifying the branch prediction table size. For each SimpleScalar run, branch prediction accuracy was measured and input into HLS to produce a corresponding data point. Note that, above 80% branch prediction accuracy, the HLS and SimpleScalar simulators are within 6% of each other. At less than 80% accuracy, HLS and SimpleScalar begin to diverge. At 75% accuracy, the error between the HLS and SimpleScalar simulators is 15%. From these results it is clear that the branch predictor model within the HLS is usable with branch prediction accuracies greater than 80%.

Figure 7 plots the reported IPC as we vary the L1 I-cache hit rate for perl. Using SimpleScalar, the hit rate is varied by changing the cache size, while in HLS the hit rate is input directly. The figure shows that for I-cache hit rates

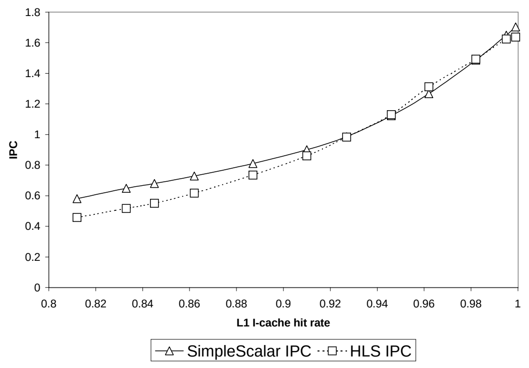


Figure 7: Correlation between SimpleScalar and HLS for L1 I-cache hit rate.

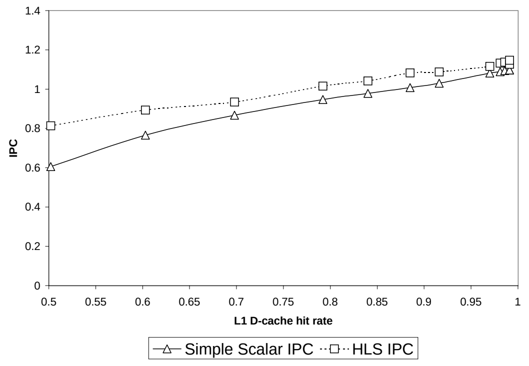


Figure 8: Correlation between SimpleScalar and the Statistical Simulator for L1 D-cache hit rate.

above 90%, HLS and SimpleScalar are within 5% of each other. For hit rates in the 80-90% range, they agree within 20%. The simulation model performs well on all of the SPECint95 benchmarks because they all exhibit L1 I-cache hit rates greater than 90% in the baseline configuration. Future work may seek a more accurate L1 I-cache model to bring down the observed performance differences.

Figure 8 is similar to figure 7, except that the IPC is plotted for varying L1 D-cache hit rates. The HLS and SimpleScalar simulators can be seen to be within 7% of each other with L1 D-cache hit rates greater than 80%. Between 70-80% hit rate, the error is 8%. The error climbs to 15-35% with cache hit rates from 50-70%. Thus, with cache hit rates greater than 80%, the HLS simulator provides reasonable results. Similar to L1 I-cache behavior, more accurate statistical models will be required to model poorly performing data caches. For this study, the focus will be on cache hit rates greater than 80%.

Finally, Table 5 lists the reported IPC from SimpleScalar and HLS when executing the xlisp benchmark compiled under various degrees of compiler optimization. At any optimization level, the HLS simulator is very accurate, with errors within the 3% range. When no optimization is used,

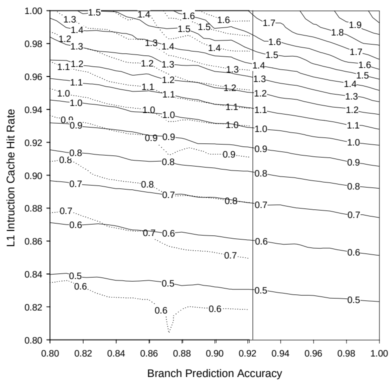


Figure 9: Multi-value correlation between SimpleScalar and the Statistical Simulator for L1 I-cache hit rate and branch prediction accuracy. (Perl)

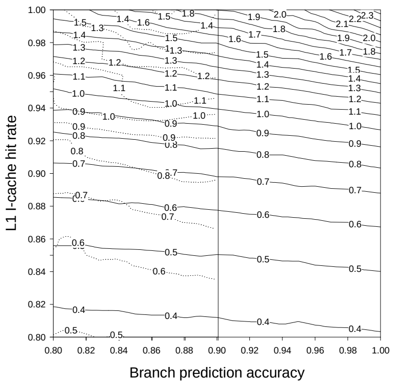


Figure 10: Multi-value correlation between SimpleScalar and the Statistical Simulator for L1 I-cache hit rate and branch prediction accuracy. (Xlisp)

the error climbs to 10%. This is attributable to the increased number of NOP instructions introduced by the compiler. These instructions cause the statistical simulator to acquire too high of a non-dependence factor in the dynamic instruction distance gathering phase. Unlike a conventional simulator, HLS will distribute these non-dependencies across all integer instructions and not focus them upon NOP instructions that are inserted after memory load references. We expect that a more detailed statistical gathering process will reduce this error, but we are primarily concerned with utilizing optimized versions of benchmarks.

3.4 Multi-value Correlations

In Section 3.1, correlations across benchmarks were presented in which several values varied, including basic block size, dynamic instruction distance, cache hit ratios, branch prediction accuracy, and instruction mix. In Section 3.3, we presented results varying a single parameter. In this section we will present results in which two parameters are systematically varied, in order to further demonstrate the validity of the HLS model.

We will focus on two benchmarks: perl and xlisp. Perl is being utilized as the representative benchmark throughout this paper. However, in this section we also present corre-

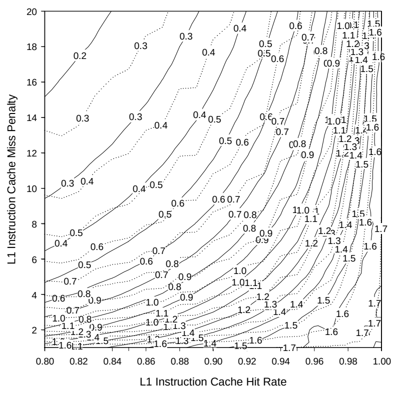


Figure 11: Multi-value correlation between SimpleScalar and the Statistical Simulator for L1 I-cache hit rate vs. L1 I-cache miss penalty. (Perl)

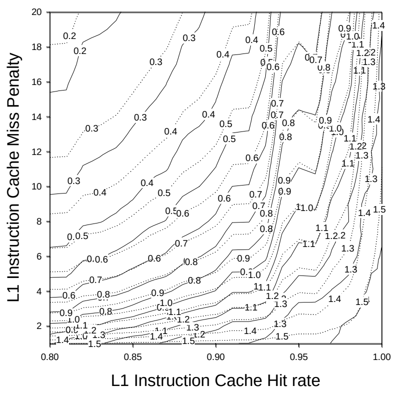


Figure 12: Multi-value correlation between SimpleScalar and the Statistical Simulator for L1 I-cache hit rate vs. L1 I-cache miss penalty. (XLisp)

lations against xlisp due to its interesting cache and branch predictor interactions.

Figures 9 and 10 depict the IPC contour space of perl and xlisp as the L1 instruction cache hit rate is varied against the branch predictor accuracy. The solid lines on the graphs represent contour lines generated from HLS, while the dotted lines were generated in SimpleScalar. To generate the SimpleScalar results we varied the branch predictor table size and instruction cache size. Note that the solid vertical line on each graph represents the limit of the branch predictor accuracy attainable in SimpleScalar – it is the point at which further increases in the branch predictor table size did not increase branch prediction accuracy. As depicted in the graph, HLS and SimpleScalar agree quite accurately for instruction cache hit rates greater than 90%. Below 90% a decreasing level of accuracy is observed.

Figures 11 and 12 depict IPC as instruction cache hit rate is varied against cache miss penalty. Here we see a similar level of accuracy to the previous instruction cache correlations. Instruction cache hit rates above 90% are very accurate, while cache hit rates below 90% are suitable for trend analysis only. One interesting feature of Figure 12 is the non-smooth contour map between 0.93 and 0.97 on the cache hit ratio axis. This non-smooth behavior leads to an interesting discussion about the use of HLS and the caution a user should have about its results. If parameters are varied within a real superscalar processor other machine parameters will not remain constant. While the HLS simulator allows the user to fix all other machine parameters, one must be aware that in reality they will change. This effect can be seen as the unified L2 cache hit rate and branch prediction accuracy are indirectly altered by varying the cache hit rate and miss penalty. This topic will be discussed further in Section 5.

3.5 Summary and Discussion

From these results, we conclude that HLS represents a viable method of simulation within the ranges of statistical profiles that we are interested in. Outside of these ranges, the accuracy of HLS is suitable only for general trend analysis and not for discrimination of closely related points. There are three major sources of error in the HLS simulator:

First, HLS generates random code based upon a statistical profile of the original source. One source of error in this code generation process is that the relationship between instructions is mostly lost. Dynamic dependence information is maintained, but it is used imprecisely with dependencies between instructions that do not correspond exactly to the original source file. Although the generated code and the original code maintain the same statistical nature, they are in fact substantially different.

Second, the cache and branch predictor are modeled as simple normal distributions. Except where noted, these normal distributions are uniform distributions with a prescribed accuracy or hit rate. It is not a coincidence that the most accurate correlation between the SimpleScalar and HLS simulators occur when the rate approaches 100%. With a perfect branch predictor or a perfect cache, the HLS and SimpleScalar cache models would be equivalent.

Third, there is no load-value or instruction fetch miss-value correlation modeled in the simulator. We suspect that not modeling these correlation effects contribute to the cache model inaccuracies at hit rates below 90%.

Despite these sources of error, if areas of study are confined to where HLS is accurate, interesting experiments can be performed that would otherwise be extremely hard or impossible to do using conventional simulation techniques. We present the results from two of these experiments in the next section.

4 Application of HLS

In order to demonstrate the power of this methodology, we decided to explore two aspects of processor performance that are uniquely suited to study using this hybrid statistical simulation approach: program characteristics and value prediction. Our results indicate that HLS correlates well with the entire SPECint95 benchmark suite. The results presented in this section will be for the perl benchmark, chosen because it has an IPC (1.1) that is in the middle of the SPECint suite.

Most of our results are presented using iso-instructions-per-cycle (iso-IPC) contour plots, which relate two parameters and plot a two-dimensional view of a three-dimensional space. Each contour line in the graph represents a constant IPC. An example of such a plot is depicted in Figure 13. Such iso-IPC plots graphically depict the design-space of

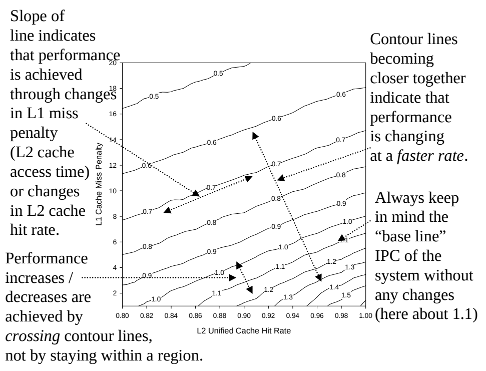


Figure 13: L2 cache hit rate versus L1 cache miss penalty

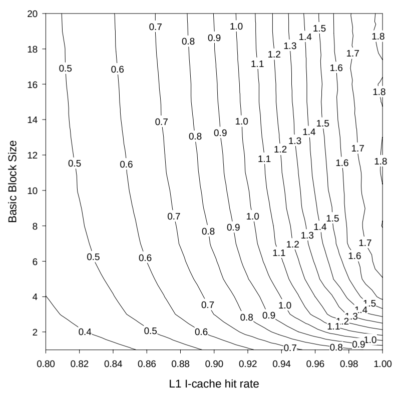


Figure 15: L1 I-cache hit rate versus basic block size

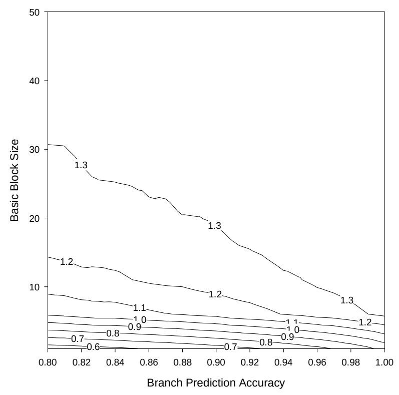


Figure 14: Branch prediction accuracy versus basic block size

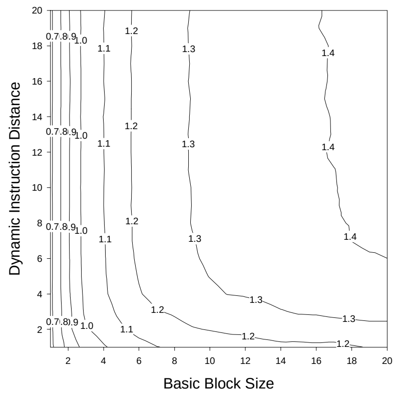


Figure 16: Basic block size versus dynamic instruction distance

two parameters. Each is generated utilizing over 400 individual data-points collected from 8,000 runs of the HLS simulator. Each contour map presented requires approximately six hours to generate on a 450 Mhz Pentium-II. By lowering the number and length of each iteration it is possible to generate a contour map in 15 minutes. Such a map does have higher standard deviation in the results, but is accurate enough to provide quick insight to the processor designer.

4.1 Varying Code Properties

The HLS simulation methodology allows the user to systematically vary the properties of the code being simulated. This is a very powerful capability and provides a way to study the effects of two code properties normally very hard to modify: the basic block size and the dynamic instruction distance.

Figure 14 depicts application performance as the branch prediction accuracy and basic block size change. This figure shows that given the accuracy of current branch predictors, moderate increases in basic block size will bring about noticeable performance improvements. As expected, the largest performance gains occur as basic block sizes grow from one to fifteen instructions. Performance gains quickly diminish after this, with no significant performance gains observed when the block size is greater than 35 instructions.

In Figure 15, the relationship between basic block size and L1 I-cache hit rate is presented. This figure shows that, for basic block sizes between 1 and 4 instructions, the basic block size has the largest effect on performance. Beyond 4 instructions, however, the L1 I-cache becomes the dominating factor.

Similarly, in Figure 16, the basic block size versus dynamic instruction distance is plotted. Two observations can be made from studying this graph: First, moderate dynamic instruction distances are required for reasonable performance. This is evident from the lower regions of the graph, with DID values of four or less. Second, the basic block size quickly becomes the dominating factor in performance.

4.2 Value Prediction

Value prediction is currently an active area of research. Several value prediction schemes have been proposed in the literature, but these schemes have shown only moderate increases in IPC [LS96] [TS99] [CRT99] [MGS99] [GM98] [LWY00]. To simulate value prediction a value predictor and the ability to do speculative execution based on predicted values was added to the basic HLS structural model.

The value predictor is added to the dispatch stage. Instructions determined to be predictable by the predictor unit are sent speculatively to the completion stage. A copy of that instruction is also sent to the scheduling phase to perform a check. Within the scheduling phase, if an instruction's operands are not directly available, but predicted operand values are, then the instruction may execute speculatively on the predicted operands. In this case, no check of the instruction is performed, because if the instruction was executing on incorrect data values a mis-speculation would have occurred earlier on a predicted instruction sent to the completion stage speculatively by the dispatch stage.

Our value prediction scheme implements two common mechanisms. First, we introduce a confidence threshold and confidence values associated with a prediction [CRT99]. Speculative execution only occurs if the confidence value of the predictor is greater than or equal to the threshold. Second, we use the same mechanism on mis-speculations that the branch predictor uses to roll-back from a mis-speculated branch [CRT99] [GM98]: The fetch, dispatch, schedule, execution and completion units are flushed of mis-speculated values, and the fetch unit is directed to refetch from the start of the mis-speculation.

To control value prediction the following additional parameters are introduced into the code stream:

- *Value prediction predictability (VPP-p)* informs the simulator of the inherent predictability [SS97] of the outcome of the instruction. This parameter is modeled using a normal distribution. The simulator models value prediction at a high level, not taking into account how the hardware may achieve a certain prediction level. This parameter along with the next two parameters determines the overall value prediction behavior.

- *Value prediction knowledge (VPK-p)* controls how well the value predictor inside the simulator will be able to predict the instruction given the instruction's inherent predictability. This parameter also is modeled using a normal distribution. To understand the difference between VPK-p and VPP-p, consider a last-value predictor versus a stride predictor on the same instruction stream. The VPP-p of that code remains the same, since there will be a certain intrinsic predictability for the stream, but the VPK-p of the stride predictor will be higher than the VPK-p of the last-value predictor.

- *Value prediction confidence (VPC-p)* specifies how confident the predictor is in the prediction. This parameter is also modeled using a normal distribution. HLS compares this parameter to a cut-off threshold to determine when the superscalar core should use a predicted value. For example, an instruction with a low VPK-p and high VPC-p will be predicted, but probably incorrectly (if VPP-p is low). However, a value with a high VPK-p but low VPC-p may not be predicted, even if the value predictor is likely to make a correct prediction.

The goal of value prediction is to break true data dependencies by predicting the data value instead of waiting for it to be generated. Another way to achieve a similar result is by lengthening the dynamic instruction distance, ensuring that dependent instructions have completed by the time an instruction is ready to execute. This relationship is illustrated in Figure 17, which utilizes a fully knowledgeable predictor and varies both the inherent value predictability and dynamic instruction distance within a code stream.

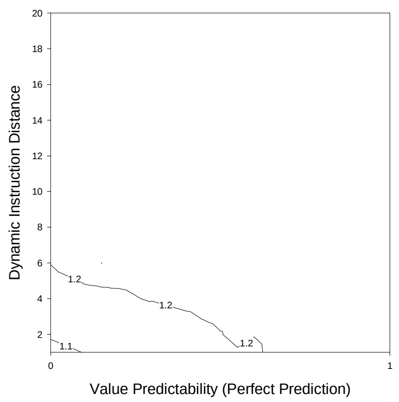


Figure 17: Inherent value predictability versus dynamic instruction distance assuming a perfect predictor

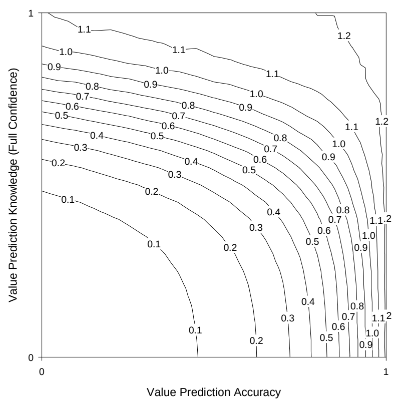


Figure 18: Inherent value predictability versus value predictor knowledge

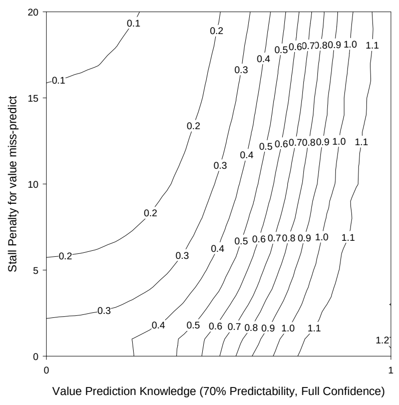


Figure 19: Value prediction knowledge versus mis-speculation penalty

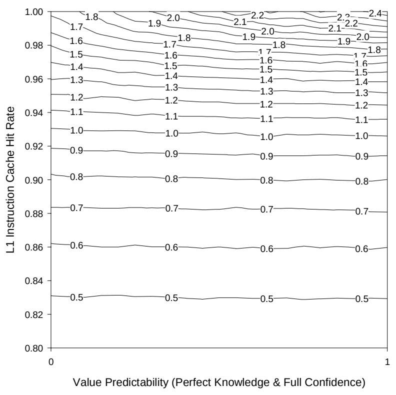


Figure 20: Inherent value predictability versus L1 I-cache hit rate assuming a perfect value predictor

The figure shows the direct relationship between DID and value predictability, and suggests that perhaps any performance gains due to value prediction might also be achieved by optimizing compilers.

Figure 18 explores the trade-off between inherent value predictability and value prediction accuracy. Since the baseline IPC for this graph is 1.1, a substantial region within this graph equates to performance decreases. This is the reason that accurate confidence prediction is important in value prediction schemes. Following the 1.1 iso-IPC contour line, the figure shows that, with moderate predictability, a highly accurate predictor is required to realize any performance gains. As inherent predictability increases beyond 95%, the accuracy of the predictor becomes significantly less important. Finally, this graph shows that the maximum IPC is between 1.2 and 1.3, which is no more than a 20% overall application improvement. And this is only achieved with substantial predictor knowledge and inherent predictability.

Since Figure 18 suggests that mis-speculations in value prediction will occur often, it is interesting to look at the cost of such mis-speculation. Assuming a moderately high inherent predictability rate of 70%, predictor accuracy versus the mis-speculation penalty is depicted in Figure 19. This figure demonstrates that (except for a really poor predictor), performance is dependent upon predictor accuracy, not mis-speculation penalty.

These experiments have explored the relationship between value prediction and dynamic instruction distance and shown that each overcomes data dependencies. However, while increasing dynamic dependence distance does not introduce any negative effects, a poorly implemented value prediction scheme can substantially reduce processor performance. Hence, designers rightly introduce confidence mechanisms. Furthermore, reducing the stall penalty on a mis-predict is not enough to support highly speculative value prediction. In addition, Figure 20 demonstrates that without increases in the instruction fetch rate, even highly accurate value predictors will not provide substantial performance gains.

5 Discussion

The HLS simulator uses a combination of statistical and structural modeling to simulate a superscalar microprocessor. One key property of real code that is not modeled

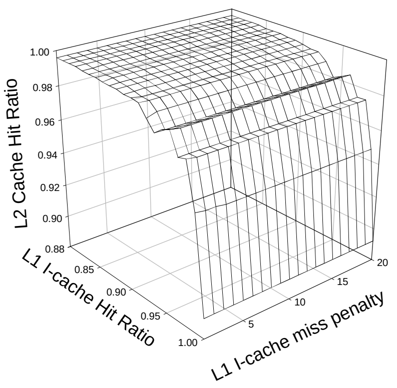


Figure 21: Variation in Unified L2 cache hit rate as L1 I-cache hit rate varies compared to L1 I-cache miss penalty

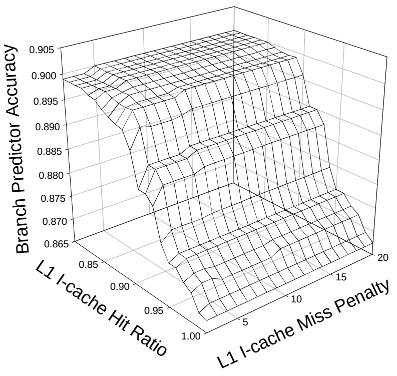


Figure 22: Variation in branch predictor accuracy as L1 I-cache hit rate varies compared to L1 I-cache miss penalty

by the current HLS simulator is the exact ordering of instructions, control flow and dependencies. Interestingly, the SPECint95 benchmarks can be modeled in the aggregate without this information. As long as the binary code is statistically correlated in terms of instructions, control flow, and dependency information, the exact ordering plays a secondary role in the performance outcome.

Program codes, however, can be deliberately written to be difficult to summarize statistically. One example would be codes that deliberately transition between two (or more) distinctly unique stages of execution. In such cases, the statistical profile used by HLS can be arbitrarily extended to accommodate program peculiarities. In the worst case, an extreme profile would cause HLS to symbolically execute every instruction in the entire program. This would be accurate but would not be any faster than using conventional simulation techniques. If we focus on conventional codes such as SPECint95, however, we can investigate program behavior using a statistical summary of the whole-program execution.

It is important that this investigative method not be abused. The HLS approach is not meant to be a replacement for detailed performance analysis. The authors envision that the ideal use for HLS is to set performance goals, to size various machine and code parameters and direct research and design efforts at a high level. Clearly, subsequent designs need to be verified with detailed cycle-by-cycle simulation on actual benchmark codes.

Even when HLS is used to set performance goals, unexpected behavior can arise. For instance, as the L1 instruction cache hit rate and miss penalty is varied, cycle-by-cycle simulation results indicate that the L2 cache hit rate and branch predictor accuracy will also change. Figure 21 depicts how the L2 cache hit rate varies from 94.7% to 99.6% as these two parameters vary. Similarly, Figure 22 depicts how the branch prediction accuracy varies from 86.7% to 90.0%. These two changes are most affected by the instruction cache hit rate; however, slight variations are seen with changes in the cache miss penalty.

In summary, while HLS is a useful tool for quick exploration of the design space, it should be used judiciously.

6 Related work

Several research efforts have approached the problem of architecture simulation by statistical means. Perhaps the effort most similar to HLS is that under taken in [CS98] [JES]. Their approach uses an actual execution trace as the starting point for symbolic execution. The program trace is systematically replaced with statistical parameters and the effects measured. Our approach is similar, but uses an execution rather than trace driven model. Efforts are currently underway to reconcile the two approaches.

A different approach to statistical performance modeling was used in [NS94]. A performance model was constructed based on the interactions between machine and program parallelism. This work extended an earlier model presented in [Jou89]. With these models, benchmarks are analyzed and performance is estimated directly by application of a formula relating program and machine parallelism. This same research group continued their statistical modeling work in [NS97]. A set of Markov models were constructed to represent machine behavior while executing a specific benchmark. These models where then linked together and performance estimated. This approach is more abstract than the direct symbolic execution method presented here, and the authors focus on more ideal machine models. However, extremely accurate performance estimates are achieved.

In some respects the goals of our approach are similar to the goals of the creators of the synthetic Whetstone [CWW76] and Dhrystone [Wei94] benchmarks. However, we attempt to provide an automated approach to generating a synthetic benchmark. Furthermore this automated approach is based upon analysis of real programs and is demonstrably more representative of actual machine performance.

7 Future work

This work can be extended in several directions. In this paper we have focused on current generation microprocessors. We also intend to explore future generation superscalar processors. For instance, Figure 23 compares issue width versus dynamic dependence distance assuming a perfect cache and a basic block size of 100 instructions. The figure shows that as superscalar processors become wider, the DID must increase commensurately. This is not a surprise, but we do note that DID must increase slightly more compared to issue width to achieve comparable performance. Future work will also explore deeper pipeline depths, since

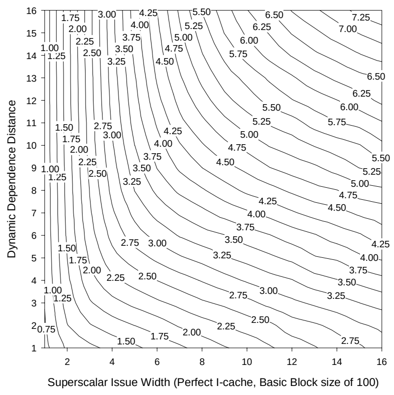


Figure 23: Super-scalar issue width versus dynamic instruction distance assuming a perfect I-cache and basic block size of 100 instructions

as clock speeds increase pipeline depths will continue to increase throughout the processor.

Our current model focuses on aggregate performance for SPECint95. Clearly, the SPEC benchmarks are not the only programs that are of interest to computer architects. We would like to pursue additional benchmarks, such as OLTP applications, and expect that the statistical collection process may have to change to accurately model these data-intensive applications.

Finally, the current simulation technique does not use any information about correlation between load instructions and misses in the caches. Future work will investigate integrating this knowledge from SimpleScalar back into HLS, in the hopes of further improving the correlations at extreme cache behaviors.

8 Conclusion

In this paper, we have presented a new simulation technology. This simulation method combines statistical profiles with symbolic execution. The simulator allows several machine parameters to be varied and their relationship studied in far finer and more accurate detail than previously possible. Furthermore, by using synthetic code streams, we can easily and systematically vary parameters such as basic block size, dynamic instruction distance, branch predictability and cache behavior. We expect this simulation methodology to be beneficial in the study of both conventional and novel architectures. The simulation technique allows the processor designer to peer into the design space, study how parameters interact, and set performance targets for individual components of an architecture. This can help refine ideas more quickly by providing empirical data to evaluate design decisions.

Acknowledgments

Thanks to Jim Smith for pointing us to prior work and suggesting the convergence graph in Figure 4. Additional, thanks to Dan Sorin, David Woods, Shubhendu Mukherjee, Tim Sherwood, Deborah Wallach, Diana Keen, and our anonymous referees. This work is supported in part by an NSF CAREER award to Fred Chong, by NSF grant CCR-9812415, by grants from Mitsubishi and Altera, and by grants from the UC Davis Academic Senate. More information is available at http://arch.cs.ucdavis.edu.

References

[BA97] D. Burger and T. Austin. The SimpleScalar tool set, v2.0. *Comp Arch News*, 25(3), June 1997.

[CRT99] Brad Calder, Glenn Reinman, and Dean M. Tullsen. Selective value prediction. In *International Symposium on Computer Architecture ISCA99*, Atlanta, Georgia, June 1999. ACM.

[CS98] Richard Carl and J.E. Smith. Modeling supersclar processors via statisical simulation. *Performance Analysis and it's Impact on Design (PAID) Workshop*, June 1998.

[CWW76] Curnow, H. J. Wichmann, and B.A. Wichmann. A synthetic benchmark. *The Computer Journal*, 1976.

[GM98] Freddy Gabbay and Avi Mendelson. The effect of instruction fetch bandwidth on value prediction. In *International Symposium on Computer Architecture ISCA98*, Barcelona, Spain, June 1998. ACM.

[JES] Personal communication with J.E. Smith, Daniel Sorin, and David Wood.

[Jou89] Norman P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, December 1989.

[LS96] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture MICRO29*, Paris, France, December 1996. ACM.

[LWY00] Sang-Jeong Lee, Yuan Wang, and Pen-Chung Yew. Decoupled value prediction on trace processors. In *International Symposium on High-Performance Computer Architecture HPCA6*, Toulouse, France, Janurary 2000. ACM.

[MGS99] Tarun Makra, Rajiv Gupta, and Mary Lou Soffa. Value prediction in VLIW machines. In *International Symposium on Computer Architecture ISCA99*, Atlanta, Georgia, June 1999. ACM.

[NS94] Derek B. Noonburg and John P. Shen. Theoretical modeling of superscalar processor performance. In *International Symposium on Microarchitecture MICRO27*. ACM, November 1994.

[NS97] Derek B. Noonburg and John P. Shen. A framework for statistical modeling of superscalar processor performance. In *International Symposium on High-Performance Computer Architecture HPCA3*. ACM, 1997.

[SS97] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *International Symposium on Microarchitecture MICRO30*, Research Triangle Park, North Carolina, December 1997. ACM.

[TS99] Dean M. Tullsen and John S. Seng. Storageless value prediction using prior register values. In *International Symposium on Computer Architecture ISCA99*, Atlanta, Georgia, June 1999. ACM.

[Wei94] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Comm. ACM*, October 1994.

[ZLTI96] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Supercomputing '96*. 1996.