Using Statistical Simulation for Studying Compiler-Microarchitecture Interactions

Lieven Eeckhout† John Cavazos‡

†ELIS Department, Ghent University, Belgium

‡School of Informatics, University of Edinburgh, UK

Abstract

I. Introduction

Simulation is an invaluable tool in the architecture and compiler communities. Cycle-by-cycle simulation allow for tracking individual instructions as they flow through the pipeline during their execution. Given the increasing complexity of contemporary processor designs, and the ever increasing complexity and size of today's applications—which is made possible thanks to the ever increasing performance of today's systems—simulation is a very time consuming process. For example, simulating an industry-standard SPEC CPU2000 benchmark takes on the order of days and even several weeks on an optimized architectural simulator, and this for a single benchmark with one input compiled with a single compiler optimization setting for a single microarchitectural configuration. Obviously, detailed full-benchmark simulation is impossible for exploring large compiler and/or microarchitecture design spaces.

Recently, researchers have proposed statistical simulation [7], [8], [9], [16], [17] as a solution to the problem of long architectural simulation times. Statistical simulation profiles an application and computes a number of important program characteristics such as instruction mix, inter-operation dependencies, cache behavior, branch behavior, etc. This statistical profile is then used as input for generating a synthetic trace that is subsequently fed into a statistical simulator. The important advantage of statistical simulation over detailed simulation is that the synthetic trace is extremely short in comparison to real application dynamic instruction counts. One million synthetic instructions is typically enough for making accurate performance and energy consumption estimates. Previous work has shown that statistical simulation is especially valuable when it comes to estimating relative performance variations across microarchitectures. This property in conjunction with the fast simulation property make statistical simulation a powerful tool for quickly exploring huge microarchitecture design spaces. Statistical simulation can thus be used to quickly identify a region of interest which can then be further explored through detailed and thus slower simulation runs.

To evaluate a new microarchitecture properly, it is necessary to have programs optimized for that microarchitecture. However, manually tuning a compiler to get the best performance is a time-consuming task and it is therefore not possible to have a hand-tuned compiler for all the configurations an architect would like to evaluate. Therefore, it would be desirable to quickly and automatically tune an optimizing compiler to any given architecture.

In recent years, iterative compilation has shown great promise in automatically finding good transformation sequences for particular programs [6], [10], [14]. However, iterative compilation can be costly since testing thousands of different optimization configurations is typically required to obtain the best performance for a program. Recently, a new technique [1] has been proposed that can greatly diminish the cost of iterative compilation. However, evaluating even a small number of compiler configurations using detailed simulation could take many days or even weeks for just one program, greatly reducing the size of the architecture design space an architect can explore.

This paper describes a feasibility study for using statistical simulation instead of full detailed simulation in conjunction with iterative compilation to explore a large spaces of microarchitecture and/or compiler design spaces. Using a number of SPEC CPU2000 benchmarks and a number of compiler optimization flags, we show that statistical simulation is capable of accurately tracking the performance/energy differences induced by different compilation settings. For example, we obtain correlation coefficients between detailed simulation results and statistical simulation results for estimating performance between 0.84 and 0.99. These preliminary results are very encouraging for persuing future research into the applicability of statistical simulation for studying compiler-microarchitecture interactions. We envision and discuss a number of potential applications for this approach in this paper, namely (i) rapid compiler development using statistical simulation instead of detailed simulation in case no real hardware is available yet, (ii) microarchitecture/compiler design space co-exploration, (iii) iterative compilation where the best performing compiler optimization flags are determined for a given application and a given microprocessor, and (iv) quickly exploring potential compiler optimizations.

This paper is organized as follows. We first discuss the statistical simulation method. We subsequently detail our experimental setup in section III. In section IV we then evaluate the accuracy of statistical

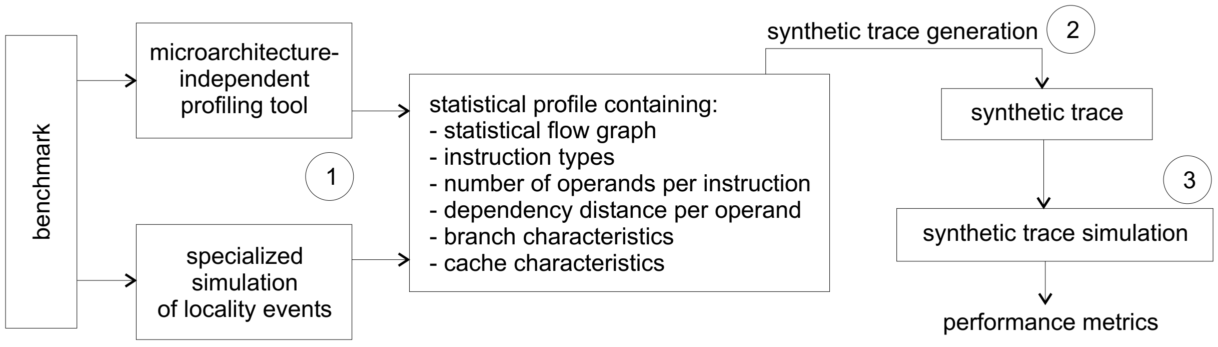
benchmark
microarchitecture-independent profiling tool
specialized simulation of locality events
1
statistical profile containing:
- statistical flow graph
- instruction types
- number of operands per instruction
- dependency distance per operand
- branch characteristics
- cache characteristics
synthetic trace generation
2
synthetic trace
3
synthetic trace simulation
performance metrics

Fig. 1. Statistical simulation: general framework.

simulation for tracking performance and energy differences between different compiler optimizations. Section V then discusses potential applications for using statistical simulation when studying compiler/microarchitecture interactions. We finally conclude in section VI and discuss future work.

II. Statistical simulation

Statistical simulation [7], [8], [9], [16], [17] consists of three steps as shown in Figure 1. We first measure a *statistical profile* which is a collection of important program execution characteristics such as instruction mix, inter-instruction dependencies, branch miss behavior, cache miss behavior, etc. Subsequently, this statistical profile is used to generate a *synthetic trace* consisting on the order of a million instructions. In the final step, this synthetic trace is simulated on a statistical simulator which yields performance metrics such as IPC and energy consumption. We now discuss these three steps in more detail.

A. Statistical profiling

In statistical profiling we make a distinction between microarchitecture-dependent and microarchitecture-independent characteristics. The microarchitecture-independent characteristics can be used across different microarchitectures during design space exploration. The microarchitecture-dependent characteristics on the other hand are particular to specific (subparts of) a microarchitecture.

1) Statistical flow graph: The key structure in the statistical profile is the *statistical flow graph (SFG)* [7] which represents the control flow in a statistical manner.

In an SFG, each node represents a particular basic block and its basic block history, *i.e.*, the predecessor basic blocks executed prior to the given basic block. The order of the SFG corresponds to the number of blocks in the basic block history leading up to a basic block in each node of the SFG. Consider, for example, the following trace of basic blocks 'ABBAABAABBA'. A 4th-order SFG would make a distinction between basic block 'A' given its basic block history 'ABBA', 'BAAB', 'AABA', 'AABB'; this SFG will thus contain the following nodes: 'A|ABBA', 'A|BAAB', 'A|AABA' and 'A|AABB'. The edges in the SFG interconnecting the nodes represent transition probabilities between the nodes. The idea behind the SFG is to model program characteristics that are correlated with path behavior. Different statistics are therefore computed for different basic block histories. For example, in a 4th-order SFG, cache miss statistics for basic block 'A' would be different when its basic block history is 'ABBA' and when its basic block history is 'BAAB'.

Such cases can be modeled in a 4th-order SFG. On the other hand, in case a correlation between program characteristics spans a number of basic blocks that is larger than the SFG's order, it will be impossible to model such correlations within the SFG, unless the order of the SFG is increased.

2) Microarchitecture-independent characteristics: The first microarchitecture-independent characteristic is the *instruction mix*. We classify the instruction types into 12 classes according to their semantics: load, store, integer conditional branch, floating-point conditional branch, indirect branch, integer alu, integer multiply, integer divide, floating-point alu, floating-point multiply, floating-point divide and floating-point square root. And for each instruction we record the number of source operands. Note that some instruction types, although classified within the same instruction class, may have a different number of source operands.

For each operand we also record the *dependency distance* which is the number of dynamically executed instructions between the production of a register value (register write) and the consumption of it (register read). We only consider read-after-write (RAW) dependencies since our focus is on out-of-order architectures in which write-after-write (WAW) and write-after-read (WAR) dependencies are dynamically removed through register renaming as long as enough physical registers are available. Note that recording the dependency distance requires storing a distribution since multiple dynamic versions of the same static instruction, *e.g.*, due to different basic block histories in the SFG, could result in multiple dependency distances.

3) Microarchitecture-dependent characteristics: In addition to these microarchitecture-independent characteristics we also measure a number of microarchitecture-dependent characteristics that are related to locality events. We model locality events in a microarchitecture-dependent way because they are hard to model using microarchitecture-independent metrics. We therefore take a pragmatic approach and collect cache miss and branch miss information for particular cache configurations and branch predictors.

For the *branch statistics* we measure (i) the probability for a taken branch, (ii) the probability for a fetch redirection (target misprediction in conjunction with a correct taken/not-taken prediction for conditional branches), and (iii) the probability for a branch misprediction.

The *cache statistics* consist of the following six probabilities: (i) the L1 I-cache miss rate, (ii) the L2 cache miss rate due to instructions only[1], (iii) the L1 D-cache miss rate, (iv) the L2 cache miss rate due to data accesses only, (v) the I-TLB miss rate and (vi) the D-TLB miss rate.

We want to emphasize that all the program characteristics discussed above, both the microarchitecture-dependent and -independent characteristics, are measured in the context of an SFG. Thus, separate statistics are kept for different basic block histories or execution paths.

B. Synthetic trace generation

The second step in the statistical simulation methodology is to generate a synthetic trace from the statistical profile. The synthetic trace generator takes as input the statistical profile and outputs a synthetic trace that is fed into a statistical simulator. Synthetic trace generation uses random number generation for generating a number in [0,1]; this random number is then used with the cumulative distribution function to determine a specific value for a given program characteristic. The synthetic trace is a linear sequence of synthetic instructions. Each instruction has an instruction type, a number of source operands, an inter-instruction dependency for each source operand (which describes the producer for the given source operand), I-cache miss info, D-cache miss info (in case of a load), and branch miss info (in case of a branch). The locality miss events are just labels in the synthetic trace describing whether the load is an L1 D-cache hit, L2 hit or L2 miss and whether the load generates a TLB miss. Similar labels are assigned for the I-cache and branch miss events.

C. Synthetic trace simulation

Simulating the synthetic trace is fairly straightforward. In fact, the synthetic trace simulator itself is very simple as it does not need to model branch predictors nor cache hierarchies—misses are just tagged to the synthetically generated instructions; also, all the ISA's instruction types are collapsed in a limited number of instruction types. The important benefit of statistical simulation is that the synthetic traces are fairly short. The performance metrics quickly converge to a steady-state value when simulating a

[1]We assume a unified L2 cache. However, we make a distinction between L2 cache misses due to instructions and due to data.

benchmark	input
vpr	lgred-route
crafty	lgred
gap	lgred
gzip	lgred-graphic
bzip2	lgred-source
vortex	lgred
twolf	lgred
mcf	lgred
parser	lgred

TABLE I
THE SPEC CPU2000 INTEGER BENCHMARKS USED IN THIS STUDY ALONG WITH THEIR INPUTS.

synthetic trace. As such, synthetic traces containing a million of instructions are sufficient for obtaining stable performance estimates.

Note that, next to estimating performance, statistical simulation can also be used for estimating energy consumption. The synthetic trace simulator needs to be enhanced with an architectural-level energy estimation tool. An architectural-level energy simulation tool basically measures unit-level activity counters and multiplies those activity counters with unit-level power consumption numbers based on the size of the units and their technological constraints. As such, incorporating architectural-level energy estimation into a synthetic trace simulator is easily done.

III. EXPERIMENTAL SETUP

We use a number of SPEC CPU2000 benchmarks in our experimental setup. The inputs used for each of these benchmarks are the large (lgred) inputs from the MinneSPEC reduced input sets [13]. The benchmarks and inputs we used are shown in Table I. The dynamic instruction count for each of these benchmarks varies between 500M and 4B instructions. We choose these reduced input sets instead of the reference inputs provided by SPEC because of the excessive simulation times for some of these benchmarks when run with their reference inputs. Moreover, we had to simulate each benchmark multiple times for each of the different compiler optimizations that we experimented with. The binaries were compiled using the Alpha `cc` compiler V6.3-025 on a Compaq Tru64 UNIX platform. All benchmarks were statically linked using the `-non_shared` flag. The nine compiler optimization flags that we consider in this paper are shown in Table II. We limited ourselves to only nine compiler optimization flags because of the excessive simulation times; for each of the optimizations we had to run a detailed simulation next to a statistical simulation. Note that this is exactly the problem we are tackling with statistical simulation.

flags	description
`-O1`	Local optimizations and recognition of common subexpressions. Global optimizations, including code motion, strength reduction and test replacement, split lifetime analysis, and code scheduling.
`-O3`	Inline expansion of static and global procedures. Additional global optimizations that improve speed (at the cost of extra code size), such as integer multiplication and division expansion (using shifts), loop unrolling, and code replication to eliminate branches.
`-O3 -noinline`	By default, `-O3` provides function inlining. The `-noinline` option disables this optimization.
`-O3 -unroll`	By default, `-O3` includes loop unrolling. The `-unroll` option lets the compiler decide whether loop unrolling is appropriate or not.
`-O3 -om`	Additional optimizations after linking such as removing NOP instructions and reallocating common symbols.
`-O4`	Software pipelining using dependency analysis, vectorization of some loops on 8-bit and 16-bit data (char and short), and insertion of NOP instructions to improve scheduling.
`-O4 -noinline`	By default, `-O4` provides function inlining. The `-noinline` option disables this optimization.
`-O4 -unroll`	By default, `-O4` includes loop unrolling. The `-unroll` option lets the compiler decide whether loop unrolling is appropriate or not.
`-O4 -om`	Additional optimizations after linking such as removing NOP instructions and reallocating common symbols.

TABLE II
THE OPTIMIZATION CONFIGURATIONS FOR THE ALPHA `cc` COMPILER THAT ARE USED IN THIS PAPER.

The simulation results that we present in this paper are obtained using SimpleScalar/Alpha v3.0 [3]. The baseline processor model that we assume here is an aggressive out-of-order microarchitecture, see Table III. The energy numbers reported here were obtained using Wattch v1.02 [5].

IV. EVALUATION

We now evaluate the ability of statistical simulation to estimate execution time, energy consumption and energy-delay product across binaries that are optimized using different compilation flags. The primary question we want to address is: “Is statistical simulation accurate enough to track performance and energy differences between application binaries compiled with different optimizations?”. Estimating the total execution time is done by multiplying CPI (cycles per committed instruction) with the total dynamic instruction count. Estimating the total energy consumption is done by multiplying EPI (energy consumption per committed instruction) with the total dynamic instruction count. Finally, the energy-delay product (EDP) [4] is a fused metric that combines execution time with energy consumption: $EDP = EPI \times CPI$. EDP is a well known energy-efficiency metric for microprocessors. The lower the EDP, the more energy-efficient the design is.

Before evaluating whether statistical simulation is indeed capable of tracking compilation variations we first discuss the prediction accuracy of statistical simulation. This is done in Figure 2 where the average

instruction cache	32KB, 2-way set-associative, 32-byte block, 2 cycles access latency
data cache	64KB, 4-way set-associative, 32-byte block, 2 cycles access latency
unified L2 cache	4MB, 4-way set-associative, 64-byte block, 20 cycles access latency
I-TLB and D-TLB	128-entry 8-way set-associative with 4KB pages
memory	150 cycle round trip access
branch predictor	8K-entry hybrid predictor selecting between an 8K-entry bimodal predictor and a two-level (8K x 8K) local branch predictor xor-ing the local history with the branch's PC, 512-entry 4-way set-associative BTB and 64-entry RAS
speculative update	at dispatch time
branch misprediction penalty	14 cycles
IFQ	32-entry instruction fetch queue
RUU and LSQ	128 entries and 32 entries, respectively
processor width	8 issue width, 8 decode width (fetch speed = 2), 8 commit width
functional units	8 integer ALUs, 4 load/store units, 2 fp adders, 2 integer and 2 fp mult/div units

TABLE III

BASELINE PROCESSOR SIMULATION MODEL.

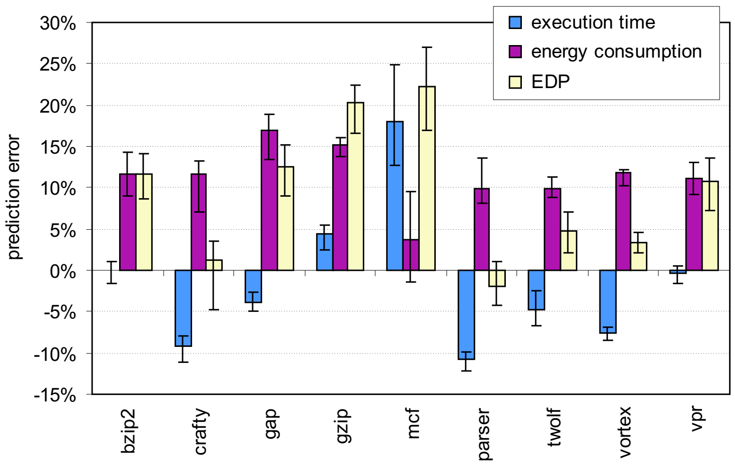

Fig. 2. Prediction errors when estimating execution time, energy consumption and EDP across the different benchmarks. The bars respresent the average error, and the error bars represent the minimum and maximum errors observed across the nine compiler optimizations.

prediction error is shown over the nine optimization configurations that we consider in this paper. This is done for estimating execution time, energy consumption and EDP. The solid bars represent the average error and the error bars show the minimum and maximum error observed over those nine optimization configurations. We observe that the prediction error is fairly small. For estimating performance, the average error across the benchmarks is only 6.5%; and the largest error is for mcf (18%). For estimating energy consumption, the error is higher (11.3% on average) and seems to be a consistent overestimation. The reason is that the architecture-level energy estimation tool uses data values for computing the number

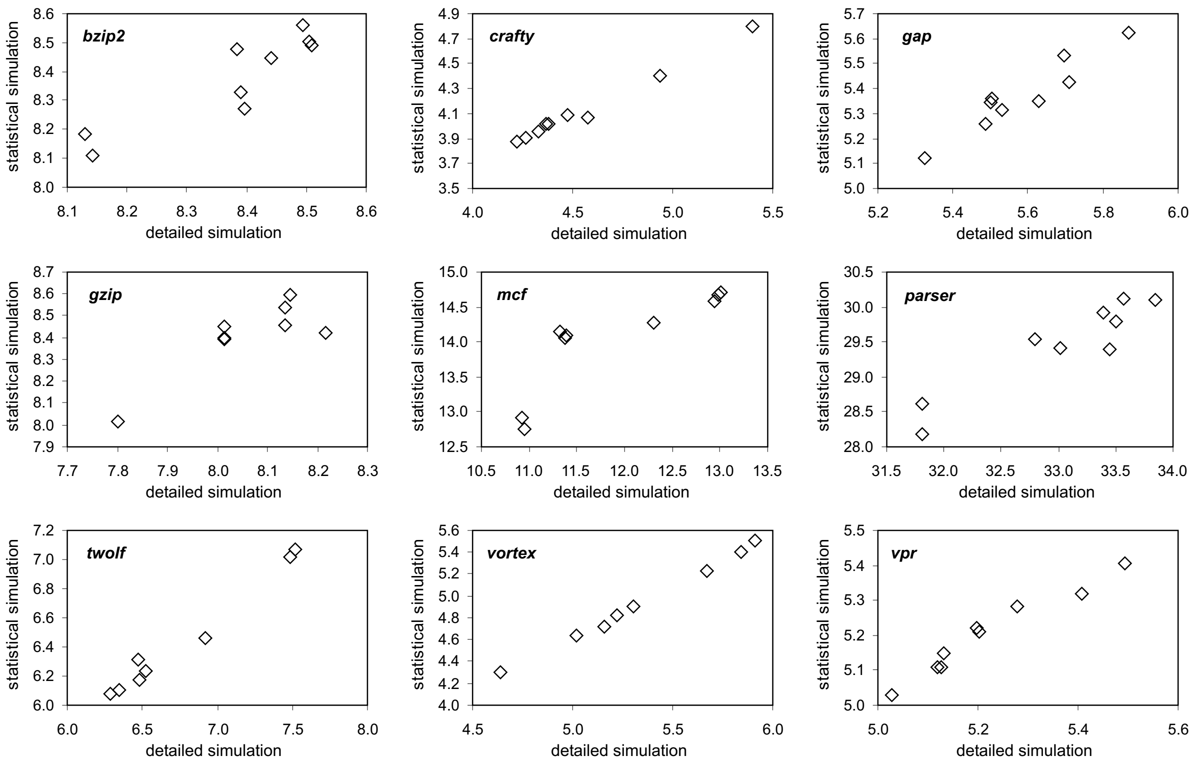

Fig. 3. Estimating total execution time using statistical simulation (Y axis) versus detailed simulation (X axis).

of bit switches per cycle. These bit switches are accounted for during detailed simulation, but they are not during statistical simulation. Statistical simulation assumes that the probability for a bit switch equals 0.5, which seems to lead to a consistent overestimation. Estimating EDP, which is the cross-product of CPI and EPI, is done with an average error of 9.8%. However, more importantly than the average errors shown in Figure 2 is that the deviation from this average error is fairly small. Indeed, the error bars are well centered around the solid bars, *i.e.*, the difference between the minimum and maximum error is fairly small compared to the average error. This suggests that although there is a consistent error when using statistical simulation, statistical simulation might be able to accurately estimate relative performance and energy differences.

We now discuss the ability of statistical simulation to measure relative performance and energy differences between different compiler options. Figures 3, 4 and 5 show scatter plots estimating execution time, energy consumption and the energy-delay product, respectively. The vertical axes in all the graphs show the respective metrics for statistical simulation; the horizontal axes show the respective metrics for detailed simulation. For each of the benchmarks there is a separate graph, and per benchmark we

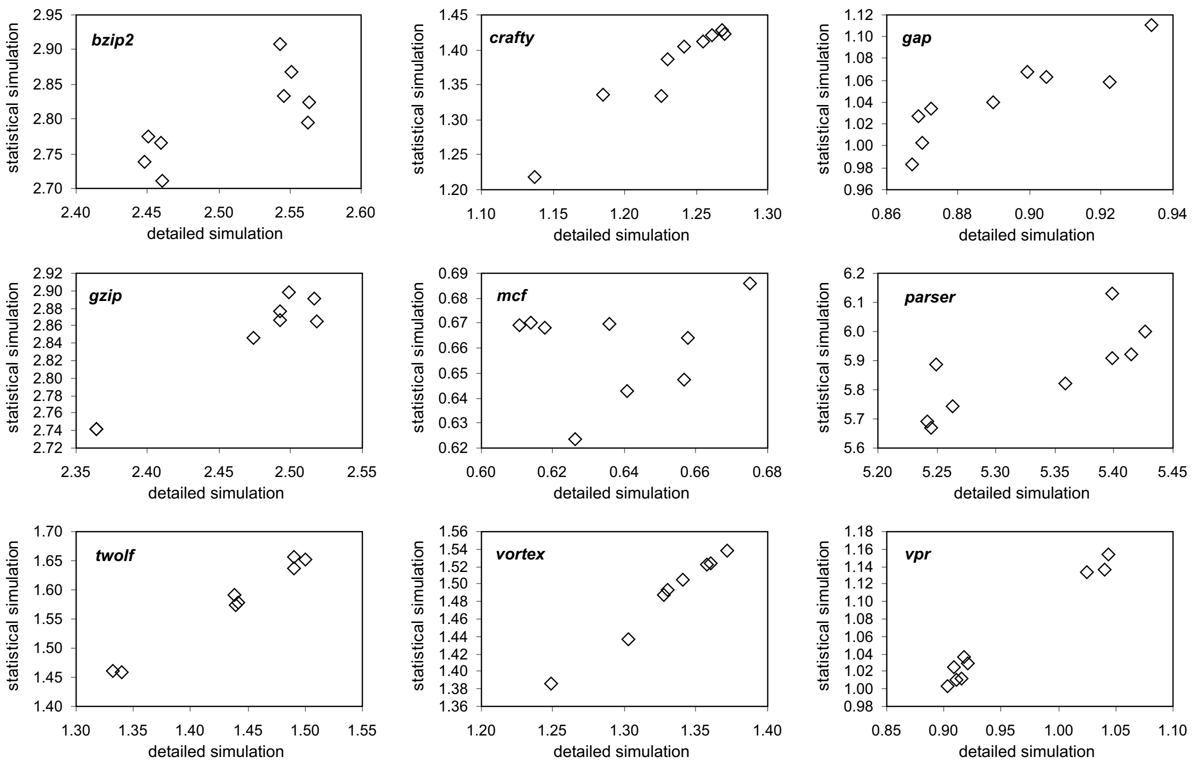

Fig. 4. Estimating total energy consumption using statistical simulation (Y axis) versus detailed simulation (X axis).

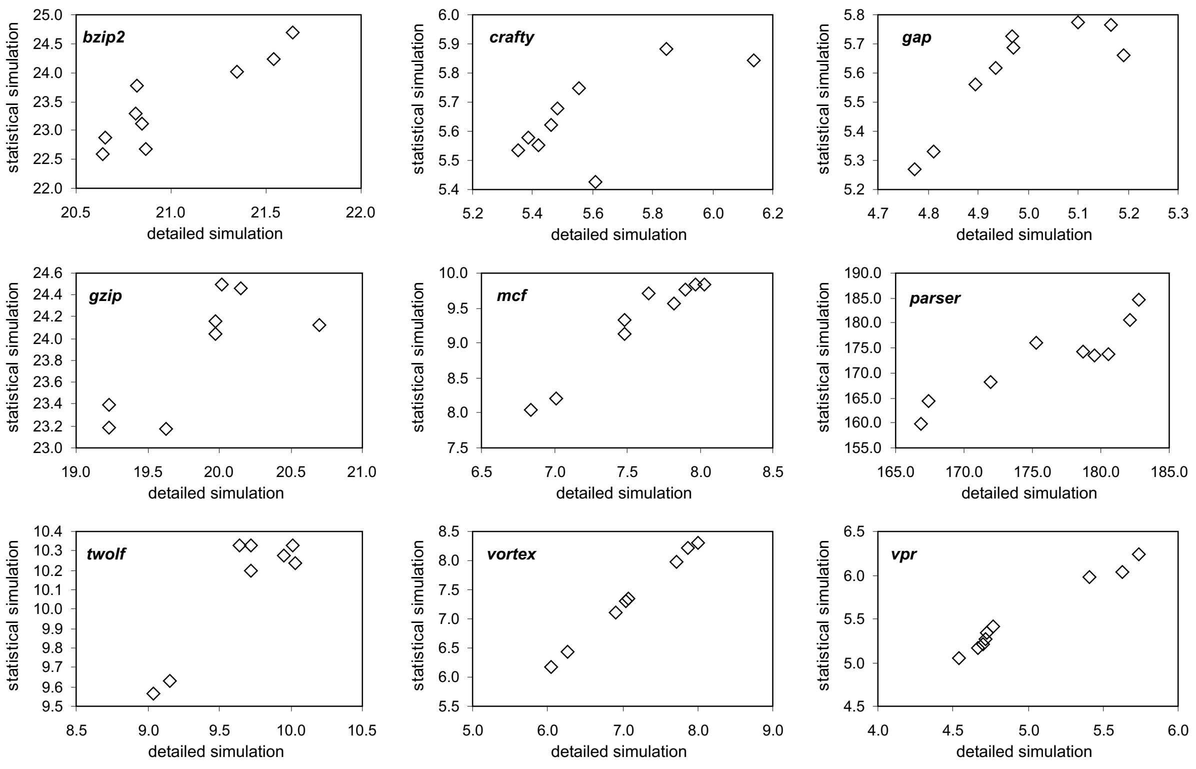

Fig. 5. Estimating the energy-delay product (EDP) using statistical simulation (Y axis) versus detailed simulation (X axis).

benchmark	R for time	R for energy	R for EDP
bzip2	0.902	0.761	0.902
crafty	0.993	0.963	0.708
gap	0.947	0.889	0.826
gzip	0.842	0.973	0.761
mcf	0.842	0.147	0.970
parser	0.940	0.781	0.917
twolf	0.989	0.994	0.905
vortex	0.998	0.991	0.999
vpr	0.976	0.990	0.989

TABLE IV
THE CORRELATION COEFFICIENTS FOR ESTIMATING EXECUTION TIME, ENERGY AND EDP BETWEEN STATISTICAL SIMULATION AND DETAILED SIMULATION.

consider all the compiler optimization configurations, where each configuration is represented as a point in the scatter plot. Obviously, the more the dots appear on the diagonal of these graphs, the better. Thus, a higher correlation between the metrics obtained through statistical simulation and the metrics obtained through detailed simulation is desired. It can visually be observed that there is a strong correlation between the metrics estimated by statistical simulation versus the metrics obtained through detailed simulation. Table IV quantifies these correlations. We observe that typically very high correlation coefficients are obtained across all three metrics (time, energy and EDP). In most cases, the correlation coefficients are above 0.9. In some cases, the correlation coefficient is around 0.7 and 0.8, which is still a very good correlation. There is however one exception, namely when estimating the energy consumption for mcf, in which case the correlation coefficient is only 0.147. Overall, we can conclude that statistical simulation is indeed a very accurate technique for tracking performance and energy differences between applications compiled with different optimization configurations.

V. APPLICATIONS

Now that we have shown that statistical simulation is indeed accurate enough for tracking performance differences across different compiler optimizations on out-of-order microarchitectures, we now discuss a number of potential applications.

A. Rapid compiler development

One possible application is speeding up the compiler development process. In cases where there is no real hardware available, simulation is the only way to estimate the performance benefit for a given compiler optimization. In such scenarios, statistical simulation could be used to speed up the compiler tuning process.

Recall that in statistical simulation, a profiling phase is first performed to compute the program's statistical profile. This step needs to be done for every potential compiler optimization that is to be evaluated.

Fortunately, statistically profiling an application is much faster than running a detailed cycle-by-cycle simulation. Profiling is typically an order of magnitude faster than detailed simulation when profiling is done through functional simulation [3]. If profiling is performed using static instrumentation (using for example ATOM [18]) or dynamic instrumentation (using for example Pin [15]) additional speedups can be expected. Generating and simulating the synthetic trace is also done fast given the very short synthetic traces. As such, statistical simulation can yield a compiler development speedup by at least one order of magnitude over detailed simulation. Nevertheless, if statistical profiling was a bottleneck in using statistical simulation for compiler development, sampled profiling [2] could be employed to reduce the overhead of collecting the statistical profile.

There are at least two scenarios where compiler development relies on simulation rather than on native hardware execution. First, when designing embedded processors, it is unlikely that hardware is available at the time the compiler is being developed. The reason is the short time-to-market constraints on the embedded market. Nevertheless, it is important that the compiler be tuned for the processor under development so that the entire product (which includes both hardware and software) gets to the market quickly. Second, compiler developers would like to determine the optimal compilation flags for a given microprocessor that is still under development. For example for the SPEC CPU benchmark suite, the SPEC peak rate refers to the best achievable performance for the given microprocessor and benchmark. When determining the SPEC peak rate, any combination of compiler optimization flags is allowed to be used for compiling the benchmark. Given the fact that a microprocessor company wants to know (or even announce) the highest possible SPEC peak rate for their upcoming product as soon as possible, it is important to be able to determine the best possible compiler flags combination efficiently before the real hardware is available. Statistical simulation may help in that pursuit.

B. Design space exploration

When designing an application-specific embedded processor it is important to optimize the hardware and the software concurrently. This is refered to as hardware/software co-evolution, *i.e.*, the hardware and software design space is explored concurrently. The reason for exploring both design spaces concurrently

	worst option	detailed simulation		statistical simulation	
benchmark		best option	improv	best option	improv
bzip2	`-O4 -unroll`	`-O3 -om`	4.5%	`-O4 -om`	4.3%
crafty	`-O3 -om`	`-O3`	21.8%	`-O3`	21.8%
gap	`-O4 -unroll`	`-O1`	9.3%	`-O1`	9.3%
gzip	`-O1`	`-O4 -om`	5.1%	`-O4 -om`	5.1%
mcf	`-O4 -unroll`	`-O4 -om`	16.0%	`-O3`	15.7%
parser	`-O4 -inline`	`-O3 -om`	6.0%	`-O3 -om`	6.0%
twolf	`-O4 -inline`	`-O3 -om`	16.4%	`-O3 -om`	16.4%
vortex	`-O4 -unroll`	`-O4 -om`	21.5%	`-O4 -om`	21.5%
vpr	`-O4 -inline`	`-O4 -om`	8.5%	`-O4 -om`	8.5%

TABLE V
OPTIMAL COMPILER OPTIONS FOR BOTH DETAILED AND STATISTICAL SIMULATION. THE IMPROVEMENTS OVER THE WORST PERFORMING COMPILER OPTION ARE ALSO SHOWN.

is that there are important interactions between the compiler's output and microarchitecture that is being designed. Obviously, during hardware/software design space co-exploration there is no hardware available. Statistical simulation could help in exploring the design space more effectively and efficiently. Previous work has shown that statistical simulation achieves a high relative accuracy making it a useful tool in the early stages of microarchitecture design space exploration [7]. The results presented in this paper, show that statistical simulation can also be used to evaluate compiler optimizations. As such, the use of statistical simulation for exploring both the microarchitecture and compiler design spaces looks like a promising and efficient hardware/software co-exploration tool.

C. *Iterative compilation*

The idea of iterative compilation is to apply search techniques for tuning compiler optimization flags and finding a good phase ordering for a given program on a given hardware platform. Previous work has demonstrated that iterative compilation can yield significant performance speedups for iteratively compiled benchmarks [6], [10], [14], [1]. Table V shows the optimal compiler options for the various benchmarks when optimizing for performance. Also, the improvement in execution time is shown compared to the worst performing compiler option in our data set. Although this is a very small example compared to what is being done in iterative compilation where hundreds or even thousands of compiler options are evaluated, this table clearly shows that the optimal compiler option varies across benchmarks. Also, this table shows that statistical simulation is accurate in determining what compiler option results in near-optimal performance. In all but a few cases, statistical simulation identifies the optimal performing compiler option; in a few cases statistical simulation gets in the neighbourhood of the optimal compiler option. Statistical simulation can thus help in determining well performing compiler options. These near-optimal compiler options can then be further evaluated using more slowly detailed simulation runs.

We see at least two applications for statistical simulation in the context of iterative compilation. First, very much as discussed above, in case hardware is not yet available, compiler developers (and application developers) will have to rely on simulation. Speeding up the iterative compilation process is a well recognized problem which is a subject of active research [11]. Again, statistical simulation could also be a viable option. Second, statistical simulation can be used to obtain a preliminary idea of how much the program can be sped up from a given compiler optimization. If the potential performance improvement is limited when optimizing a given program characteristic, it might not be worth the effort to further optimize the compiler settings. Fursin *et al.* [12] proposed an algorithm that estimates the performance to be expected from perfect memory behavior. Their approach estimates this speedup by modifying a program to convert all loads and store to access a local array instead. Unfortunately, their approach only applies to caches and cannot be extended to other architecture features, *e.g.*, branch predictors. In contrast, statistical simulation can be used to estimate perfect performance of several architecture features. If one wants to know the estimated performance under perfect cache behavior, or if one wants to know the estimated speedup for a perfect branch predictor, statistical simulation can be used as an efficient and accurate tool for estimating these upperlimits in performance.

A special form of iterative compilation is called Optimization Space Exploration (OSE) proposed by Triantafyllis *et al.* [19]. OSE could be viewed as a special class of iterative compilation that is applicable to general-purpose computing environments. As briefly mentioned above, iterative compilation typically requires a very large number of optimization flags and optimization orders to be explored. Evaluating all those optimization configurations can take a substantial amount of time. Although this is acceptable for application-specific embedded processor design where compilation time is not a major issue, it is not acceptable for general-purpose computing. In general-purpose computing, compilation time is an important constraint. Therefore, Triantafyllis *et al.* [19] proposed OSE for general-purpose compilers which builds a tree of well performing optimization flag combinations and orderings. Based on an analytical model they decide what path of the tree to take. Statistical simulation could be used alternatively instead of analytical models for evaluating the various optimization configurations. We believe that statistical simulation would be more accurate than an analytical model, however, this comes at the cost of having to profile the application for computing the statistical profile to drive the statistical simulation.

D. Exploring potential compiler optimizations

Another important application for statistical simulation is to study *what-if* questions. Statistical simulation allows for varying the different program characteristics that are used as input to the synthetic trace generation. And all of these can be varied independently of each other. There are a number of important applications that are enabled by doing this. First, a benchmark suite rarely covers the entire workload space, *i.e.*, *weak spots* exist in the workload space that are not covered by the benchmark suite. Addressing these weak spots can be done in two ways. One way could be to try to find real benchmarks to fill these holes, which is hard to do. Alternatively, one could generate synthetic traces that estimate the performance in those holes. A second application enabled by varying program characteristics that we are specifically interested in here is to quickly investigate performance benefits of potential compiler optimizations. For example, if a compiler optimization would target reducing the branch misprediction penalty by focusing on load misses that feed mispredicted branches, then we could easily estimate the upperbound potential performance benefit achievable for such an optimization. When generating the synthetic trace we can easily make sure that all the load misses that lead to a branch misprediction are cache hits instead of cache misses. Thus, compiler developers could use statistical simulation to do a preliminary feasibility study on a potential compiler optimization on a set of benchmarks before hardware is available without having to rely on expensive detailed simulations.

VI. Conclusion and future work

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *CGO-4: The Fourth Annual International Symposium on Code Generation and Optimization*, 2006.

[2] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'01)*, pages 168–179, June 2001.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[4] D. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, November/December 2000.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pages 83–94, June 2000.

[6] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 69–77, 2005.

[7] L. Eeckhout, R. H. Bell Jr., B. Stougie, K. De Bosschere, and L. K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA-31)*, pages 350–361, June 2004.

[8] L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pages 25–34, Sept. 2001.

[9] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, Sept/Oct 2003.

[10] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 78–86, 2005.

[11] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 2005 HiPEAC International Conference on High Performance Embedded Architectures and Compilers*, Nov. 2005.

[12] G. Fursin, M. O'Boyle, O. Temam, and G. Watts. Fast and accurate method for determining a lower bound on execution time. *Concurrency Practica and Experience*, 16(2-3):271–292, 2004.

[13] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(2):10–13, June 2002.

[14] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 171–182, 2004.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'05)*, pages 190–200, June 2005.

[16] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pages 15–24, Sept. 2001.

[17] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*, pages 71–82, June 2000.

[18] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, Mar. 1994.

[19] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. *Journal of Instruction-level Parallelism*, Jan. 2005. Accessible at `http://www.jilp.org/vol7`.