Cross Binary Simulation Points

Erez Perelman[†] Jeremy Lau[†] Harish Patil[‡] Aamer Jaleel[‡]
Greg Hamerly[◇] Brad Calder[†*]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Intel Corporation
[*]Microsoft Corporation
[◇]Department of Computer Science, Baylor University

{eperelma,jl,calder}@cs.ucsd.edu
{harish.patil,aamer.jaleel}@intel.com
greg_hamerly@baylor.edu

Abstract

Architectures are usually compared by running the same workload on each architecture and comparing performance. When a single compiled binary of a program is executed on many different architectures, techniques like SimPoint can be used to find a small set of samples that represent the majority of the program's execution. Architectures can be compared by simulating their behavior on the code samples selected by SimPoint, to quickly determine which architecture has the best performance.

Architectural design space exploration becomes more difficult when different binaries must be used for the same program. These cases arise when evaluating architectures that include ISA extensions, and when evaluating compiler optimizations. This problem domain is the focus of our paper. When multiple binaries are used to evaluate a program, one approach is to create a separate set of simulation points for each binary. This approach works reasonably well for many applications, but breaks down when the simulation points chosen for the different binaries emphasize different parts of the program's execution. This problem can be avoided if simulation points are selected consistently across the different binaries, to ensure that the same parts of program execution are represented in all binaries.

In this paper we present an approach that finds a single set of simulation points to be used across all binaries for a single program. This allows for simulation of the same parts of program execution despite changes in the binary due to ISA changes or compiler optimizations.

1 Introduction

Modern computer architecture research requires understanding the cycle level behavior of a processor as it executes a program. To gain this understanding, researchers typically employ detailed simulators that model the processor's cycle-level behavior. Unfortunately, this level of detail comes at the cost of speed. Even on the fastest simulators, modeling the full execution of a single benchmark can take weeks or months to complete, and nearly all industry standard benchmarks require the execution of a *suite* of programs. Therefore, instead of simulating entire programs, a few small samples of each program's execution are sampled instead. The samples we focus on are in the range of 10 to 500 million instructions.

One of the challenges is to determine which simulation samples most accurately represent the program's full execution. To address this problem we created a tool called SimPoint [11, 16] that uses clustering algorithms from machine learning to automatically find repetitive patterns in a program's execution. By simulating one representative of each repetitive behavior pattern, simulation time could be reduced to minutes instead of weeks for standard benchmark programs, with very little accuracy cost. Several researchers have shown the SimPoint approach works well when exploring architecture designs with the same program binary [6, 11, 16, 17], but we have not yet seen a study focused on using SimPoint to compare the results of multiple binaries (compilations) of the same source code.

There are three main scenarios we have encountered where it is necessary to compare multiple binaries during architecture simulation. In these scenarios, we are using the *same* source code for a program, producing *different* binaries from the source, and running the binaries with the *same* input. The binaries are created using different compilers and/or different optimization levels. All three scenarios involve quickly evaluating architecture design decisions, which requires representative architecture simulation.

The first area deals with ISA extensions, where a new binary is created that uses some ISA extensions, such as the 64-bit x86 extensions. In this case, we must compare the performance of the original binary, which does not use the extensions, to the performance of the new binary, which does use the extensions. For example, one of the questions Intel architects want to answer is how their new processors will perform with 32-bit (IA32) and 64-bit (Intel64) binaries, and what is the difference in performance. This requires comparing the simulated performance of two different binaries. The second case deals with examining completely different architectures, such as Itanium and 64-bit x86. In this case, different compilers will be used, and it is important to identify the same parts of execution for the simulation samples. Finally, for a new architecture, the compiler team needs to evaluate the performance effects of compiler optimizations using simulation, before working prototypes of the processor are available. In this case, a compiler may use the same ISA but produce different binaries as optimizations are enabled, disabled, and reordered.

We consider two approaches for representative simulation for multiple binaries compiled from the same source. The first approach applies the prior SimPoint approach separately on each binary. SimPoint examines an execution trace and groups similar portions of execution into phases (clusters). The most representative interval from each phase is chosen as the simulation point to represent that cluster. This approach provides very accurate results when a single binary is used across different architectures, because the same simulation points are being simulated for each architecture, and each simulation point always represents the same portion of execution.

Using SimPoint with multiple binaries for a single program can result in different clusterings for each binary. This means that part of a program's execution in one binary may be assigned to a different phase in another binary for the same program, so phases may be weighted inconsistently. More importantly, the simulation points chosen in each binary might represent different behaviors. Results in Section 5.2.1 show the effects of these issues, which are especially important when determining which (binary, architecture) pair performs the best.

To address this issue, we propose a technique we call *Cross Binary SimPoint*. This approach finds simulation regions that are semantically the same across multiple binaries, and uses those regions to compare program performance. For this approach, we profile each binary with the input used for simulation, and identify a set of points in each binary that can be mapped to any other binary in the set. These *mappable points* are instructions in each binary corresponding to procedure calls and loop branches that can be consistently found in all of the binaries examined. These mappable points are potential boundaries for simulation regions. We break the execution intervals passed to SimPoint on these mappable points, and we use SimPoint to choose a set of simulation points we can map across all of the binaries. Then we use these mapped simulation points to compare performance across binaries.

The contributions of our paper are:

- We describe the need for a sampled simulation infrastructure that is accurate across multiple binaries for the same program.
- We examine the accuracy of the standard SimPoint approach when comparing the performance of multiple binary representations of a program.
- We present a new approach that picks mappable simulation points that can be identified across binaries of the same program. The simulation points may execute varying numbers of instructions, but they are semantically equivalent. This approach is accurate, and compares the same parts of execution across binaries, unlike the per-binary SimPoint approach.

2 Picking Per-Binary Simulation Points

In this section we give an overview of the current SimPoint 3.0 approach [2, 16], and how it is used to compare the performance of a given program across multiple compilations (binaries).

2.1 Breaking Execution into Intervals

An interval is a section of continuous execution (a slice in time) of a program's execution. All intervals are assumed to be non-overlapping, so SimPoint breaks a program's execution into contiguous non-overlapping intervals. The current SimPoint approach uses intervals of the same size. Intervals are measured by the number of instructions committed in an interval (e.g., interval sizes of 1, 10, or 100 million instructions were used in [13]). We call this the *Fixed Length Interval* (FLI) approach. This is the approach SimPoint promotes. In this section, we use this FLI approach with an interval size of 100 million instructions for picking per-binary simulation points.

SimPoint 3.0 provides support for *Variable Length Intervals* (VLIs), which allows intervals to represent different amounts of executed instructions as described in [4, 5]. Prior work has not proposed a good method to break a program into VLIs for architecture simulation. The approach we provide in this paper, examined in Section 3, provides the first usable approach for breaking programs into VLIs for SimPoint.

2.2 Basic Block Vectors

Each interval is represented by a frequency vector, which represents the program's execution during that interval. One type of frequency vector that is commonly used with SimPoint is the Basic Block Vector (BBV), which is a list of static basic blocks [15]. When tracking basic block usage with frequency vectors, SimPoint counts the number of times each basic block in the program has been entered in the current interval, and it records that count in the frequency vector, weighted by the number of instructions in the basic block.

Each element in the frequency vector is a count of how many times the corresponding basic block has been entered in that interval of execution, multiplied by the number of instructions in that basic block.

We use basic block vectors (BBV) for the results in this paper. The intuition behind this is that the behavior of the program at a given time is directly related to the code executed during that interval [15]. SimPoint uses the basic block vectors as signatures for each interval of execution: each vector tells us what portions of code are executed, and how frequently those portions of code are executed. By comparing the BBVs of two intervals, SimPoint can evaluate the similarity of the two intervals. If two intervals have similar BBVs, then the two intervals spend about the same amount of time in roughly the same code, and therefore we expect the behavior of those two intervals to be similar.

2.3 The SimPoint Approach

We now give a brief summary of how SimPoint works with fixed length intervals of basic block vectors to pick simulation points. The following steps summarize SimPoint's phase clustering algorithm at a high level. We refer the interested reader to [16] for a more detailed description of each step.

1. Profile the program by dividing the program's execution into contiguous intervals, and record a frequency vector for each interval. Normalize each frequency vector so that the sum of all the elements equals 1.
2. Reduce the dimensionality of the frequency vector data to a smaller number of dimensions using random linear projection.
3. Run the k-means clustering algorithm on the reduced-dimension data for a set of k values.
4. Choose from among these different clusterings a well-formed clustering that also has a small number of clusters. To compare and evaluate the different clusters formed for different values of k, SimPoint uses the *Bayesian Information Criterion* (BIC) [12] as a measure of the "goodness of fit" of a clustering to a dataset. SimPoint chooses the clustering with the smallest k, such that its BIC score is close to the best score that has been seen. The chosen clustering represents the final grouping of intervals into phases.
5. Select the simulation points for the chosen clustering. For each cluster, SimPoint chooses one representative interval that will be simulated in detail to represent the behavior of the whole cluster. By simulating *only* one representative interval per phase SimPoint can extrapolate and capture the behavior of the entire program. To choose a representative, SimPoint picks the interval in each cluster that is closest to the *centroid* (center) of each cluster. Each simulation point also has an associated weight, which reflects the fraction of executed instructions that cluster represents.
6. With the weights and the detailed simulation results of each simulation point, SimPoint computes a weighted average for the architecture metric of interest (CPI, miss rate, etc.). This weighted average of the simulation points gives an accurate representation of the complete execution of the program/input pair.

2.4 Using Per Binary Simulation Points

The focus of this paper is on architecture studies we have encountered that require us to compare multiple binaries for the same program (no source code changes) for the same input using detailed simulation.

The first approach we examine is to create a separate set of simulation points for each binary being compared for a program. This allows SimPoint to select a subset of the full execution that approximates the overall behavior of the full execution. This approach can be used to compare simulation results across binaries of the same program, but issues can arise, as described below.

SimPoint tries to capture the majority of behaviors during execution to create a faithful estimate of the complete execution, but it cannot capture every behavior. This creates bias (error). Prior work indicates the bias is low [6], but for studies involving multiple binaries, the bias across the different binaries examined must be consistent. Prior work has shown that this bias (relative error) is consistent for a single binary across many different architectures [13], but that work only considered the case where the same simulation points are used with the same binary across different architectures.

The focus of this work is to use *different binaries* (compiled from the same source) across *different architectures*. If we choose different simulation points for each binary, then the same behaviors may not be captured in the simulation points across the different binaries. This can result in different biases used for each binary, which can cause additional inaccuracies when trying to compare the performance of many architecture/binary combinations.

Related to this is a problem of representing all the unique behaviors a program exhibits with a small number of phases (each phase having a single simulation point representative). If there are more unique behaviors than allowed phases (since an architect may limit the number of simulation points used), then SimPoint cannot represent all the behaviors as separate phases. Therefore, some unique behaviors must be grouped into the same phase. If these groupings are not performed consistently across different binaries, then simulation points from one binary will represent different combinations of unique behaviors than simulation points from another binary. Simulation results using different simulation points can not be meaningfully compared, since the different simulation points may focus on different behaviors.

Both of these issues are addressed by the next approach we examine, which finds a single set of simulation points that can be used across all of the binaries, ensuring that the same behaviors are simulated for each binary.

3 Picking a Single Set of Cross-Binary Simulation Points

In this section we describe our approach for picking the same simulation point across a set of binaries for a program/input pair, and how we use these mappable simulation points and weight them appropriately.

3.1 Why Variable Length Intervals Are Necessary

In picking a single set of simulation points to represent execution across multiple binaries, we cannot just use the simulation points chosen from the fixed length approach described in the prior section. For that approach, the start of a simulation point is identified by when it occurs during execution, in terms of dynamic instruction count.

One problem is that a simulation point in binary A may start at dynamic instruction count X, but the semantically equivalent part of execution in binary B starts at dynamic instruction count Y (and $X \neq Y$). The other problem is that the semantically-equivalent sample for binary A may execute a different number of instructions than the same sample in binary B. Therefore, we cannot use dynamic instruction counts to identify the beginning and end of a sample. Instead, we must find samples whose boundaries correlate with source code so we can find the same sample across the execution of two different binaries.

3.2 Steps for Mappable Simulation Points

We now describe at a high level our mappable simulation point algorithm, which has the following steps:

1. **Create Call and Branch Profile for Each Binary:** Generate a profile for each binary for the input being examined. This along with symbol information will be used to find the set of mappable points.
2. **Find a Set of Mappable Points that Exist in All Binaries:** Use symbol information, the profile counts, and source line information to find a set of instructions in the binaries that exist in all of the binaries, and serve the same purpose (procedure entry points, loop back edges, loop entry points, etc).
3. **Create Variable Length Intervals Using Mappable Points:** Use the mappable points to partition execution for one input into variable length intervals (VLIs), where both the start and end of each VLI are mappable points. This allows us to accurately map the intervals across all of the binaries.
4. **Pick Simulation Points for the Primary Binary:** Pick a set of simulation points by running SimPoint on variable-length frequency vectors collected from one of the binaries (the “primary binary”).
5. **Map the Simulation Points to All Binaries:** Map the simulation points chosen from the primary binary to all of the other binaries, creating mappable simulation points across all of the binaries that semantically represent the same part of execution.
6. **Recalculate Weights for Mapped Simulation Points:** Simulation results for each SimPoint in each binary must be appropriately weighted by the size of its cluster in each binary.

We now go through each of these steps in more detail.

3.2.1 Create Call and Branch Profile for Each Binary

We generate a profile for each binary for the input being examined using Pin [9], a dynamic instrumentation system. We then find instructions in the binaries that are mappable; i.e. they exist across all binaries that mark the same exact point of execution. This will be used to locate the mappable points for each binary.

For each binary we profile all procedure entry points and loop branches and keep track of the total number of times each code structure is executed. For procedures we simply keep track of how many times each procedure is executed for the entire execution. Loops on the other hand can be considered as two entities: a loop entry point and the loop body. For the loop entry points we capture how many times the loop has been entered regardless of how many iterations the loop executes each time it is entered. This provides a coarse representation of loops, similar to procedures.

We also keep track of how many times the loop body executes. This is the number of times the loop has iterated over the entire execution. Each time a loop is entered we increment the loop entry count once and increment the loop body count by the number of times the loop iterates. The loop body count provides a much more detailed picture of the loop execution and is typically much larger than the loop entry count.

We want to break down loops like this so we can use either the entry point into a loop or specific loop iteration branches as mappable points. This provides a larger set of mappable points to choose from as will become apparent in the following step.

3.2.2 Find a Set of Mappable Points that Exist in All Binaries

We use the mappable points to partition the execution of a single input into variable length intervals (VLIs), where each interval starts and ends at a mappable point. This allows us to accurately map the intervals across all of the binaries.

The mappable points consist of (a) procedure entry points and (b) loop branches in the binaries. The notion is that if we can find the exact same loop branches and procedure entry points across all of the binaries then we can use these mappable points to define interval boundaries and pick simulation points which start and end at these mappable points. These simulation points can then be mapped to any other binary in the set of binaries considered.

For all the binaries being considered for a program, we first match up the procedure entry points with the same procedure names across all of the binaries, using debug symbol information. These procedure entry points represent the same exact point in execution across all of the binaries.

We also identify matching loop branches across all the binaries. For this we use two pieces of information: execution counts from the profiles collected in the previous step, and debug line number information associated with each branch. If the execution counts and line numbers for a branch match across all binaries, then that branch represents the same part of execution across all binaries.

For both procedure entry points and loop branches, the execution count across all binary versions must match. This guarantees that the mappable points will execute the same number of times across all binaries, which allows us to specify regions in the execution of any binary in the set by using mappable points as delimiters - for example, a simulation region can start at mappable point A after it has executed X times and end when mappable point B has executed Y times. This representation allows us to capture the same regions across the executions of different binaries.

3.2.3 Create Variable Length Intervals Using Mappable Points

We want to partition the execution into intervals that are close to a desired size specified by the user (e.g. 100 million instructions). For each interval we collect a basic block frequency vector which is given to SimPoint in the following step. Only one binary (the primary binary) is profiled in this step.

The mappable points (markers) selected in the prior step are used to break execution into variable length intervals. As the program executes, we keep track of how often each mappable marker is encountered, because any mappable marker could be used as an interval boundary. For example, if the desired interval size is 100 million instructions, and we have just executed 100 million instructions, we need to create an interval boundary on the next mappable marker we encounter. When the next mappable marker is reached, we record its marker ID and the number of times it has executed since the start of execution to bound the interval. We do this from the start of execution, ending intervals every time we reach the desired interval size at the next mappable marker encountered during execution. The execution count is critical, since markers can execute many times. Each (marker ID, execution count) pair uniquely identifies a specific point in execution that can be mapped to other binaries.

3.2.4 Pick Simulation Points for the Primary Binary

Next we run SimPoint on the basic block vectors collected for the mappable intervals from the primary binary to pick simulation points. We use SimPoint 3.0, which supports variable length intervals and considers the number of instructions in each interval during the clustering process and the search for simulation points.

For the primary binary, SimPoint generates simulation points, weights for each simulation point, and phase labels for every interval. Each simulation point represents a unique phase, and the weight associated with the simulation point reflects the fraction of executed instructions in that phase.

The primary binary can be selected arbitrarily from the set of binaries available, but it should be noted that interval sizes can expand or contract depending on which binary is chosen as the primary. One interval is created approximately every 100M instructions executed by the primary binary, so if the other binaries execute more or fewer instructions between interval boundaries, the mapped intervals can be bigger or smaller in the other binaries.

3.2.5 Map the Simulation Points to All Binaries

We next map the simulation points chosen from the primary binary to the other binaries, creating simulation points across all binaries that all represent the same part of execution.

The start and end of each simulation point is defined by a (marker ID, execution count) pair. This pair represents the simulation point across all binaries, and can be used during simulation to represent the start and end of that simulation point when executing each binary.

Because our simulation points are defined by mappable markers, nothing needs to be done in this step.

3.2.6 Recalculate Weights for Mapped Simulation Points

Finally, we need to appropriately weight the simulation points relative to the size of the clusters for each binary. Weights must be readjusted because the amount of execution in each phase can change across binaries.

A simulation point's weight is the fraction of the total dynamic instructions that the program executes in the phase it represents. For example, if a program executes 60% of its dynamic instructions in phase P, the simulation point for phase P will have a weight of 60%. We calculate the correct weight for each simulation point in each binary by running each binary, and counting the number of dynamic instructions executed in each phase.

3.3 Dealing With Optimized Code Regions

Compiler optimizations may modify a binary's call-loop structure. A procedure that has been inlined in the optimized version of a binary will not be mappable, since it will no longer have the procedure name and entry point associated with it. Although the process mentioned above for mapping points across the binaries does not handle this case, we have extended the mapping of points to handle some of these optimization cases.

We can detect inlined procedures by their parent nodes and the loop structure within the procedure. Consider a procedure that has a loop that executes N times, which is called from another procedure that has a loop that executes M times. If this procedure has been inlined and its loop structure maintained, we expect the caller to now have two loops, executing N and M times respectively. We can still map the loop of the inlined procedure because we can identify it based on its call count. Of course, if $N = M$, we can not determine which loop belongs to the inlined procedure based on the call counts.

Additional techniques can be done to map optimized code, and that is part of future research.

4 Methodology

We evaluate our approach for selecting cross binary simulation points using CMP$im [3], a Pin [9] based multi-core simulator. CMP$im models an in-order processor and can simulate the performance of applications run to completion. CMP$im is configured to model a single-core processor with a three-level non-inclusive cache hierarchy with parameters as shown in Table 1. All caches use a 64B line-size and LRU replacement policy.

To evaluate our approach, we compiled the SPEC2000 programs with debug information (-g compiler flag) on 32-bit (x86) and 64-bit (x86_64) Linux. The programs were compiled using version 9.0 of Intel's C/C++ and Fortran compilers. For each program we also compiled unoptimized and optimized versions, for a total of four binaries per SPEC program: 32-bit Optimized, 32-bit Unoptimized, 64-bit Optimized, and 64-bit Unoptimized. We then compare the performance of these binaries and examine how well the SimPoint based techniques estimate the speedup between the different binaries. We selected a subset of benchmarks from the SPEC2000 suite that would provide a representative sample and also include a wide range of programs with interesting behaviors. For each of the programs we selected we provide results using the reference inputs.

Simulation regions are represented with PinPoints files [11], which is a Pin tool chain that generates basic block vectors for each interval and then runs them through SimPoint 3.0 to get the simulation points and weights. We ran each binary under CMP$im configured as above with the PinPoints file describing the simulation regions for the binary for the given input. Using statistics (reported by CMP$im) and weights (reported by SimPoint) for each simulation region, we compute a prediction for whole-program statistics and compare the results of the prediction to the actual whole-program statistics reported by CMP$im.

5 Results

In this section we evaluate SimPoint's performance estimates across different binaries compiled from the same program source. We show that our proposed mappable SimPoint technique is an improvement over the per-binary SimPoint approach because it allows us to simulate the same regions across different binaries.

5.1 SimPoint Performance Estimation

SimPoint can be run with different configurations which may result with different simulation points being selected. To fairly compare the two SimPoint methods we used the same SimPoint configurations for both techniques.

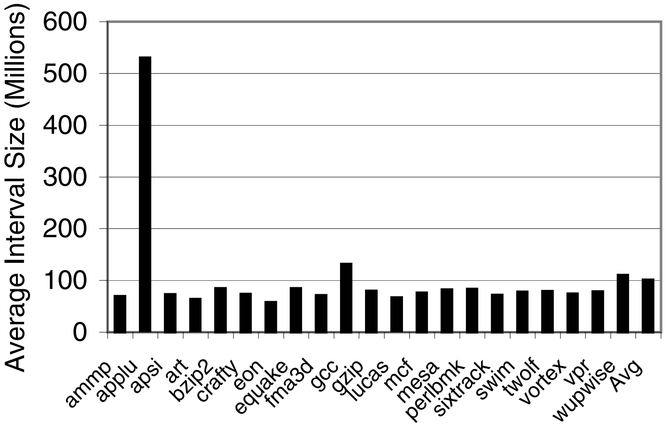

Figure 2: Interval Size for mappable SimPoint (VLI). Each bar shows the average across all four binaries. The size of each interval in per-binary SimPoint (using FLIs) is constant at 100 million instructions.

We limited SimPoint's maximum number of clusters to 10. This is an upper limit on the number of clusters SimPoint can use to characterize the phase behavior of a program. One simulation point is generated for each cluster (phase) identified. SimPoint generally picks fewer simulation points than the upper limit because it usually finds a good phase characterization with fewer clusters.

Figures 1 and 2 show the number of simulation points picked and the average interval size respectively for each benchmark we examined. Figures 1 shows results for per-binary SimPoint (FLI) and mappable SimPoint (VLI), while Figure 2 only shows results for mappable SimPoint, because the interval size for per-binary SimPoint is fixed at 100 million instructions. We compiled four different binaries for each benchmark, and for each benchmark we are showing the average across these four binaries.

Figure 1 shows that both techniques select a similar number of simulation points on average. This is expected since the binaries all represent the same program, so we are still observing the same behaviors.

To understand the interval size differences between per-binary SimPoint and mappable SimPoint shown in Figure 2, recall that per-binary SimPoint and mappable SimPoint split executions into intervals differently. Per-binary SimPoint splits every execution of a program binary into fixed length 100 million instruction intervals, while the mappable SimPoint approach produces intervals of at least 100 million instructions. In mappable SimPoint, an interval ends only when a mappable marker is reached, so intervals can be larger than 100 million instructions.

In addition, mappable SimPoint constructs intervals from the execution of one binary and maps the intervals to the other binaries. The same interval in another binary may not execute

Cache Level	Capacity	Associativity	Line Size	Hit Latency	Type
FLC(L1D)	32KB	2-way	64 bytes	3 cycles	WriteBack
MLC(L2D)	512KB	8-way	64 bytes	14 cycles	WriteBack
LLC(L3D)	1024KB	16-way	64 bytes	35 cycles	WriteBack
DRAM				250 cycles	

Table 1: Memory System Configuration

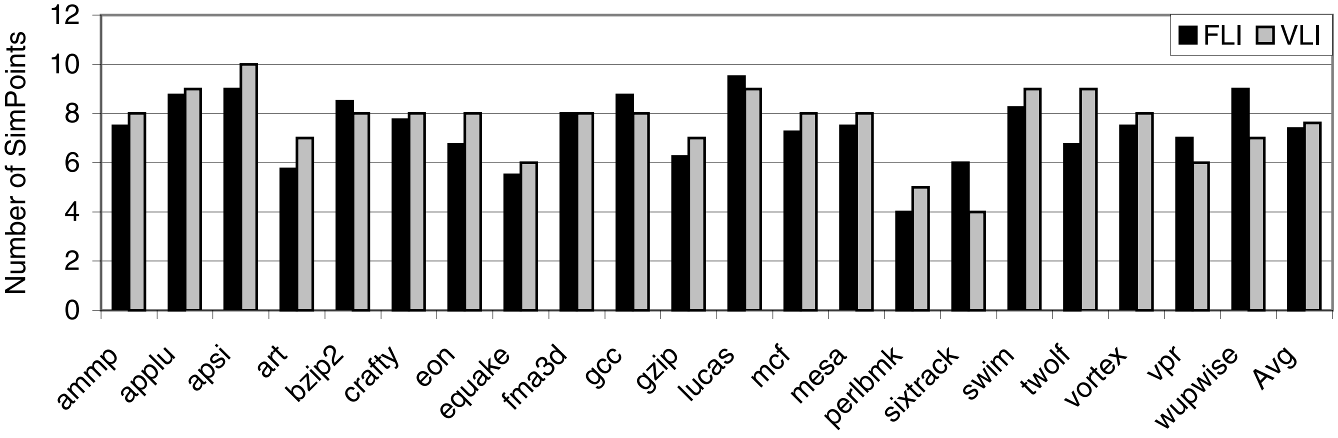

Figure 1: Number of SimPoints for per-binary SimPoint (FLI) and mappable SimPoint (VLI). Each bar shows the average across all four binaries.

the same number of dynamic instructions. Suppose, for example, that an unoptimized binary executes 10 times more instructions than an optimized binary. If we use the unoptimized binary as the primary binary, we will construct mappable 100 million instruction intervals, but when the intervals are mapped to the optimized binary, the intervals will shrink to 10 million instructions on average.

This is why we see a smaller average simulation point size in Figure 2. The intervals we constructed for the primary binary, when mapped to the other binary versions became smaller in most cases.

applu has a much larger interval size because our technique was unable to find mappable markers across all four binaries for large execution regions. In these execution regions, a loop calls five procedures that each solve a partial differential equation. Each of the five procedures has a similar looping structure since they are doing a similar operation. In the optimized version of this binary, all five procedures are inlined into the loop. Furthermore, the loops were split by the optimizer, and code was moved within this loop. While our technique can deal with simple cases of inlining, in this case there was not enough structure left after optimization to map the optimized code to the unoptimized code. Our approach can be extended to handle more of these optimization cases with more powerful mapping techniques, and this is part of our future work.

Figure 3 shows the relative error in estimated CPI for each benchmark. As in the previous figure, due to space constraints each bar in this graph is the average across four binaries. For each binary we calculate the CPI error for that binary using the simulation points compared to a full simulation of the program. We then averaged this CPI error across the four binaries for the results shown. On average we see that both techniques accurately estimate the performance of the programs. The per-binary technique accurately estimate the performance for each binary when compared to the full execution of that binary, using a different set of simulation points for each binary. Figure 3 also shows that mappable simulation points achieve accurate performance estimates, but the figure does not show that the biases are consistent in the errors across the binaries, which is the focus of the next result.

5.2 Speedup Comparison

When using sampled simulation for design space exploration, it is very important to have a consistent bias across the experiments to make meaningful performance comparisons. Here we calculate the actual speedup between different binaries and compare how the two SimPoint methods perform in estimating the speedup. We find that the mappable SimPoint method has a more consistent bias than the per-binary SimPoint method, and is more accurate when comparing results across different binaries.

Figures 4 and 5 show the error in speedup estimation across several binary pair configurations. Figure 4 shows bi-

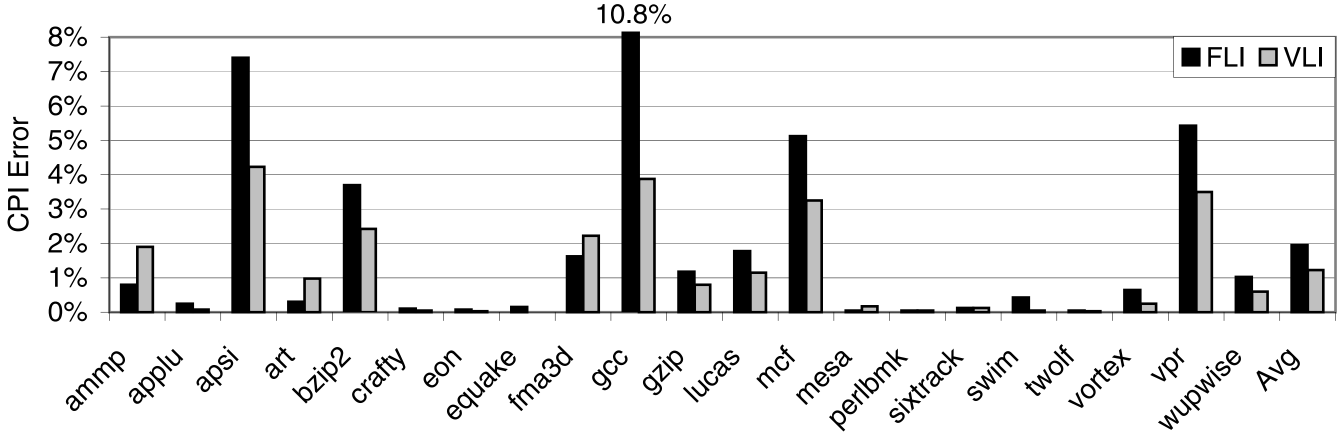

Figure 3: CPI Error for per-binary SimPoint (FLI) and mappable SimPoint (VLI). Each bar shows the average across all four binaries.

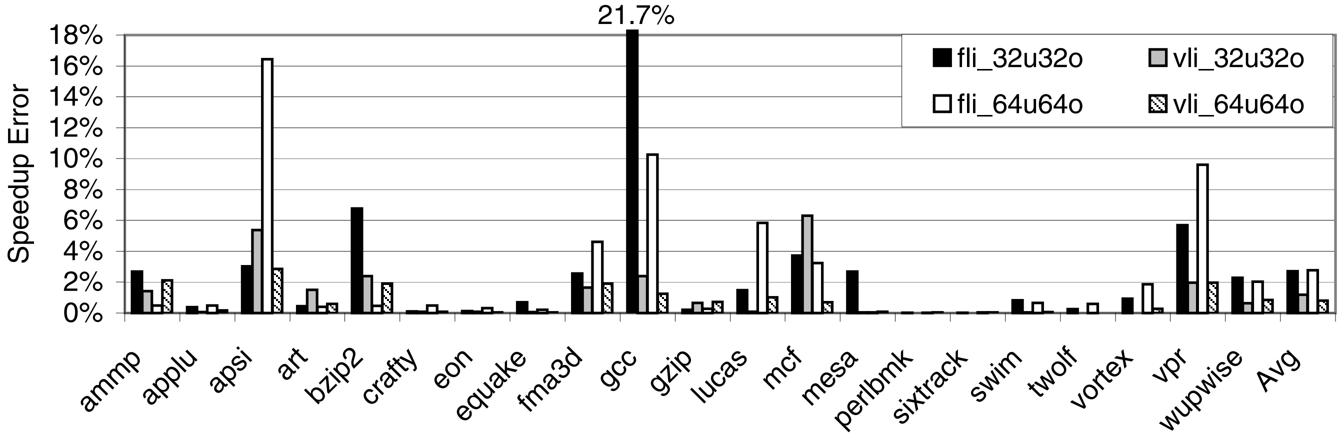

Figure 4: Speedup error for per-binary SimPoint (fli) and mappable SimPoint (vli). Speedup is computed across different binary pair configurations on the same platform and the error is based on how closely the estimated speedup is to the true speedup. 32U is 32-bit Unoptimized, 32O is 32-bit Optimized, etc.

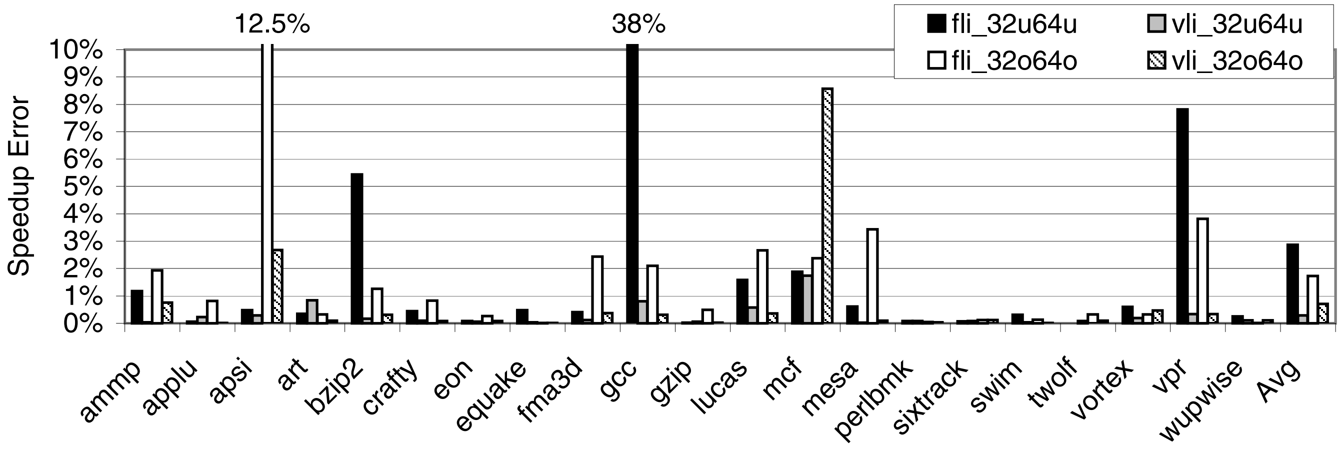

Figure 5: Speedup error for per-binary SimPoint (fli) and mappable SimPoint (vli). Speedup is computed across different binary pair configurations across different platforms (32-bit and 64-bit) and the error is based on how closely the estimated speedup is to the true speedup. 32U is 32-bit Unoptimized, 32O is 32-bit Optimized, etc.

nary pair configurations on the same platform varying the optimization levels, while Figure 5 shows binary pair configurations across platforms for the same optimization level. Each figure shows how closely the estimated speedup of either per-binary or mappable SimPoint is to the true speedup. We compute the error in speedup as the following: $|(TrueSpeedup - EstimatedSpeedup)/TrueSpeedup|$. The $TrueSpeedup$ is computed as the ratio of total cycles executed for two binary versions. For example, the $TrueSpeedup$ for $32u32o$ configuration is the ratio of the number of cycles executed with the 32-bit unoptimized version and the 32-bit optimized version. The $EstimatedSpeedup$ is computed just like the $TrueSpeedup$ but instead of using the true number of cycles that execute we are estimating the number of cycles using sampled simulation. The error in speedup tells us how close our speedup estimates are to the actual speedup seen between the two binaries.

For each benchmark in these figures we show 4 pairs of configurations for speedup analysis: 32-bit unoptimized to 32-bit optimized and 64-bit unoptimized to 64-bit optimized in Figure 4 and 32-bit unoptimized to 64-bit unoptimized and 32-bit optimized to 64-bit optimized in Figure 5. For each configuration we estimate the speedups using the per-binary and mappable SimPoint methods and compute the speedup error as described above.

The results in these figures show that mappable SimPoint results in a lower error in speedup estimation on average than per-binary SimPoint, for the binaries we examine. This result can be explained by the lack of behavioral consistency in samples that are chosen as simulation points across the different binaries for the per-binary approach. Whenever we use simulation on a small portion of a program to estimate the performance of the whole program, there will be some unavoidable error for behaviors that are not well-represented in the simulation. We call this a bias in the simulation, and it occurs because it is impossible to simulate a behavior that is not represented by a simulation point. Because mappable SimPoint uses the same execution regions across different binaries, errors in performance estimation due to lack of representation will occur consistently across all the binary executions. Thus the error that occurs due to bias is consistent across all our estimates. This consistency allows us to obtain performance estimates that are more accurate when comparing performance across binaries, allowing us to make better design decisions.

5.2.1 Phase Bias Comparisons

In the per-binary SimPoint approach we are picking a different set of simulation points for each binary version. Each set of simulation points will be accurate in representing the overall execution for that binary. However, a particular program behavior may be more representative in one binary (since a simulation point may be chosen directly from that behavior), and less representative in another binary. Thus the unavoidable error due to using a small fraction of program execution to represent the whole program execution will not be consistent across all the binary versions using the per-binary SimPoint approach. Per-binary SimPoint can give semantically similar simulation points that represent common program behaviors across binaries, but it is not guaranteed.

As a particular example of the benefits of consistent bias for making design decisions, we consider two benchmarks in detail: `gcc` and `apsi`. Both benchmarks have higher speedup error using per-binary SimPoint than our mappable SimPoint technique. Tables 2 and 3 compare phase statistics across two binary versions for `gcc` and `apsi` respectively. Table 2 compares the largest three phases found with per-binary SimPoint and mappable SimPoint across 32-bit unoptimized and 64-bit unoptimized `gcc` binary versions. Table 3 compares the largest three phases found with per-binary SimPoint and mappable SimPoint across 32-bit optimized and 64-bit optimized `apsi` binary versions. Both tables show for each phase the phase ID, the weight of the phase (the percentage of executed instructions in that phase), the true CPI of the phase (the average CPI across all intervals in that phase), the estimated CPI using the SimPoint techniques, and the relative error between the true CPI and the SimPoint CPI.

Table 2 shows the problem of picking simulation points for each binary using the per-binary (FLI) approach. For `gcc` we see that the weights for the three phases for FLI changes from 36% to 22% for phase 1, then from 31% to 18% for phase 2, and then from 9% to 16% for phase 3. This shows that for the 64-bit binary, a large portion of execution is grouped into different phases compared to the 32-bit binary. This is further shown by the significant changes in CPI error for the phases between the two binaries. These weights and CPI errors are the bias that SimPoint introduces. This bias is perfectly fine to have when using a single binary to compare options across a design exploration, because the bias is consistent and does not change.

When different binaries are used to explore a design space, the biases can change, but they must be consistent. Table 2 shows that per-phase biases can change significantly between the binaries when using the per-binary (FLI) SimPoint approach. For example, when using FLI the second phase in `gcc` has an error of 56% for the 32-bit binary and -17% for the 64-bit binary. Similarly the 32-bit binary has 19% error and the 64-bit binary has -8% error. This change in bias is the reason for the 38% error in speedup for `gcc` in Figure 5.

In comparison, the mappable (VLI) SimPoint approach proposed in this paper has a consistent bias across the phases. This is because the simulation points chosen across the binaries represent the same part of execution. The weights have slightly changed for VLI, but this is to be expected due to differences in compilation. Similar results can be seen for `apsi` in Table 3. For `apsi` the bias for the per-binary FLI approach for phase 2 changes from -0.7% to 37%, whereas the bias is kept consistent across the phases for our mappable SimPoint approach.

	gcc/166									
	32-bit Unoptimized					64-bit Unoptimized				
	Phase	Weight	True CPI	SP CPI	CPI Error	Phase	Weight	True CPI	SP CPI	CPI Error
VLI	1	0.35	3.16	3.15	0.2%	1	0.28	2.97	2.97	-0.1%
	2	0.26	3.99	2.93	27%	2	0.21	4.11	2.93	29%
	3	0.14	4.47	5.17	-16%	3	0.17	5.49	6.34	-16%
FLI	1	0.36	3.16	3.16	0%	1	0.22	2.98	2.97	0.5%
	2	0.31	6.54	2.90	56%	2	0.18	6.04	7.04	-17%
	3	0.09	5.00	4.04	19%	3	0.16	6.66	7.19	-8.0%

Table 2: Phase comparison across 32-bit unoptimized and 64-bit unoptimized gcc *binary versions*

	apsi/ref									
	32-bit Optimized					64-bit Optimized				
	Phase	Weight	True CPI	SP CPI	CPI Error	Phase	Weight	True CPI	SP CPI	CPI Error
VLI	1	0.52	3.04	2.91	4.5%	1	0.52	2.59	2.44	5.9%
	2	0.19	3.57	3.10	13%	2	0.18	3.16	2.66	16%
	3	0.05	4.66	4.70	-0.9%	3	0.05	3.64	3.63	0.3%
FLI	1	0.71	3.50	3.00	14%	1	0.65	2.77	2.50	0.9%
	2	0.05	4.58	4.61	-0.7%	2	0.08	5.34	3.39	37%
	3	0.05	4.60	4.63	-0.7%	3	0.06	7.61	7.55	0.8%

Table 3: Phase comparison across 32-bit optimized and 64-bit optimized apsi *binary versions*

Finally, we want to emphasize that the error seen for SimPoint for a given phase in Tables 2 and 3 is to be expected. The error can occur because the single simulation point used for the phase did not represent all of the behavior in that phase, just the majority of the behavior for the phase. SimPoint is used to find a small set of the most representative behaviors, and because of this not every behavior can be appropriately represented. From our several years of using SimPoint, our experience has shown that the majority of behaviors will be captured, and this allows us to perform accurate architecture design comparisons. This is achievable when using a single binary for design exploration, since the same simulation points are used, which results in a consistent bias and error across the architectures examined. When using multiple binaries, our mappable SimPoint is needed in order to capture the same representative part of execution for the simulation points across the different binaries. This maintains a consistent bias and error for the cross binary simulation points.

6 Related Work

We now briefly compare our approach to prior techniques in simulation that use procedure and call boundaries to define intervals of program execution.

Huang et al. [8] considered procedures and loops to partition a program's execution. The partitioning determined where and when statistical samples should be taken during architecture simulation. Their analysis broke up a program's execution at static call sites, and if a procedure executed for too long, they divided the procedure's execution into its major loops. To determine the sample rate, they examine the variability of several architecture metrics for each program region.

Shen et al. [14] used Wavelets [1] and Sequitur [10] on a trace of data reuse distances to build a hierarchy of phases which reflect the program's behavior patterns. They identify basic blocks that indicate the start of each phase when executed, but they do not consider the problem of finding markers that can be mapped across binaries.

Another study by Lau et al. [4] presented an automated profiling approach to identify code constructs (branches, procedure calls and returns) that indicate phase changes when executed. They built a hierarchical procedure call-loop graph to represent a program's execution, where each edge also tracks the average hierarchical execution variability on paths from that edge. They used the call-loop graph to identify branch points with low variability, and then examined splitting the program's execution at those points. This created intervals with very similar phase behavior aligned to the natural phase behavior found in the code, but the interval sizes were too large to use for simulation. In addition, two other studies similarly examined breaking up sample intervals at loop and procedure call boundaries [5, 7], with similar results.

Our approach focuses on a simulation scenario not addressed in any of the above research. All of the above research examined only applying their approach to one binary for a single program/input combination, whereas we are focusing on how to effectively compare multiple binaries for a single program/input combination. The difference in our approach is that we need to (a) perform analysis to find mappable points across all of the binaries, even in the face of compiler optimizations, whereas the prior techniques break intervals at any arbitrary branch point, and (b) we have to correctly calculate the weights of the mapped simulation points for each binary, which the prior techniques did not have to deal with.

7 Summary

Researchers testing a new ISA extension, examining a new architecture, or trying a new compiler optimization may need to analyze and evaluate performance across different binaries of a program. Due to the slow nature of performance simulators, it has become a standard practice to use representative sampling simulation techniques. In this paper we examined two approaches for simulation when there are multiple binaries for a single program/input.

The first approach simply applies the existing SimPoint approach separately on each binary, creating a different set of simulation points for each binary. This approach can accurately estimate the performance for each binary by using different simulation points for each binary, but the approach can have significant error when comparing performance across binaries, since the different simulation points may emphasize different behaviors.

The second approach identifies simulation points that represent the same behaviors across all binary representations of a program. This allows us to simulate the same parts of execution as we change the ISA or compiler optimizations during design space exploration. Our approach finds phase transitions during execution that are identifiable in all of the binaries considered. We use these phase markers along with SimPoint to pick simulation points to represent the full execution of the program, and to identify the exact same start and end of execution for the simulation points in each binary. Our results show that this method does not suffer from changing biases that can occur with the first approach, so cross-binary simulation points can be used to accurately compare performance across binaries.

Acknowledgments

We would like to thank the anonymous reviewers and Leslie Barnes for providing helpful comments on this paper. This work was funded in part by NSF grant No. CCR-0311710, NSF grant No. ACR-0342522, UC MICRO grant No. 03-010, and a grant from Intel and Microsoft.

References

[1] A. Cohen and R. D. Ryan. *Wavelets and Multiscale Signal Processing*. Chapman & Hall, 1995.

[2] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7, Sept. 2005.

[3] A. Jaleel, R. S. Cohn, C. Luk, and B. Jacob. Cmp$im: A binary instrumentation approach to modeling memory behavior of workloads on cmps. Technical Report UMDSCA-2006-01, Intel, Jan. 2006.

[4] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *International Symposium on Code Generation and Optimization*, Mar. 2006.

[5] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

[6] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[7] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2004.

[8] W. Liu and M. Huang. EXPERT: Expedited simulation exploiting program behavior repetition. In *International Conference on Supercomputing*, June 2004.

[9] C. 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[10] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. In *The Computer Journal vol. 40*, 1997.

[11] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture*, Dec. 2004.

[12] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, San Francisco, CA, 2000.

[13] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[14] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[15] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[17] J. Yi, S. Kodakara, R. Sendag, D. Lilja, and D. Hawkins. Characterizing and comparing prevailing simulation techniques. In *International Symposium on High-Performance Computer Architecture*, Feb 2005.