EXPERT: Expedited Simulation Exploiting Program Behavior Repetition

Wei Liu and Michael C. Huang
Department of Electrical & Computer Engineering
University of Rochester
{weliu, michael.huang}@ece.rochester.edu

ABSTRACT

Studying program behavior is a central component in architectural designs. In this paper, we study and exploit one aspect of program behavior, the behavior repetition, to expedite simulation.

Detailed architectural simulation can be long and computationally expensive. Various alternatives are commonly used to simulate a much smaller instruction stream to evaluate design choices: using a reduced input set or simulating only a small window of the instruction stream. In this paper, we propose to reduce the amount of detailed simulation by avoiding simulating repeated code sections that demonstrate stable behavior. By characterizing program behavior repetition and use the information to select a subset of instructions for detailed simulation, we can significantly speed up the process without affecting the accuracy. In most cases, simulation time of full-length SPEC CPU2000 benchmarks is reduced from hundreds of hours to a few hours. The average error incurred is only about 1% or less for a range of metrics.

Categories and Subject Descriptors

I.6.7 [**Computing methodologies**]: Simulation and modeling—*Simulation support systems*; C.4 [**Computer systems organization**]: Performance of systems—*Measurement techniques*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

behavior repetition, statistical sampling, fast simulation

1. INTRODUCTION

Studying and understanding program behavior is a very important step in computer system design. In the microprocessor or system design process, an architectural simulator is an indispensable tool to evaluate different design choices at a relatively high level. Unfortunately, even such a high-level evaluation can be very time-consuming, sometimes agonizingly so. A very popular simulator tool set in the architecture community is SimpleScalar [1]. A typical detailed simulation of an out-of-order microprocessor using SimpleScalar runs about a few hundred thousand instructions a second on a high-end workstation. Compared to the hardware that is being simulated, this represents a slowdown of approximately four orders of magnitude. Minutes of native execution take days to simulate. Furthermore, architectural studies generally require exploring a design space of a non-trivial size and evaluating a broad range of applications to fully understand the overall effect. Combining all these factors, detailed timing simulation of large programs to completion is usually impractical. A common practice is to use a small window of dynamic instructions, or smaller input sets, or a combination of both to drastically reduce simulation time. Whether the behavior of the reduced version fully represents that of the original program remains to be seen, especially as new benchmarks continue to emerge.

In this paper, we address the problem by exploiting the repetition of program behavior within a single application. Computer programs' behavior is repetitive by nature: different iterations of a loop or different invocations of a subroutine can bear great similarities from an architecture perspective. Understanding how program behavior repeats is important as it allows efficient and accurate study of the behavior of large scale programs, which is an important step toward hardware and software optimizations. We propose a methodology to characterize the behavior *repetition* of dynamic instances of static code sections and, based on that information, only perform detailed simulation on selected sample instances. The more "stable" the behavior, the less the amount of sampling is required.

Skipping portions of the code without simulation may seem inherently inaccurate, but given limited computing resources, this sampling approach can be much more accurate than simulating only a small window of dynamic instructions. Indeed, while behavior repetition-based sampling approximate the statistics of a skipped unit using those from a unit known to be similar, the typical simulation window approach simply ignores the issue of representativeness.

One unique aspect of our approach is the use of instances of static code sections as a natural granularity of program behavior repetition. This approach is intuitive, offering further confidence in its viability in addition to what quantitative analyses suggest. It also avoids the need of a haphazardly selected fixed-size granularity that is unlikely to coincide with the periodicity of different applications. Moreover, studying program behavior repetition of static code sections is also useful in other optimizations in general. We call our simulation strategy *Expedited simulation eXploiting Program bEhavior RepeTition*, or *EXPERT*.

Our analyses show that dynamic instances of the same subrou-

ICS'04, June 26–July 1, 2004, Malo, France.

tine and iterations of the same loop often exhibit great similarities. This is demonstrated by the low coefficient of variation for a range of architectural metrics such as CPI (cycles per instruction) and the cache miss rate. Thus, despite the fact that we only select a small subset of the program, we still obtain high accuracy on all the measurements. Our experiments show that by applying the *EXPERT* strategy, simulation time of full-length SPEC 2000 applications can be reduced from weeks to hours or even minutes and yet the average error in CPI is only about 0.9%. The maximum error of a range of metrics is less than 3.8%, compared to full-blown detailed simulations.

The rest of this paper is organized as follows: Section 2 shows some observations, Section 3 discusses the methodology to exploit behavior repetition at code section level to expedite simulation, Section 4 shows the experimental setup, Section 5 presents the evaluation and analysis of our system, Section 6 discusses related work, and finally, Section 7 concludes.

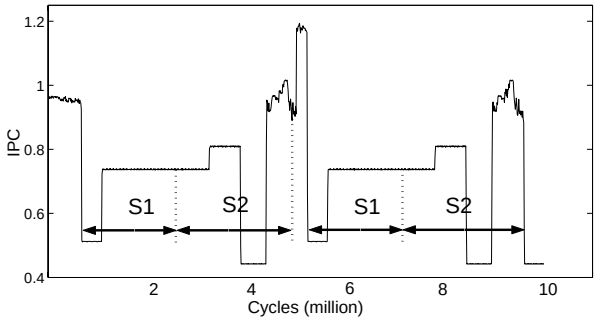
2. BEHAVIOR REPETITION

Computer programs rely heavily on repetition to perform any significant operations. The number of static instructions for SPEC CPU2000 benchmark binaries compiled for our simulator is no more than a few hundred thousands, while a typical run consists of tens or hundreds of billions of dynamic instructions. Repeated execution of loop iterations or subroutine invocations are common examples of this repetition. Intuitively, code has a large bearing on the actual behavior[1]. Repeated execution of the same code could yield very similar behavior. Figure 1 shows this repetition visually. We can see the repeating patterns in the IPC traces of the two subroutines and a few iterations of a loop. The repeating patterns suggest repeated behavior. In general, this repetition can be exploited to predict future code behavior based on observed history.

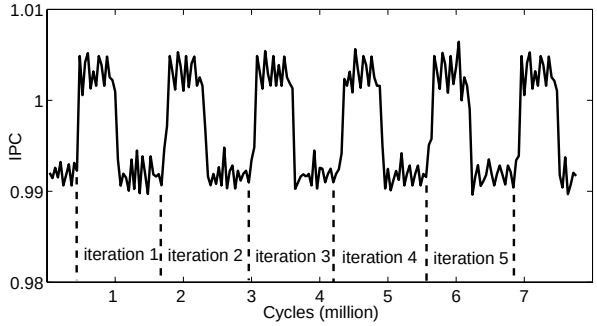
To characterize the behavior repetition of repeated executions of a static code section, we measure the standard deviation of a range of statistics: CPI, L1 data cache hit rate, basic block size, percentage of memory reference instructions, and branch prediction rate. In Figure 2, we show these measurements for a couple of subroutines inside one application: *bzip2*. (The detail of the experimental setup is discussed later in Section 4.) We select subroutines that have an average invocation length of more than 50,000 dynamic instructions and only show the results for the top 8 subroutines based on dynamic instruction count. When nesting occurs, the callee's statistics (*e.g.*, execution time and number of instructions executed) are not counted in the caller again. We also include the weighted average of per-subroutine results. To meaningfully compare standard deviation of metrics having different units and magnitudes, we divide the standard deviation by the mean of the metric. This is also known as the *coefficient of variation*, often denoted as COV. We use hit rate and (correct) branch prediction rate rather than miss rate or misprediction rate. This is to avoid exaggerating the variation by dividing a very small mean value,

As we can see from Figure 2, the coefficient of variation is only about a few percentage points. An approximate interpretation is that *on average*, these statistics collected on one sample invocation of the subroutine would be just a few percent different from the overall average of all invocations of the same subroutine. Stated differently, compared to simulating a few sample invocations, faithfully simulating all invocations would only increase the accuracy marginally.

[1] In the scope of this paper, behavior is characterized by a range of statistics, architecture-specific or -independent, such as cache miss rate or percentage of memory reference instructions.



(a) Repetition of subroutine invocations



(b) Repetition of loop iterations

Figure 1. Two time-based IPC traces for SPEC FP application *art*. The boundaries of two subroutines (a) and loop iterations (b) are marked.

Indeed, in statistics theory, it is shown that given a distribution (with mean μ and variance σ^2) of a metric in a certain population, the average of the metric ($\overline{X}$) from random samples is a random variable. Its expectation $E(\overline{X})$ equals μ and its variance $Var(\overline{X})$ equals $\frac{\sigma^2}{n}$, where n is the size of the sample. As $n \to \infty$, the distribution of $\overline{X}$ becomes a normal distribution. To be exact, $\frac{\overline{X}-\mu}{\sigma/\sqrt{n}}$ becomes a standard normal distribution. This is commonly known as the *Central Limit Theorem*. Usually, when the sample size is sufficiently large, the distribution of $\overline{X}$ is treated as a normal distribution. Thus, in theory, if the population's variance is known in advance, the sample size can be mathematically determined such that the sampling error can be bounded with a set confidence. In practice, such bounding is only approximate in nature as the population's variance can only be estimated and skipping portions of the code in simulation creates non-sampling biases [2] that can not be bounded analytically. Nevertheless, such theoretical foundation substantiates our empirical methodology and provides us with confidence when interpreting the low error results we report later.

3. METHODOLOGY

Based on the behavior repetition of static code sections observed in Section 2, we propose *EXPERT*, a simulation strategy that samples selected dynamic code section instances.

In a straightforward implementation, the simulator alternates between a slow, detailed timing simulation mode and a fast, functional mode. Of all the dynamic instances of any particular code section, we simulate selected instances in detail, tallying architectural statistics that are of interest. The statistics of theses instances can then be used to accurately estimate those of all the dynamic instances for that code section. Combining the estimations of all code sec-

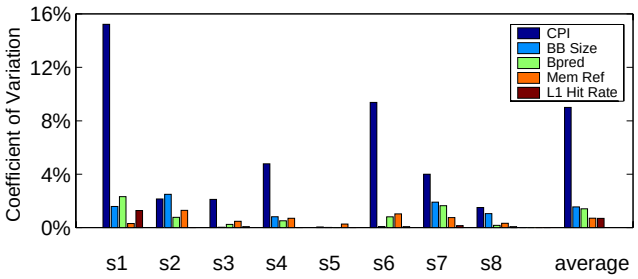


Figure 2. COV results of all 5 metrics for one application (*bzip2*) showing the top 8 subroutines (by dynamic instruction count) and the overall weighted average. The input we use in this experiment is the *train* input from SPEC CPU2000 distribution. *CPI, BB size, Bpred, Mem Ref,* and *L1 Hit Rate* refer to cycles per instruction, basic block size (number of instructions), rate of correct branch predictions, percentage of memory reference instructions, and hit rate of L1 data cache, respectively.

tions, taking into account the relative weight of different sections, we can obtain needed statistics for the entire application. This way, speed is greatly improved by avoiding repeated simulation of the same code section with little behavior variation. Accuracy is not compromised since simulating those instances would provide little extra information content.

To carry out the *EXPERT* strategy, we need to answer two questions for any application that we want to simulate. In our current implementation, we use straightforward approaches when answering the two questions. (1) How do we break down the application into different code sections? We break down the static code into natural units with manageable sizes. (2) How many and which dynamic instances to sample? We determine the number of the instances based on the variability of individual code sections. We separate the process of measuring code sections' behavior variability and the mechanism for the actual sampling. This leads to a three-step systematic process:

1. Partitioning: divide an application into static code sections,
2. Characterization: characterize the behavior repetition of these sections, and
3. Selective simulation: use the characterization to control the degree of sampling in an architectural simulation.

We consider the issues in these steps in turn, in the following sections.

3.1 Partitioning

The first step in the procedure is to partition the program binary into smaller units (*e.g.*, subroutines). At runtime, the execution of the program is then naturally partitioned and can be viewed as a sequence of dynamic instances of these static units. There are several reasons that lead us to using this code-based approach rather than the more popular alternative of directly partitioning the dynamic instruction stream into fixed-size windows. First, analyzing behavior repetition of instances of the same static code provides reassurance that the similarity in externally observed metrics is not a mere coincidence: two unrelated dynamic execution windows can have similarly low IPC due to two entirely different reasons, and thus when input or architecture changes, we have little confidence that their execution would again show similar IPCs. However, if the two windows are the dynamic instances of the same code section, then their behavior similarity is less likely to be merely coincidental. Second, unlike a fixed-size instruction chunk (of a haphazardly chosen size), these structures are the natural units that match the repetition periods as can be seen from Figure 1.

Notice that because of nesting, a dynamic instance of a static code section may contain portions that are non-consecutive at runtime, such as the portion before calling a subroutine and that after the return. Figure 3 shows a simplified example: subroutine A calls B and thus, one dynamic instance of A consists of A_1 and A_2. In this case, we aggregate the statistics collected in both segments to represent a complete logical instance of the static code section A.

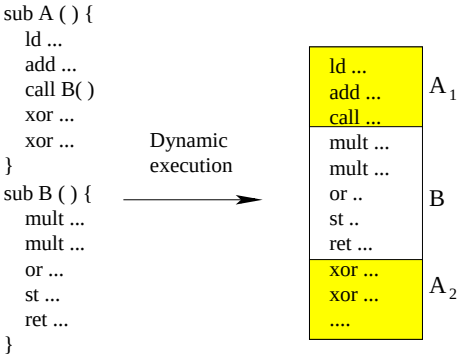


Figure 3. An example of non-contiguous dynamic instance of subroutine A. The statistics of this instance of A would be the aggregation of those of A_1 and A_2.

The dynamic instances of a particular static unit usually exhibit behavior repetition that can be exploited for different purposes. The ideal granularity of such a unit depends on the particular goal of the exploitation. In our case, the goal is to expedite architectural simulation. For this purpose we need the unit to be of a "medium" size. On the one hand, a typical instance can not be too small: modern architectures tend to be deeply pipelined, when the code unit is too small, certain simulation statistics will be determined more by the microarchitectural state than by the code itself. Thus, we can not properly characterize its behavior in an isolated fashion. For example, it is almost meaningless to discuss the IPC of a basic block in isolation. On the other hand, if an instance is too big, it may contain smaller subunits that also exhibit behavior repetition which can be easily exploited to further speed up simulation. In this paper, we focus on using subroutines and loops as the unit to statically partition the program. We start partitioning with the subroutines of the application. Since not all subroutines have the ideal grain size or same importance, we treat each one differently. In particular, we do not choose any short subroutine as a unit, we further partition long subroutines based on loops, and we ignore unimportant subroutines as explained in the following sections. The statistics used in this process are obtained from profiling.

3.1.1 Short subroutines

As mentioned earlier, taking statistics of a small code unit (such as a short subroutine) in isolation is highly susceptible to high non-sampling bias during simulation. Thus we treat short subroutines just as extended parts of their caller's execution. In Figure 3's example, if B is considered as a short subroutine (and A is not), then the dynamic instance of A includes all the instructions in the box.

We use a lower-bound limit I_{lower} and subroutines whose average per-invocation dynamic instruction count is less than I_{lower} are considered short. This dynamic instruction count does not include instructions from a callee subroutine, unless the callee is considered

a short subroutine. We profile the application to get the average exclusive dynamic instruction count for every subroutine and perform a post-processing to find out short subroutines.

During an *EXPERT* simulation, we may fast-forward to the beginning of a subroutine and start detailed simulation of one dynamic instance. Fastforwarding renders the microarchitectural state imprecise. The primary purpose of the threshold I_{lower} is to limit the influence of an imprecise microarchitectural state (or the non-sampling bias [2]). We choose a threshold of 50,000 instructions to make this warm-up period relatively insignificant.

3.1.2 Long subroutines

For subroutines that have long invocations, simulating even one such invocation in detail can be very time-consuming. Within these long subroutines, we further exploit behavior repetition of loop iterations to speed up the simulation. We identify iteration boundaries and loop termination by marking the backward branch and all the side exit branches of all chosen loops. At simulation time, the simulator can switch between the detailed mode and the fast-forward mode to perform sampling of iterations. For simplicity, we only work with major outer loops. We found that to be enough for the purpose of expediting simulations, for the applications we look at.

We classify a subroutine as a long subroutine if the average instruction count per invocation is higher than an upper limit I_{upper}. In this paper, I_{upper} is empirically set to 1 million instructions. We note that, for the applications we studied, the actual partitioning result is quite insensitive to the values of these two thresholds. When sampling loop iterations, we also want to ensure that a sample instance[2] contains enough instructions to make non-sampling bias insignificant. Thus, we make sure each instance contains enough iterations such that the total number of instructions simulated is at least I_{lower}.

When the simulator switches mode (between detailed and fast-forward mode), there is not only certain loss of accuracy but also switching overhead. Naturally, to profit from sampling a loop, the entire loop instance should be significantly bigger than the chunk we sample. Otherwise, simulating the entire loop is a better option. Thus when selecting loops, we only select long loops that have an average instance size higher than a threshold. We use the same threshold I_{upper} that we use for identifying long subroutines. This is because the principle is the same: if a code instance is shorter than a certain size, we do not want to switch mode during its simulation, as that would yield limited return of speed improvement, and increase the inaccuracy due to imprecise architectural state.

3.1.3 Unimportant subroutines

There are code sections that are executed so infrequently that they can be completely ignored. We sort the code sections by their total weight of dynamic instruction count and create a cutoff line below which the *combined* weight of all the sections is less than 0.5%. Before this pruning process, the average number of static code sections is about 40 for all the applications we studied. After the pruning, the average is only 13.7 (Section 5.1). We have verified that the combined weight of all the ignored sections is less than 0.5% in the production run for every application. We note that even if the threshold is set to a much tighter 0.1% we can still ignore about half of the static code sections.

Once all the important, properly-sized static code sections are identified, we can mark their boundaries by instrumentation so that at runtime we can track the control flow entering and exiting these sections. In our implementation, this instrumentation is done dynamically when the binary is initially loaded into the simulator (after reading the data file where we keep the partition information). We also maintain a call stack: when a code section finishes execution, it returns to the section from which the control is transferred.

[2] In statistics terminology, a *sample* is a collection of units selected from the population. In this paper, these individual units are also referred to as (sample) instances.

3.2 Characterization

After identifying all the code sections with proper sizes in the partitioning stage, we characterize the degree of behavior variability. This variability will be used to determine the rate of sampling: the more stable the behavior is, the less sampling is needed.

For every static code section, we measure a range of metrics per dynamic instance and compute the COV of these metrics. In the case of a subroutine, an instance is one invocation. In the case of a loop, an instance would be a chunk of iterations containing no less than I_{lower} instructions. These metrics reflect different aspects of the program behavior and thus it is normal to have a small COV value on one metric, but a large value on another. Naturally, only when all the values are small does it suggest stable behavior. Thus we pick the largest COV value out of all metrics as a measurement of the degree of behavior variation. For convenience, we call this measurement the variation factor, denoted V.

Intuitively the variability of the behavior is to a large extent determined by the code. However, input and architectural changes do affect the variability. Ideally, we want the variability characterization to be independent of input and architectural changes, so as to avoid repeated characterization processes. Thus, we perform the measurement experiment in four different settings using two inputs and two machine configurations. For each code section, we pick the maximum V computed in all settings.

From Figure 2, we see that different subroutines have different degree of variation, but in general, the variation is small. We summarize the result of the characterization experiments in Figure 4. For each metric, we compute the COV of all the instances of every code section. These per-code section results are then weighted averaged to get the application-wide COV of that metric. We use the total number of dynamic instructions inside each code section as the weight. For brevity, we only show this weighted average value per application for all five metrics. Figure 4 shows these results obtained in one of the four settings. In general, the stability characterizations under different settings agree quite well. This observation is in line with the intuition that the code has a very large bearing on the behavior (and its variability). Therefore, besides using simulation, the measurements needed in this step could also be taken by actually running the program on a similar, but existing (previous-generation) architecture.

We emphasize that this characterization process is a one-time effort for each application. The results will dictate future sampling rate for that application regardless of the input used or the architecture configuration used. One possible alternative to pre-characterization is to carry out the characterization process online, together with an *EXPERT* simulation. We defer the discussion of this alternative to Section 5.4.

3.3 Selective Simulation

Finally, with the characterization of behavior repetition we can perform the expedited simulation, skipping portions of the execution while performing detailed simulation for the rest. In this process, we need to determine two things:

1. Selection: how many and which instances to select for detailed simulation.
2. Skipping: what needs to be done for the instances not selected.

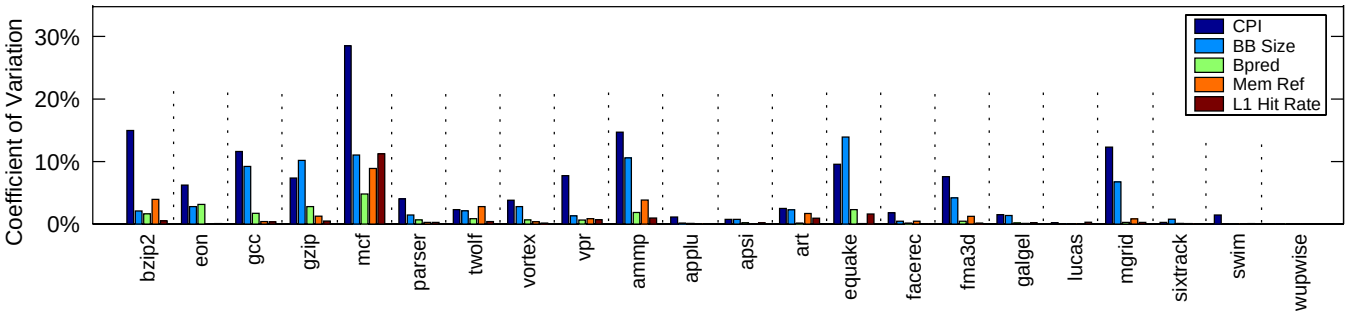


Figure 4. COV of five metrics measured for code section instances. Results for each application are the weighted average of per-section results. The input is *train* from SPEC 2000, and the architecture is an 8-issue processor similar to IBM POWER4.

We discuss these two issues under two different scenarios: whether or not we can perform some preprocessing.

3.3.1 With preprocessing

Selection Recall that if we select n instances, the mean of a metric from these instances is a random variable with a variance of σ^2/n, with σ being the population's standard deviation of the metric. Given a desired confidence level $(1-\alpha)$ and the tolerable margin of error E, we can compute the necessary sample size $n = (z_{\frac{\alpha}{2}} * \frac{\sigma}{E})^2 = (z_{\frac{\alpha}{2}} * \frac{\sigma/\mu}{E/\mu})^2$, where $z_{\frac{\alpha}{2}}$ is a constant. We use the variation factor V to approximate σ/μ (recall that V is the highest COV of a range of metrics in a set of testing experiments). Thus, the sample size is $n = (z_{\frac{\alpha}{2}} * \frac{V}{E/\mu})^2$, where E/μ is the tolerable relative error.

Given that during simulation, the instances of a code section are presented in sequence and that they are *not* truly randomly distributed due to program locality, simple systematic sampling, which picks instances evenly distanced, can be quite effective. To perform systematic sampling with a desired sample size, we have to know the population size (N) in advance so that we can pick one out of every N/n units. The population size can be obtained using some preprocessing. Usually this could be done by executing an instrumented version of the binary on some hardware or, in the worst case, by using a fast functional simulation. In many cases of using simulation, we need to simulate the same application multiple times, *e.g.*, to perform a design space search or a sensitivity analysis. In these cases, the preprocessing overhead is amortized.

Skipping For instances that are not sampled, the only thing we need to do is to make sure that their effect on the architectural state (memory and registers) and microarchitectural states is reflected when we start to simulate the next selected instance. For microarchitectural states, we only keep track of caches and branch predictor tables. In keeping these the microarchitectural state, we pay a relatively small price of speed but significantly reduce the cold-start effect, a form of non-sampling bias.

When preprocessing the simulated applications, since we can predetermine which instances to choose, we can create checkpoint files reflecting the snapshot of architectural and microarchitectural states at the beginning of selected sample instances. At production simulation time, after finishing the detailed simulation of one instance, we can load the next checkpoint file, and in effect, instantly fast-forward over an arbitrary number of instructions. Checkpointing requires a full-length simulation for each program-input pair. However, since timing is not necessary, this simulation is basically a modified functional-level emulation that can be quite fast. Furthermore, the assumption is that the overhead of checkpoint creation can be amortized over many simulation runs. To maintain accurate microarchitectural states, caches and branch predictors are updated during the functional emulation.

An alternative to keeping the microarchitectural states also in the checkpoints is to create checkpoints at a certain number of instructions prior to the start of sample instances. The extra instructions would be executed during actual simulations to warm up the microarchitectural states. In this paper, we choose the first approach which is easier to implement.

We notice that although the size of a single checkpoint file is usually small, the total number of checkpoints can be quite high for certain applications. We find that placing a cap on the sample size of any individual code section is a very effective way of reducing the total number of checkpoints. To select the cap, we tested a few prime numbers ranging from 100 to 800. There is very little impact these caps have on the sampling error. We use 127 as the cap in this paper.

3.3.2 Without preprocessing

As we will see later in Section 5, preprocessing can significantly reduce the time for the production simulation. However, there is an initial cost that has to be paid. When the application is to be simulated only once, preprocessing can be very cumbersome: the application may need to be scanned twice before the actual simulation, once to get the population size so as to decide which ones to sample and another time to create checkpoints[3]. In this case, a different strategy can be used to avoid the unnecessary overhead. We first discuss the easier issue of skipping.

Skipping Obviously, if the application is to be simulated only once, we do not need a separate run to create checkpoints. For instances that are not sampled, we simply fast-forward through them using functional simulation to update the states.

Selection When drawing a sample of code section instances, not only the sample size is important, where to pick the instances is equally important. For example, it is not surprising that selecting instances only from part of the program (*e.g.*, the beginning), even in large numbers, may create a big sampling error. Indeed, if we follow the statistic theory and compute the sample size needed to guarantee an error of less than 3% with a 99% confidence, but instead of randomly picking n instances, we pick the very first n instances, then the sampling error can be 10-15% for many applications we looked at. Clearly, in addition to selecting sufficient instances to provide statistical confidence, we need to spread them

[3] In theory one can scan only once, creating checkpoints at all potential positions and keep the right ones when the population size is finally known. In practice, this incurs impractical space requirements. Furthermore, for incremental checkpoints, which we use, an extra pass is still needed.

across the entire execution to reflect the program's behavior. This latter point is as important, if not more so.

To ensure that the sample instances are drawn from all over the execution, we can simply maintain a sampling *rate*. We will keep on sampling until the end of the program, even after we have acquired enough instances. Obviously, this can lead to unnecessary over-sampling, especially for loops with a large number of iterations. However, we note that the speed of fast-forwarding sets an upper bound on the achievable simulation speedup. Therefore, moderate over-sampling is not a concern for speed of simulation. We find that if we can limit the degree of over-sampling for code sections with a large number of instances, the simulation speedup will largely be determined by that of fast-forwarding. Hence we use a simple empirical formula to decide the base sampling rate for each code section: aV^2. Here V is the variation factor and a is a constant that we need to decide. In this paper we use $a = 1$ (see Section 5.3). Additionally, we identify code sections with a large number of dynamic instances (having more than 10,000 instances in the profiling run), and set a 1% cap on the sampling rate. Overall, as our experiments will show later, this approach is practical and effective.

During the selective simulation, without prior knowledge of an upcoming code section instance, we may select a short instance (*e.g.*, a subroutine invocation taking a side-exit) for detailed simulation. We choose to discard the statistics of such an instance and obtain one more (the next) instance. This is because when the number of instructions is small, the statistics collected can be much more susceptible to state-induced bias[4] (*e.g.*, pipeline filling and draining). Furthermore, since we compute weighted average, the result of an instance with very few instructions is simply noise. We classify selected instances with less than $10\% * I_{lower}$ dynamic instructions as short instances. Though in theory, discarding side-exit instances create a sampling bias, making short instances underrepresented, in practice we have verified that this does not create a problem. These side exits are rare and unimportant. Execution weight of all short instances combined is below 0.1% of every single application we studied.

3.3.3 *Summarizing statistics*

Finally, after selective simulation, for each code section, we obtain a range of statistics of interest for the selected instances and the total number of instructions in all instances. Based on the sample instances, we can extrapolate the statistics for all instances and then combine them into program-wide results. We note that in this process, for many ratio-type statistics such as branch prediction rate, the mathematically correct way to compute its program-wide result is to separately predict the numerator and the denominator. Using weighted average of per-section ratio to compute program-wide ratio creates only a small error (*e.g.*, about 0.3% for branch prediction rate). However, in our system, this is actually higher than the sampling error (about 0.2%).

4. EXPERIMENTAL SETUP

To evaluate our proposal, we perform a set of experiments using the SimpleScalar [1] 3.0b tool set and 22 applications from the SPEC CPU2000 benchmark suite. The applications are listed in Table 1. We use Alpha binaries.

We use different settings for different purposes. In the characterization phase, each application is simulated four times using two different input sets and on top of two different system configurations. The two different input sets are the *train* input set from SPEC 2000 and the MinneSPEC input set (large size) [11]. The two chosen configurations are modeled after MIPS R10000 and IBM POWER4, respectively.

Suite	Applications
SPEC Int	bzip2, eon, gcc, gzip, mcf, parser, twolf, vortex, vpr
SPEC FP	ammp, applu, apsi, art, equake, facerec, fma3d, galgel, lucas, mgrid, sixtrack, swim, wupwise

Table 1. Applications used in the experiment.

In the production phase, we compare *EXPERT* to a faithful detailed simulation of long running applications. The baseline simulator is a modified out-of-order processor simulator. The *EXPERT* version has extra support for mode switching and for loading partitioning information and variability characterization information. Both versions simulate all 22 applications using the standard reference (*ref*) input running on an architecture that is similar to Alpha 21264. Table 2 lists some of the parameters for this architecture. This production setting is different from all four settings used in the characterization phase.

Processor core	
Issue/Decode/Commit width	6 / 4 / 11
Issue queue size	20 INT, 20 FP
Integer ALUs	4 + 1 mul/div unit
Floating-point ALUs	2 + 1 mul/div unit
Branch predictor	Bimodal and 2-level PAg
- Level1	1024 entries, 10-bit history reg
- Level2	1024 entries
- Bimodal predictor size	1024 entries
- Meta table size	4096 entries
- BTB	4096 sets, 2 way
Branch misprediction latency	7+ cycles
Reorder buffer size	128
Load/Store queue size	64
Memory hierarchy	
L1 instruction cache	64KB, 2-way, 2 cycles
L1 data cache	64KB, 2-way, 2 cycles
L2 unified cache	1MB, 4-way, 12 cycles
Memory access latency	100 cycles

Table 2. System configuration.

5. EVALUATION AND ANALYSIS

In this section, we evaluate various aspects of our proposed technique and present some quantitative analyses. We report some statistics of the code sections obtained in the partitioning step in Section 5.1; we discuss the characterization step in Section 5.2; we evaluate the effectiveness of our approach in terms of simulation errors and speedup in Section 5.3; and finally, we discuss checkpointing and some other optimizations in Section 5.4.

5.1 Properties of Partitioned Code

Code section granularity In Table 3, we list some statistics of the code sections our partitioning algorithm generates. In the interest of brevity, we only show the range of these statistics. We see that different applications have widely different characteristics. The partitioning algorithm in general chooses medium-grain sections.

The effect of pruning Filtering out unimportant code sections is a very useful optimization, as these sections would otherwise be simulated in detail, reducing potential speedup. We estimate

[4] In this paper, we avoid using the term "cold-start bias" as it tends to suggest that the caches and branch predictors are not updated at all during fast-forwarding. We use "state-induced (non-sampling) bias" instead.

Metric	Value range
Number of static code sections	13 (mgrid) - 90 (galgel, twolf)
After pruning	4 (sixtrack, twolf) - 33 (gcc)
Total number of dynamic instances	22K (eon) - 6.6M (twolf)
Average instance size	52K (twolf) - 6.1M (wupwise)

Table 3. Code section property statistics. The last two items are collected in the production setting. *Average instance size* is measured in number of dynamic instructions.

that pruning reduces the number of checkpoints by about 30%. Its impact on the speedup of preprocessing-based simulation could be similar.

5.2 Variation Characterization

An important component of the *EXPERT* strategy is to characterize the code sections' behavior variability. In this one-time characterization step, we use multiple training input sets and vary the architecture configuration. By doing so, we hope to reveal the potential variability of the code under a wide variety of input sets and system configurations. To avoid underestimating the variation, for each code section, we use the maximum COV of any metric using any training settings to serve as the behavior variation factor V. To see if the code sections have (significantly) higher variability using the production setting – which suggests that using training settings to characterize behavior variation is invalid – the weighted average of the underestimation factor is computed and shown in Figure 5. For each code section, this underestimation factor is defined as the difference between the variation factor V and the COV of a given metric measured in the production setting, **if** the COV is greater than V. As we can see, only a small portion of the applications have noticeable underestimation. Even for them, the degree of underestimation is small. This experiment and the fact that characterization using different settings agree relatively well suggest that using pre-characterization of behavior repetition is a viable approach.

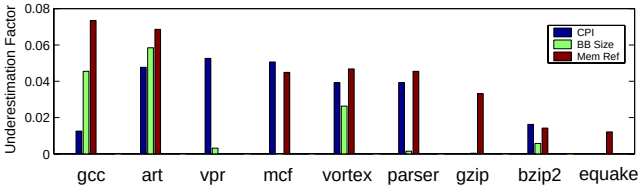


Figure 5. The underestimation factor for 3 metrics and 9 applications. The underestimation factors of all metrics are less than 0.5% in the other 13 applications. The underestimation factor for branch prediction rate and L1 data cache hit rate is below 1% for all applications.

To put earlier results of code-based COV measurements (Figure 4) into perspectives, we compare them to the COV computed over fixed-size instruction chunks. The comparison is shown in Figure 6. We only show the this comparison for CPI. The setting used is the same as that in Figure 4 (*train* input set and 8-issue processor). The chosen chunk sizes are 1, 10, 100 thousand, and 1 million instructions. The code section size varies. The size of an individual instance ranges from 2 thousand to over 10 million instructions. Program-wide average instance size ranges between 40 thousand and 2 million instructions.

This comparison is to show that COV across code section instances is low because of repeating program behavior, not because

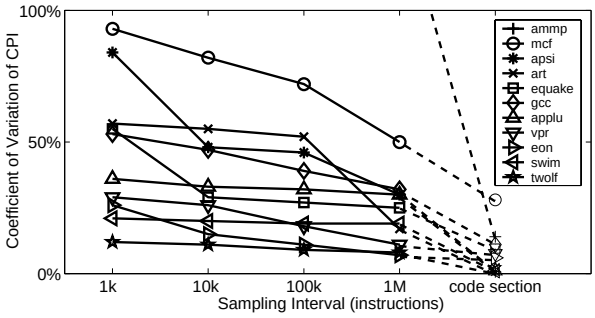


Figure 6. COV of CPI measured in fixed-length intervals compared to COV of CPI measured in code-section instances. To reduce clutter, we sort the applications by their value in 1 thousand-instruction chunk and show every other application in the order.

these metrics inherently have little fluctuation. As can be seen, the CPI measured at fixed-size intervals of all sizes generally has a much higher COV. Depending on the interval length, the average of interval-based COV for all applications ranges from 36% to 56%, and individual values can be as high as 280%. In contrast, the average of code-section based COV is only about 5%.

5.3 Accuracy and Speed

To decide on the various sampling parameters, we faithfully simulate the whole applications in detail and collect traces of all the statistics of interest for individual code section instances. We then try different sampling methods and calculate the sampling error and potential speed improvement. Since actual simulation speed depends on many factors, we simplify this analysis by using the *reduction factor*, which is the ratio between the total number of instructions in the entire execution and that in the sampled code section instances.

We first look at the impact of sample size. In essence, our selective simulation is a form of stratified sampling. In each stratum (*i.e.*, code section), we use systematic sampling. We either have a fixed sample size $n = (z_{\frac{\alpha}{2}} * \frac{V}{E/\mu})^2$ (with preprocessing) or a fixed sampling rate aV^2 (without preprocessing), both with a cap (Section 3.3). We vary different parameters, including confidence (1-α), error tolerance (E/μ), minimum sample size (per stratum), and the constant factor a. In Table 4, we show the resulting sampling errors and the reduction factors. For brevity of presentation, we only show the sampling error for CPI and we summarize the error and the reduction factor results from all 22 applications into maximum, minimum, and average. We use geometric mean when averaging the reduction factor.

From the table we can see that increasing sampling rate in general lowers the error. However, it does not guarantee reduction in error (99-3-3 has a lower average error than 99-3-30). Also, although not evident from the table, having a smaller sample size does not guarantee a higher reduction factor because the instances sampled can have different numbers of instructions. We notice that the minimum sample size does not affect the error results noticeably, at least in our experimental settings, but can significantly affect the reduction factor. We find that a small value suffices[5].

From the data, we believe that the differences among various

[5] If bounding the confidence interval is of paramount importance, we note that choosing the typical minimum size of 30 for each code section (stratum) can be suboptimal in terms of simulation speedup. The formula to find globally optimal sample sizes for all code sections should be used instead.

	E_{avg}	E_{max}	E_{min}	R_{avg}	R_{max}	R_{min}
99-3-30	0.65%	3.81%	0.03%	181	68252	2.8
99-3-3	0.57%	3.81%	0.02%	427	242598	2.8
95-3-30	0.66%	3.87%	0.04%	194	68252	2.9
95-3-3	0.67%	3.87%	0.02%	511	415276	3.1
90-3-30	0.95%	4.59%	0.04%	204	68252	3.1
90-3-3	0.99%	4.59%	0.03%	583	502179	3.2
$a = 4$	0.52%	2.58%	0.01%	84	529	5.5
$a = 2$	0.76%	4.24%	0.03%	124	1014	13.8
$a = 1$	0.84%	4.99%	0.04%	173	1669	17.0
$a = 1/2$	0.89%	5.31%	0.04%	262	4672	53.2
$a = 1/4$	1.00%	7.24%	0.01%	384	4918	88.1

Table 4. Sampling error and reduction factor for different sampling methods. E_{xxx} and R_{xxx} refer to sampling errors and reduction factors respectively. The upper part shows fixed-size sampling. X-Y-Z indicates a fixed-size sampling with a confidence $(1 - \alpha)$ of X%, an error tolerance (E/μ) of Y%, and a minimum sample size of Z. The minimum reduction factor comes from application *gcc*. Excluding *gcc*, the minimum reduction factor is around 40. The lower part shows fixed-rate sampling with different constant factor a.

fixed-size sampling methods are relatively small, and all these methods are reasonable choices. We use 99-3-3 in the following experiments. For fixed-rate sampling, we choose $a = 1$, which is a good design point.

In systematic sampling, every k^{th} unit is selected. It is very convenient in simulation since all code section instances are naturally sequenced. Given a desired sample size and the population size, or given the sampling rate, k can be easily computed. The only issue left is to choose the starting point from the first k units. There are a few obvious options: choosing the first, the $\lceil k/2 \rceil^{th}$ (center), the k^{th} (last), and a randomly selected one. Knowing that some code has monotonically changing behavior, we believe that the instance in the center of a group tends to best reflect the average condition of the group. We try out these options and show the maximum and average sampling error in Table 5. We only show the results for fixed-size sampling of 99-3-3, and fixed-rate sampling of V^2 (*i.e.*, $a = 1$). We see from the table that picking the center unit as the starting point is a good option, while picking the first or a random starting point are less desirable. Data shown in Table 4 is using this method. Finally, less intuitively, picking the last sample in every group is also very effective.

		first	center	random	last
V^2	E_{avg}	1.32%	0.84%	1.03%	0.87%
	E_{max}	6.91%	4.99%	4.34%	4.39%
99-3-3	E_{avg}	1.01%	0.57%	1.05%	0.44%
	E_{max}	8.24%	3.81%	6.03%	1.87%

Table 5. Sampling error and reduction factor for different starting positions.

We now analyze the actual speedup and error obtained from expedited simulation.

With preprocessing Figure 7 shows the reduction factor and simulation speedup when preprocessing is used. Speedup is calculated using wall-clock time and is thus subject to noise due to hardware and load differences. We can see that significant speedups are achieved and the actual speedup tracks reduction factor rather well as expected. Out of the 22 applications, except *gcc* which takes about 10 hours, all finishes under 5 hours on mid-range PCs (*e.g.*, 2GHz Pentium 4-based PCs). More than half of them finish within an hour. Fully detailed simulation, on the other hand, takes from 40 to more than 500 hours.

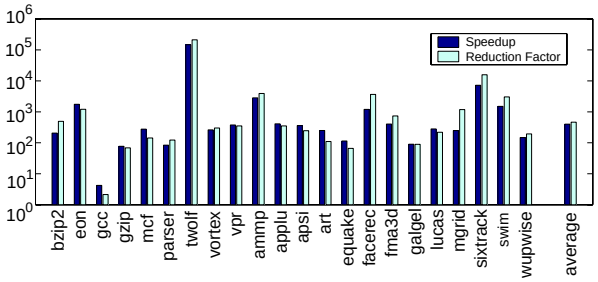


Figure 7. Simulation speedup and reduction factor shown in log scale. The average is shown in geometric mean.

Figure 8 shows the errors of all metrics for every application. From the figure, we see that all the errors are relatively small (less than 3.8%). The average error ranges from 0.2% for branch prediction rate to 1.1% for basic block size. The average error in CPI is about 0.89%. On average, floating-point applications tend to have slightly smaller errors than integer applications. For some applications, such as *apsi* and *galgel*, almost all the errors are negligible. We also see that CPI is not necessarily the most error-prone metric. Thus, when evaluating an expedited simulation infrastructure, we should look at broader metrics.

Without preprocessing Without the benefit of preprocessing the applications, we need to fast-forward over the dynamic instances not selected for detailed simulation. This slows downs the simulation process. Figure 9 shows the simulation speedup. We note that the simulations are performed over many different machines with different configurations and loads. Therefore the speedup calculated has limited precision. We also show the speedup of a fast-forwarding simulation (with cache and branch predictor state update). This essentially serves as an upper bound of speedup. Since the average speedup comes very close to this upper bound, under the current implementation, further improving the sampling mechanism has a limited impact on the final simulation speed. However, improving the speed of the fast-forwarding mechanisms is part of our future work.

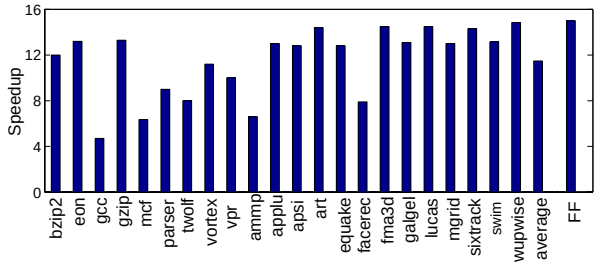


Figure 9. Speedup for selective simulation without preprocessing. The average is shown in geometric mean. FF stands for fast-forwarding.

The errors in fixed-rate sampling are quite similar to those reported in Figure 8. The average errors ranges from 0.12% for L1 hit rate to 0.83% for CPI. Except a 5.3% error of CPI in application *gcc*, all the other errors are below 3.8%.

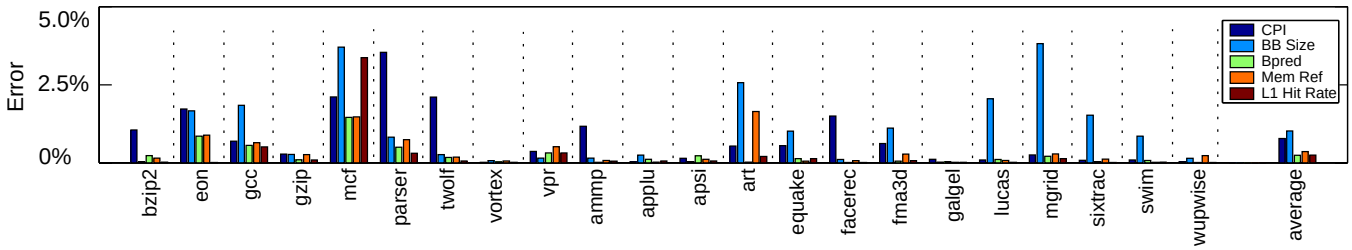


Figure 8. Simulation error.

5.4 Discussions

5.4.1 *Checkpointing*

In our selective simulation process, since we only simulate a small percentage of all dynamic instructions, fast-forwarding the rest of the instructions becomes a bottleneck. As mentioned before, creating and loading checkpoints can significantly speed up fast-forwarding. This is quite obvious if we compare the results in Figures 7 and 9. However, checkpointing does incur a non-negligible space overhead. The number of checkpoints (in 99-3-3) ranges from 20 to over 4000. The total storage size for the checkpoint files stored in compressed format ranges from a few megabytes to over 5 gigabytes.

5.4.2 *Other optimizations*

One possible optimization of *EXPERT* is to eliminate or greatly simplify the process of offline characterization. We have experimented with two different approaches: online characterization, and characterization without timing simulations. We summarize our current findings below.

Online characterization It is conceivable to create a simulation infrastructure that automatically performs the variation characterization at simulation time. For the first k instances of every code section, the simulator performs detailed simulation and figures out the code's variability. Then it starts to sample at a certain rate. As more sample instances are drawn, the variability characterization is also updated, and the sampling rate is adjusted accordingly. Our experience shows that making this approach both accurate and efficient at the same time is challenging. Using small values of k, we have seen maximum sampling error as high as 20%. Without the benefit of pre-characterization, we really need a large sample size so that the standard deviation of the sample (S) can be a reasonable estimator of that of the population (σ). This requirement can significantly lower the potential speedup.

Inexpensive characterization In the pre-characterization phase, among other metrics, timing information (CPI) is used. It is significantly slower to obtain than other metrics. To find out if the timing information is dispensable, we perform two experiments to conduct the characterization in a different, inexpensive way. First, we replace the CPI information in the out-of-order processor model by that obtained from an in-order processor model. Second, we eliminate CPI from the calculation of the variation factor V altogether. We find that, not surprisingly, using these characterization methods to dictate sample sizes leads to increased sampling error. Although the average error is still tolerable (2-3%), the maximum error can be as high as 11% using in-order processor model and 25% when discarding CPI altogether.

6. RELATED WORK

Architecture-level simulation, is one of the most important tools in architectural studies and design, especially in the early stages. As the complexity of computer systems increases, even this relatively high-level approach will become increasingly time-consuming. Many techniques have been proposed to reduce the amount of detailed simulation, and still maintain the accuracy of measurements to a certain extent. Conte *et al.* are among the first to look at sampling based simulations [2]. They also addressed issues such as reduction in sampling and non-sampling biases.

Sherwood *et al.* propose SimPoint [20], an automatic system to reduce the dynamic instruction stream into one or a handful of representative chunks, each consisting 100 million instructions. To find this subset, every chunk of 100 million instructions of the entire instruction stream is represented as a basic block vector. This transforms the whole dynamic instruction stream into a set of points in a high-dimensional space. These points are then manipulated and eventually clustered based on their mutual distances. Inside each cluster, one point close to the center of weight is chosen to represent the whole cluster. This point corresponds to a chunk of instructions that should be simulated in detail. A particularly important advantage of SimPoint is that the large sampling unit size makes the warm-up error negligible. However, the down side of this approach is that it constraints the number of sample points and results in relatively high sampling error.

Wunderlich *et al.* propose SMARTS, which uses a different way to select instruction chunks [21]. Instead of using clustering algorithm to identify mutually similar chunks, systematic sampling is used. The chunk size used, around 1000 instructions, is much smaller compared to SimPoint. SMARTS uses rigorous theoretical guidelines to determine sampling rate which results in low sampling error.

While SimPoint and SMARTS are very powerful, *EXPERT* is an important alternative. The uniqueness of *EXPERT* is that instead of operating on fixed-size instruction chunks, it operates on high-level program constructs that naturally coincide with program behavior changes. This allows us to reduce the simulation requirement by exploiting program behavior repetition in a straightforward manner. Moreover, it allows a much easier analyze-and-optimize cycle for joint architecture and code optimizations. Since these aforementioned approaches essentially treat the instruction stream as a black box, even a simple software transformation of a subset of the code necessitates a complete rerun or even a re-characterization. *EXPERT*, however, can be used to quickly evaluate the effect of localized software transformations on select code sections. Overall, future simulation infrastructures may well include mechanisms from both paradigms.

Marculescu *et al.* also use sampling of instruction chunks to speed up simulation [12]. Instead of simple systematic sampling, they use a hot-spot detector [13] to identify frequently executed regions of code. Inside a hot-spot, sampling of fixed-size instruction chunks is used to obtain various statistics.

Schnarr and Larus use memoization to replay actions stored in a *processor-action cache* when the current microarchitectural state matches a previously seen state [19]. There is no approximation

or loss in accuracy by doing memoization. The tradeoff is that the current state has to match exactly what was seen previously. As microarchitectural structures become larger and more complex, memoization could become less applicable.

Duesterwald *et al.* also exploit program behavior repetition in [4]. They have found that the repeating pattern of different metrics have similar rate of repeatability. This property of correlation is used to design an asymmetric predictor, where the history information of *other* metrics is used to predict the target metric. Philosophically, this is similar to correlating branch predictors. The underlying reason for the observed correlation is that as code sections repeat in a certain pattern, externally visible metrics repeat in the same pattern. In contrast, we exploit behavior repetition in an explicit fashion by tying the repetition with the code that generates it. Our earlier work also showed that behavior repetition can be exploited to control adaptive hardware [8, 7].

Haskins and Skadron introduced minimal subset evaluation (MSE), a probabilistic approach to minimize the amount of instructions needed to warm up buffers like caches [9]. In [10], they look at using memory reference reuse latency information to reduce the number of instructions needed to warm-up processor states. Incorporating their scheme could potentially improve speed of fastforwarding. We leave this as future work.

Finally, there are other approaches to expedite program simulation or analysis that are rather orthogonal to *EXPERT*'s method. These include input set design [11], statistical synthetic workload generation or simulation [17, 5, 16, 15], parallel simulations [14, 6], and machine-independent analyses [18, 3].

7. CONCLUSIONS

In this paper, we have shown that there is a large degree of behavior repetition in common applications when the same code sections are repeatedly executed. This behavior repetition is exemplified by the low variation of metrics collected over invocations of subroutines and iterations of loops. By exploiting this fact, we can greatly reduce the amount of detailed simulation needed to evaluate designs. Many statistics can be collected from sample instances of a static code section, rather than from faithfully simulating every instance. We have shown that, program behavior repetition can be pre-characterized, and we can spend less time simulating code sections that are quite stable. Based on this principle, we have proposed an expedited simulation strategy called *EXPERT*. For a set of 22 SPEC 2000 applications, our experiments show that simulation time can be reduced to a few hours or even several minutes if preprocessing is used. The speed improvement is achieved with little loss in accuracy. The average error of a range of metrics is about 1% or less.

Acknowledgments

The authors wish to thank the anonymous reviewers for their insightful comments and suggestions.

8. REFERENCES

[1] D. Burger and T. Austin. The simplescalar tool set,version 2.0. Technical report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[2] T. Conte, M. Hirsch, and K. Menezes. Reducing State Loss for Effective Trace Sampling of Superscalar Processors. In *International Conference on Computer Design*, pages 468–477, Austin, Texas, October 1996.

[3] C. Ding and Y. Zhong. Predicting Whole-Program Locality through Reuse Distance Analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, California, June 2003.

[4] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 220–231, New Orleans, Louisiana, September 2003.

[5] L. Eeckhout, K. De Bosschere, and H. Neefs. Performance Analysis through Synthetic Trace Generation. In *International Symposium on Performance Analysis of Systems and Software*, pages 1–6, Austin, Texas, April 2000.

[6] S. Girbal, G. Mouchard, A. Cohen, and O. Temam. DiST: A Simple, Reliable and Scalable Method to Significantly Reduce Processor Architecture Simulation Time. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 1–12, San Diego, California, June 2003.

[7] M. Huang, D. Chaver, L. Piñuel, M. Prieto, and F. Tirado. Customizing the Branch Predictor to Reduce Complexity and Energy Consumption. *IEEE Micro*, 23(5):12–25, September 2003.

[8] M. Huang, J. Renau, and J. Torrellas. Positional Adaptation of Processors: Application to Energy Reduction. In *International Symposium on Computer Architecture*, pages 157–168, San Diego, California, June 2003.

[9] J. Haskins Jr. and K. Skadron. Minimal Subset Evaluation: Rapid Warm-up for Simulated Hardware State. In *International Conference on Computer Design*, pages 32–39, Austin, Texas, September 2001.

[10] J. Haskins Jr. and K. Skadron. Memory Reference Reuse Latency: Accelerated Sampled Microarchitecture Simulation. In *International Symposium on Performance Analysis of Systems and Software*, pages 195–203, Austin, Texas, March 2003.

[11] A. KleinOsowski and D. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, June 2002.

[12] D. Marculescu and A. Iyer. Application-Driven Processor Design Exploration for Power-Performance Trade-off Analysis. In *International Conference on Computer-Aided Design*, pages 306–313, San Jose, California, November 2001.

[13] M. Merten, A. Trick, C. George, J. Gyllenhaal, and W. Hwu. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In *International Symposium on Computer Architecture*, pages 136–147, Atlanta, Georgia, May 1999.

[14] A. Nguyen, M. Michael, A. Nanda, K. Ekanadham, and P. Bose. Accuracy and Speed-up of Parallel Trace-Driven Architectural Simulation. In *International Parallel Processing Symposium*, pages 39–44, Geneva, Switzerland, April 1997.

[15] D. Noonburg and J. Shen. A Framework for Statistical Modeling of Superscalar Processor Performance. In *International Symposium on High-Performance Computer Architecture*, pages 298–309, San Antonio, Texas, February 1997.

[16] S. Nussbaum and J. Smith. Modeling Superscalar Processors via Statistical Simulation. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, Barcelona, Spain, September 2002.

[17] M. Oskin, F. Chong, and M. Farrens. HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs. In *International Symposium on Computer Architecture*, pages 71–82, Vancouver, Canada, June 2000.

[18] R. Saavedra and A. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, November 1996.

[19] E. Schnarr and J. Larus. Fast Our-of-Order Processor Simulation Using Memoization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, San Jose, California, October 1998.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, San Jose, California, October 2002.

[21] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *International Symposium on Computer Architecture*, pages 84–95, San Diego, California, June 2003.