Representative Multiprogram Workloads for Multithreaded Processor Simulation

Michael Van Biesbrouck[†] Lieven Eeckhout[‡] Brad Calder[†*]

[†]CSE, University of California, San Diego, USA
[‡]ELIS, Ghent University, Belgium
[*]Microsoft
Email: {mvanbies,calder}@cs.ucsd.edu, leeckhou@elis.UGent.be

Abstract

Almost all new consumer-grade processors are capable of executing multiple programs simultaneously. The analysis of multiprogrammed workloads for multicore and SMT processors is challenging and time-consuming because there are many possible combinations of benchmarks to execute and each combination may exhibit several different interesting behaviors. Missing particular combinations of program behaviors could hide performance problems with designs. It is thus of utmost importance to have a representative multiprogrammed workload when evaluating multithreaded processor designs.

This paper presents a methodology that uses phase analysis, principal components analysis (PCA) and cluster analysis (CA) applied to microarchitecture-independent program characteristics in order to find important program interactions in multiprogrammed workloads. The end result is a small set of co-phases with associated weights that are representative for a multiprogrammed workload across multithreaded processor architectures. Applying our methodology to the SPEC CPU 2000 benchmark suite yields 50 distinct combinations for two-context multithreaded processor simulation that researchers and architects can use for simulation. Each combination is simulated for 50 million instructions, giving a total of 2.5 billion instructions to be simulated for the SPEC CPU2000 benchmark suite. The performance prediction error with these representative combinations is under 2.5% of the real workload for absolute throughput prediction and can be used to make relative throughput comparisons across processor architectures.

1 Introduction

Most of today's processors run multiple programs simultaneously through Simultaneous Multithreading (SMT) or multicore processing. Some programs are designed to run identical threads simultaneously for CPU-intensive tasks, but most users will run a heterogeneous set of programs and individual programs may run many threads that accomplish distinct tasks. For example, a web browser and its helper applications may be simultaneously parsing HTML, decoding JPEG images, decompressing gzip-encoded web pages, playing audio and showing Flash animations while other programs run on the same system. A thorough analysis of a new system requires understanding the consequences of running all combinations of these threads and programs.

Most computer architecture studies require cycle-accurate simulation of real-life programs to help evaluate new architectural features. Current industry-standard benchmarks, such as SPEC CPU2000 execute hundreds of billions of instructions; SPEC CPU2006 has programs that run for trillions of instructions. Even on today's fastest architectural simulators, simulating some of the SPEC CPU2000 benchmarks takes several weeks to complete. Because the processors to be simulated are becoming more complex and the length of the benchmarks to be simulated is increasing, simulation time is increasing even though the computers running the simulations are becoming faster.

One common technique to reduce simulation time is *representative sampling* [15, 11]. Phase analysis groups instruction intervals from a program's execution into phases that have similar behavior, and simulate only one sample from each phase. This technique is normally applied to single programs, so it does not capture the similarities between different benchmarks, nor the effects of running multiple programs at the same time.

On a multithreaded or multicore processor, the number of distinct behaviors that can occur from a set of programs is a combination of the number of unique phases in each application. To try to capture all of those behaviors, Van Biesbrouck *et al.* [18] proposed the *Co-Phase Matrix* to guide the simulation of an SMT processor for a multiprogrammed workload. The co-phase matrix represents all of the potential phase combinations, called a co-phase, of a multiprogrammed workload to be ex-

simulation approach populates the co-phase matrix with samples during simulation. Once a co-phase has an appropriate sample, they no longer need to simulate that co-phase and can just fast-forward execution to the next co-phase. The amount to fast-forward for each thread is determined by the performance data stored in the co-phase matrix for that particular co-phase. This allows the performance of a pair of programs to be estimated from any given pair of starting points.

A later paper by Van Biesbrouck *et al.* [17] extends the co-phase technique to efficiently estimate average performance over all starting points when running multiple programs on an SMT processor. They construct the co-phase matrix as before, but use it to estimate the average performance from many starting points. The starting points are randomly picked and they analytically simulate from each starting point. The average of the collected performance statistics converges after on the order of 1000 runs. Since the analytical simulation is done very efficiently, the whole SMT simulation for a set of starting points completes in minutes. The drawback to this technique is that constructing co-phase matrices for every pair of programs is time-consuming, and the number of instructions to be simulated for a workload can be more than a researcher has time to analyze when doing design space exploration. Analyzing all pairs of reference inputs for the whole SPEC CPU2000 suite would require 3.5 trillion instructions to be simulated.

In this paper we propose a technique to determine the most significant co-phase behaviors in a benchmark suite. To make this possible, we propose a technique based on *Principal Components Analysis* (PCA) and *Cluster Analysis* for reducing the required number of *co-simulation points*. The proposed technique detects similarities in phases from different benchmarks and different inputs to the same benchmark. Additional savings come from the elimination of rare behaviors that do not contribute significantly to performance estimates and by avoiding fine-grain simulation of co-phase behavior. Despite small number of co-simulation points — 50 co-simulation points for two-thread workloads for SPEC CPU2000, or 2.5 billion instructions in total — we are able to accurately estimate the throughput of all benchmark combinations.

Our approach uses long enough samples to avoid most warmup problems. In addition, we insure that the samples contain homogeneous behavior relative to the sample size. Homogeneous behavior within a sample allows for faithful comparisons of simulation results across processor architectures. The reason is that when comparing performance numbers across multithreaded processor architectures, the various threads may have different progress rates. Homogeneous behavior allows for stopping the simulation at any point while yielding representen-

Our technique is scalable to architectures with many cores and contexts per core. Improvements to the clustering methodology allow us to use many threads when identifying thread combinations to simulate. Furthermore, due to our use of interpolation, the number of combinations to simulate can be kept small; we examine the use of just 50 co-simulation points.

2 Prior Work

2.1 Multithreaded Simulation Methodologies

Most of the research done on multithreaded processors uses an ad-hoc simulation methodology. Researchers typically pick a limited number of arbitrary samples (or in many cases just a single sample) from a randomly chosen set of benchmarks. Then they simulate these randomly-picked samples together. An important pitfall with this methodology though is that it is unlikely that this reduced workload gives a faithful image of the real behavior that is observed on real multithreaded processors. There are two reasons for this. For one, the randomly-picked samples may not be representative for the real program behavior. And second, when running these randomly-picked samples together, there are no weights available to combine the performance numbers for each of these co-samples into a meaningful performance number for the overall workload.

Raasch and Reinhardt [14] used an improved SMT simulation methodology in their study on how partitioned resources affect SMT performance. They selected a set of diverse co-sample behaviors rather than randomly chosen co-sample behaviors. First, they find single simulation points using SimPoint [15]. They then run all possible two-context co-phase combinations on a given microprocessor configuration — in their setup they ran 351 co-phases. For each of those co-phases, they compute a number of microarchitecture-dependent characteristics such as per-thread IPC, ROB occupancy, issue rate, L1 miss rate, L2 miss rate, functional unit occupancy, *etc.* Using the methodology from [6], they then apply principal components analysis (PCA) and cluster analysis (CA) to come to a limited number of 15 two-context co-phases. There are at least three pitfalls with this methodology. First, a single simulation point is chosen per benchmark. This could give a distorted view for what is being seen in a real system where programs go through multiple phases. Second, the single simulation points selected by SimPoint may represent heterogeneous program behavior which makes comparing co-phase behavior across processor architectures questionable — different portions of the workload may be executed under different processor architectures. To address

homogeneous phases. Third, this approach is driven by microarchitecture-dependent characteristics. As a result, the distinct co-phase behaviors obtained through PCA and CA will be representative for the processor architecture for which the characteristics were measured. However, it is questionable whether these co-phases will be representative when applied to other processor architectures. In this paper, we address this pitfall by considering microarchitecture-independent characteristics for determining representative co-phases.

Van Biesbrouck *et al.* [18] proposed the co-phase SMT simulation approach for accurately predicting SMT performance where each thread starts at a specific starting point. They keep track of the performance data of previously executed co-phases in a co-phase matrix; whenever a co-phase gets executed again, the performance data is picked from the co-phase matrix without re-simulating the co-phase. By doing this, each unique co-phase gets simulated only once, which greatly reduces the overall simulation time. In their follow-on work [17], Van Biesbrouck *et al.* noted that in order to make an accurate estimate of SMT performance it is important to consider multiple starting points. As such, they extended the co-phase matrix approach to consider all possible starting points. They simulate all possible co-phases once, store these performance numbers in the co-phase matrix and then perform analytical simulations using the co-phase matrix for multiple starting points. Note that this prior work by Van Biesbrouck *et al.* focused on estimating SMT performance when running just single pairs of threads; in other words, they are focusing on providing an accurate performance estimate when running threads simultaneously without considering the performance of entire workloads. The focus of this paper is on finding what threads to run simultaneously in a reduced simulation workload that is representative for a much larger workload consisting of multiple programs.

Alameldeen and Wood [1] showed that for multi-threaded workloads running on real systems, performance can be different for different runs from the same initial state. This variability is not modeled in deterministic simulations. To account for this variability, they argue to inject randomness into the simulation environment, and to apply statistics for making statistically rigorous conclusions.

Ekman and Stenström [7] use random sampling for simulating multiprocessor systems and they use the well-established matched pair statistical method to show that the variability in the system's throughput decreases with an increasing number of processors when running multiprogrammed workloads. As a result, fewer samples need to be taken in order to estimate overall system performance on a multiprocessor system through sampled simulation. However, they do not provide a method for representative multiprogrammed workload.

2.2 Workload Analysis Through PCA

Eeckhout *et al.* [6] proposed a workload reduction approach that picks a number of program-input representatives from a large set of program-input pairs. They first measure a number of program characteristics of the complete execution for each program-input pair. They subsequently apply principal components analysis (PCA) in order to get rid of the correlation in the data set. (In section 3.4 we will discuss why this is important.) As a final step, cluster analysis (CA) computes the similarities between the various program-input pairs in the rescaled PCA space. Program-input pairs that are close to each other in the rescaled PCA space exhibit similar behavior; program-input pairs that are further away from each other are dissimilar. As such, these similarity metrics can be used for selecting a reduced workload. For example, there is little benefit in selecting two program-input pairs for inclusion in the reduced workload if both exhibit similar behavior.

This initial work on workload reduction used microarchitecture-dependent and microarchitecture-independent characteristics as input to the workload analysis. The main disadvantage of using microarchitecture-dependent characteristics is that it is unclear whether the results are directly applicable for other microarchitectural configurations. Phansalkar *et al.* [13] made a step forward by choosing microarchitecture-independent characteristics only. This makes the reduced workload more robust across different microarchitectures. Eeckhout *et al.* [5] further extended this workload analysis approach by looking into similarities between program-input pairs at the phase level. Instead of measuring aggregate microarchitecture-independent metrics over the complete benchmark execution, they measure those metrics at the phase level. They subsequently used the PCA/CA workload analysis methodology to identify similarities across the benchmarks and inputs at the phase level. The end result from their analysis is a set of representative phases across the various program-input pairs; these phases along with an appropriate weighting, allow them to make accurate performance estimates of the complete benchmark suite.

All of this prior work focused on finding representative workloads for single-threaded processor simulation. This paper uses and extends the PCA/CA workload analysis methodology based on microarchitecture-independent metrics to select representative co-phases to be simulated in an accurate multiprogrammed multi-threaded processor simulation methodology.

sentative Workload

We now present our methodology for building a reduced but representative multiprogrammed workload for driving the simulation of multithreaded processors. The workload that we start with is a set of benchmarks that the computer architect considers a viable set of benchmarks. The set of benchmarks that we consider in this paper is the SPEC CPU2000 benchmark suite. The overall goal of the methodology proposed in this paper is to reduce this workload into a workload that is viable for simulation purposes while being representative for the multiprogrammed behavior that is to be expected with the original workload. Our workload reduction methodology consists of four steps.

1. The first step determines the prominent phase behaviors for each of the benchmarks in the original workload. The prominent phase behaviors are represented by simulation points — a simulation point is represented by a position in the dynamic instruction stream where the phase behavior starts. In practice, a simulation point is stored on disk using checkpoints [16]. The number of simulation points is limited to a few per benchmark. The goal is for each simulation point to exhibit homogeneous program behavior.

2. For each of these simulation points, we then measure a number of microarchitecture-independent characteristics. This is done very efficiently through profiling.

3. Next, principal components analysis (PCA) [10] is applied to a data set that represents all of the simulation points. The goal of principal components analysis is to identify similar phases based on their microarchitecture-independent behavior.

4. In the final step, cluster analysis (CA) is applied to the results of the PCA in order to find groups or clusters of *co-phases* (combinations of simulation points) that exhibit similar microarchitecture-independent behavior. A representative co-phase is then chosen for each cluster and each co-phase is represented as a weighted sum of neighboring representative co-phases. The set of representative co-phases and sum of associated weights then constitute the reduced workload.

We now discuss each of these four steps in greater detail in the following subsections.

3.1 Finding Homogeneous Intervals

The first issue we need to address is how we end our multiprogram simulation samples, since during simulation workload will have different rates of progress yet we do not want the nature of the workload to change due to a change in relative rates of progress when we use a different microarchitecture configuration. Inside the 50M-instruction intervals there will be small-scale repetitive behavior from loops. If the repetitive behavior is small enough then it is fine to simulate only 10M instructions from one program's 50M interval while simulating 40M instructions from the other interval, even though another microarchitectural configuration might allow them to progress at the same rate. On the other hand, a 15M-instruction loop with different behavior in the last 5M instructions would not provide a workload that is consistent between these two configurations.

To create homogeneous intervals we perform SimPoint phase analysis at a smaller interval size than the 50M-instruction interval granularity to verify whether the phase behavior within the 50M instructions is constant. In most cases, especially for floating-point benchmarks, all of the frequent phases contain instances that have consistent behavior for at least 50M instructions.

3.2 Microarchitecture-Independent Characteristics

In order to be able to identify similarity across simulation points, we consider microarchitecture-independent characteristics measured over these simulation points. As mentioned before, the reason we consider microarchitecture-independent characteristics is that the workload analysis needs to be done only once so that its results can be used multiple times for estimating the performance of a collection of processor configurations. This is important since we want to run our analysis once on a benchmark suite and use the co-phases found across all the different architecture configurations during design space explorations. Table 1 summarizes the microarchitecture-independent characteristics that we use in this paper, which we now describe.

The range of microarchitecture-independent characteristics is fairly broad in order to cover all major program behaviors such as instruction mix, inherent ILP, working set sizes, memory strides, branch predictability, *etc.* The results given in the evaluation section of this paper confirm that this set of characteristics is indeed broad enough for accurately characterizing cross-program and cross-input similarity. We include the following characteristics:

Instruction mix. We include the percentage of loads, stores, control transfers, arithmetic operations, integer multiplies and floating-point operations.

ILP. In order to quantify the amount of instruction-level parallelism (ILP), we consider an idealized out-of-order processor model in which everything is idealized

Instruction Mix	1	% loads
	2	% stores
	3	% control transfers
	4	% integer operations
	5	% floating-point operations
	6	% no-operations
	7	% software prefetch operations
ILP	8	32-entry window
	9	64-entry window
	10	128-entry window
	11	256-entry window
Register Traffic	12	avg. number of input operands
	13	avg. degree of use
	14	prob. register dependence = 1
	15	prob. register dependence $\leq$ 2
	16	prob. register dependence $\leq$ 4
	17	prob. register dependence $\leq$ 8
	18	prob. register dependence $\leq$ 16
	19	prob. register dependence $\leq$ 32
	20	prob. register dependence $\leq$ 64
Working Set Size	21	I-stream at the 32B block level
	22	I-stream at the 4KB page level
	23	D-stream at the 32B block level
	24	D-stream at the 4KB-page level
Data Stream Strides	25	prob. local load stride = 0
	26	prob. local load stride $\leq$ 8
	27	prob. local load stride $\leq$ 64
	28	prob. local load stride $\leq$ 512
	29	prob. local load stride $\leq$ 4096
	30	prob. local store stride = 0
	31	prob. local store stride $\leq$ 8
	32	prob. local store stride $\leq$ 64
	33	prob. local store stride $\leq$ 512
	34	prob. local store stride $\leq$ 4096
	35	prob. global load stride = 0
	36	prob. global load stride $\leq$ 8
	37	prob. global load stride $\leq$ 64
	38	prob. global load stride $\leq$ 512
	39	prob. global load stride $\leq$ 4096
	40	prob. global store stride = 0
	41	prob. global store stride $\leq$ 8
	42	prob. global store stride $\leq$ 64
	43	prob. global store stride $\leq$ 512
	44	prob. global store stride $\leq$ 4096
Branch Predictability	45	GAg PPM predictor
	46	PAg PPM predictor
	47	GAs PPM predictor
	48	PAs PPM predictor

Table 1: Microarchitecture-independent characteristics.

or unlimited except for the window size. We measure for a given window size over a set of 32, 64, 128 and 256 in-flight instructions how many independent instructions there are within the current window.

Register traffic characteristics. We collect a number of characteristics concerning registers [8]. Our first characteristic is the average number of input operands to an instruction. Our second characteristic is the average degree of use, or the average number of times a register instance is consumed (register read) since its production (register write). The third set of characteristics concerns the register dependency distance. The register dependency distance is defined as the number of dynamic instructions between writing a register and reading it.

Working set. We characterize the working set size of the instruction and data stream. For each interval, we count how many unique 32-byte blocks were touched and how many unique 4KB pages were touched for both instruction and data accesses.

terized with respect to local and global data strides [12]. A global stride is defined as the difference in the data memory addresses between temporally adjacent memory accesses. A local stride is defined identically except that both memory accesses come from a single instruction—this is done by tracking memory addresses for each memory operation. When computing the data stream strides we make a distinction between loads and stores.

Branch predictability. The final characteristic we want to capture is branch behavior. The most important aspect would be how predictable the branches are for a given interval of execution. In order to capture branch predictability in a microarchitecture-independent manner we used the Prediction by Partial Matching (PPM) predictor proposed by Chen *et al.* [4], which is a universal compression/prediction technique.

A PPM predictor is built on the notion of a Markov predictor. A Markov predictor of order k predicts the next branch outcome based upon k preceding branch outcomes. Each entry in the Markov predictor records the number of next branch outcomes for the given history. To predict the next branch outcome, the Markov predictor outputs the most likely branch direction for the given k-bit history. An m-order PPM predictor consists of $(m+1)$ Markov predictors of orders 0 up to m. The PPM predictor uses the m-bit history to index the mth-order Markov predictor. If the search succeeds, *i.e.* the history of branch outcomes occurred previously, the PPM predictor outputs the prediction by the mth order Markov predictor. If the search does not succeed, the PPM predictor uses the $(m-1)$-bit history to index the $(m-1)^{\text{th}}$-order Markov predictor. In case the search misses again, the PPM predictor indexes the $(m-2)^{\text{th}}$-order Markov predictor, *etc.* Updating the PPM predictor is done by updating the Markov predictor that makes the prediction and all its higher order Markov predictors. In this paper, we consider four variations of the PPM predictor: GAg, PAg, GAs and PAs. 'G' means global branch history whereas 'P' stands for per-address or local branch history; 'g' means one global predictor table shared by all branches and 's' means separate tables per branch. We want to emphasize that these metrics for computing the branch predictability are microarchitecture-independent. The reason is that the PPM predictor is to be viewed as a theoretical basis for branch prediction rather than an actual predictor that is to be built in hardware.

3.3 Workload Characterization

The collected microarchitecture-independent data is useful for research purposes with minimal further analysis, as PCA reveals important quantitative, microarchitecture-independent, properties of each interval and simple statistical techniques reveal the nature of

the collected data for all intervals, finding the highest and lowest values as well as grouping the intervals into quintiles: very low, low, medium, high and very high. Determining mean values and standard deviation is also possible but might not be useful for many characteristics since probabilities are unlikely to be normally distributed. Once we have classified all intervals according to their characteristics there are three important things that we can do.

First, we can evaluate a processor using combinations of the extreme behaviors. Researchers can pick intervals with properties appropriate for their experiments, just as entire benchmarks known to have particular execution properties are used for single-threaded experiments. For example, we can run only intervals with very low ILP or very high probability of control transfer and low probabilities for branch predictors. The working set size and data stream predictors can be used to test cache configurations. The highest and lowest values for a characteristic indicate intervals suitable for studying the limits of processors without resorting to a synthetic workload. For example, our collection of 50M-instruction intervals for SPEC CPU 20002000 includes one that is mostly stores with no loads and another that is mostly loads with no stores.

Second, we can identify the most average intervals, those that are never categorized as very high nor very low and have the most medium categorizations. Collections of these intervals form a baseline performance model when investigating the effects of changing workloads on a fixed machine configuration.

Third, after running a number of interval combinations and finding that some of them do poorly, we can correlate performance with quintiles for each characteristic. It is easy to determine which characteristics are independent of problems and focus on the remaining ones. In many cases the characteristics will suggest which resources, such as functional units or prefetchers, need to be improved to handle the problematic workloads.

3.4 Principal Components Analysis

Principal components analysis (PCA) [10] is a statistical data analysis technique that presents a different view on a given data set. The two most important features of PCA are that (*i*) PCA is a data reduction technique that reduces the dimensionality of a data set and (*ii*) PCA removes correlation from the data set. Both features are important to increase the understandability of the data set. For one, analyzing a q-dimensional space is obviously easier than analyzing a p-dimensional space in case $q \ll p$. Second, analyzing correlated data might give a distorted view; non-correlated data does not have that problem. The reason is that a distance measure in a correlated space gives too much weight to correlated same underlying program characteristic; the underlying characteristic would thus have too much weight in the overall distance measure.

The input to PCA is a matrix in which the rows are the *cases* and the columns are the *variables*. In this paper, each row represents a single 50-million instruction interval. The columns represent the 48 microarchitecture-independent characteristics presented in the previous subsection for each of the phases in a co-phase.

PCA computes new variables, called *principal components*, which are *linear combinations* of the original variables, such that all principal components are uncorrelated. PCA transforms the p variables $X_1, X_2, \ldots, X_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} X_j$. This transformation has the properties (i) $Var[Z_1] \geq Var[Z_2] \geq \ldots \geq Var[Z_p]$ — this means Z_1 contains the most information and Z_p the least; and (ii) $Cov[Z_i, Z_j] = 0, \forall i \neq j$ — this means there is no information overlap between the principal components. Note that the total variance in the data (variables) remains the same before and after the transformation, namely $\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i]$. In this paper, X_i is the ith microarchitecture-independent characteristic; Z_i then is the ith principal component after PCA. $Var[X_i]$ is the variance of the original microarchitecture-independent characteristic X_i computed over all intervals. Likewise, $Var[Z_i]$ is the variance of the principal component Z_i over all intervals.

As stated in the first property in the previous paragraph, some of the principal components will have a high variance while others will have a small variance. By removing the principal components with the lowest variance from the analysis, we can reduce the dimensionality of the data while controlling the amount of information that is thrown away.

We retain q principal components which is a significant information reduction since $q \ll p$ in most cases. To measure the fraction of information retained in this q-dimensional space, we use the amount of variance $(\sum_{i=1}^{q} Var[Z_i])/(\sum_{i=1}^{p} Var[X_i])$ accounted for by these q principal components. For example, criteria such as '70%, 80% or 90% of the total variance should be explained by the retained principal components' could be used for data reduction. An alternative criterion is to retain all principal components for which the individual retained principal component explains a fraction of the total variance that is at least as large as the minimum variance of the original variables.

By examining the most important q principal components, which are linear combinations of the original variables ($Z_i = \sum_{j=1}^{p} a_{ij} X_j, i = 1, \ldots, q$), meaningful interpretations can be given to these principal components in terms of the original microarchitecture-independent characteristics. A coefficient a_{ij} that is close to +1 or -1

on the principal component Z_i. A coefficient a_{ij} that is close to 0 on the other hand, implies no impact.

In principal components analysis, one can either work with normalized or non-normalized data — the data is normalized when the mean of each variable is zero and its variance is one. In the case of non-normalized data, a higher weight is given in the analysis to variables with a higher variance. In our experiments, we have used normalized data because of our heterogeneous data; *e.g.*, the variance of the ILP is orders of magnitude larger than the variance of the instruction mix.

The output obtained from PCA is a matrix in which the rows are the 50M phases and the columns are the retained principal components. Before we proceed to the next step we make sure we normalize the principal components, *i.e.*, we rescale the principal components to unit variance. The reason is that a non-unit variance of a principal component is a consequence of the correlation as observed in the original data set. And since our next step in the data analysis uses a distance measure to compute the similarity between cases, we make sure correlation does not give a higher weight to correlated variables.

3.5 Cluster Analysis

The next step in our workload reduction methodology is to perform cluster analysis (CA) [10] on co-phases. There exist two commonly used strategies for applying cluster analysis, namely linkage clustering and K-means clustering. Since K-means clustering is less compute-intensive than linkage clustering, we use K-means in this paper. The K-means algorithm is an iterative process that works in two steps per iteration. The first step is to compute the Euclidean distance of each point in the multi-dimensional space to each cluster center. In the second step, each point gets assigned to the closest cluster. As such, new clusters are formed and new cluster centers are to be computed. This algorithm is iterated until convergence is observed, and cluster membership ceases to change between iterations. For this paper, we use the SimPoint 3.0 software release.

The input to the cluster analysis is a matrix in which the rows are all possible co-phases and the columns are the retained principal components for each phase in the co-phase. Cluster analysis thus finds a number of groups or clusters of co-phases that exhibit similar microarchitecture-independent behavior. We only include distinct co-phases in the matrix: if A and B are phases, then co-phases AB and BA are considered identical. For dies with multiple multithreaded cores, each group of threads on a core may be reordered and the cores may be reordered without making a distinctly different co-phase, but intermixing the threads of two cores will produce a distinct co-phase. It is easy to tell if two the sort the cores in lexicographic order. The resulting structures will be identical if the co-phases are equivalent (not distinct). We can easily generate distinct co-phases randomly or exhaustively using this canonical form.

This definition of distinct co-phases minimizes that number of inputs to the clustering and results in the best possible results for a given number of clusters. Unfortunately, it results in different simulation points for different core and SMT context configurations even when the total number of threads are the same. If this is undesirable, then the clustering can be done using a chip layout that requires a superset of the reordering restrictions of all of the relevant chip configurations. For example, 1×8, 8×1, 4×2 and 2×4 chip layouts can be approximated by a $2 \times 2 \times 2$ chip layout. This ensures that if two threads are distinct in one of the original configurations then they will be distinct in the new configuration.

Our definition of distinct co-phases causes a problem for clustering. If A and D are phases with similar properties and so are B and C, then we would like the co-phases AB and CD to be similar. The normal Euclidean distance metric would consider the rows of statistics representing these co-phases to be far apart unless all of the phases were similar. The co-phases AB and DC would be close together, however. We avoid this problem by using a different distance metric. In this metric, the distance between two co-phases is the minimum of the Euclidean distances between the first co-phase and all equivalent permutations of the second one.

Calculating this distance metric naïvely would be an expensive operation because it requires $n!$ distance computations for n threads. For example, for 8 threads, we would need to compute $8! = 40320$ distances. Fortunately, this number assumes a lot of repeated work. Each time the distance between a pair of phases is calculated there are 4 dimensions to consider (each a subtraction followed by a multiplication). There are only 64 pairs of phases, so the distances between the phases in each pair need only be calculated once. Each of the 8! distances between co-phases is reduced to the summation of 8 table lookups, and an obvious stack-based algorithm will average under 3 table lookups and additions per distance calculated. These optimizations alone reduces the slowdown to a factor of about 1700 on a machine with fast multiplies (better on other machines). If each core has two threads the normal distance metric will take twice as long. The 64-entry table will take four times as long to generate (twice as many columns and two orders), but the 8! part of the algorithm will run at the same speed, so the slowdown is nearly halved. Memoization can reduce the slowdown to 30–40 times for both 8-core configurations using a table with only 2^8 entries. Each entry in the table represents the minimum cost of mapping a subset of the 8 cores in the permuted co-phase to

The number of co-phases for an 8-core SMT machine would cause the clustering algorithm a much greater slowdown than the distance metric — there are about 10^{29} distinct co-phases that can be formed from our 50-million instruction intervals. When we use more than two cores or threads we use a random sample of co-phases. Weighting the probability of selection of a co-phase according to the product of the weights of the component phases ensures that the chosen centers will be near to the co-phases with greatest weight.

Using these optimizations, a careful implementation of our distance metric and random sampling of co-phases, we can analyze multithreaded workloads consisting of large numbers of threads. This analysis will allow us to simulate just a few representative instances, vastly reducing analysis time for modern computer architectures.

3.6 Interpolation of Cluster Centers

In standard SimPoint, the co-phase that is closest to each cluster's centroid is called the representative co-phase. The weight assigned to this representative co-phase, referred to as the *co-simulation point* is the sum of the weights of the co-phases that are members of the given cluster divided by the total weight of all co-phases. Only the representative simulation points need to be simulated when we estimate performance numbers.

To improve our accuracy without increasing simulation time, we observe that points that are between other points should have in-between performance. In a Euclidean metric space we could choose cluster centers that form a convex hull around the target point and use geometry to determine the weight of each selected center. This is much more challenging in our metric, so we use a simpler scheme. Each point is computed as a weighted average of its n nearest neighbors. If the closest neighbor is at distance d_0, then each point has relative weight $e^{-c\frac{d}{d_0}}$. Appropriate choices for n and c depend upon both the number of cluster centers and each other — a large c will compensate for an overly-large n by discounting faraway neighbors; a small c requires a small n so that faraway points are not included).

3.7 Weighting Average Throughput

To simplify comparisons between techniques to reduce the number co-simulation points, we propose a single weighted average throughput metric. We consider two types of weights, the weight of a pair of benchmarks and the weight of co-phases for each pair of benchmarks.

Each benchmark consists of a program and its input. Programs have from one to five inputs, but a program with many inputs is not necessarily more significant than a program with a single input. Thus, we consider each grams. Each input is equally important as any other for the same program. In this scheme, the weight of `lucas` and `mesa` is 25 times that of `gcc-166` and `gzip-program` since `gcc` and `gzip` each have 5 inputs.

For a given pair of benchmarks, we must subdivide the weights between co-phases. Unlike programs and inputs, the co-phases clearly should have distinct weights because the phases that compose them are known to have particular weights. Some phases represent less than 5% of a benchmark whereas others represent over 90% of benchmark. The weight that we give to a co-phase is equal to the product of the weights of the constituent phases. Thus if one phase represents 20% of a benchmark and the other 30%, the weight of the co-phase is 6% that of the pair of benchmarks.

When the number of threads is large there may be too many co-phases to estimate their performance efficiently. Random sampling allows the weighted average to be estimated efficiently at any desired level of accuracy. Our analysis procedures allow the average weighted throughput to be estimated using detailed simulation of only a small number co-simulation points.

4 Experimental Setup

4.1 Baseline Simulator

We use the M5 simulator [2] from the University of Michigan, which is based on SimpleScalar3.0c [3] as our SMT simulation environment. The configurations used for this simulator are shown in Table 2. It is configured to support an intensive multithreaded workload; hence the abundant reorder buffer and processor width. The memory hierarchy is based on current-generation processors. For the L1 caches, unified L2 cache and branch predictor we considered two design points each, for eight possible combinations. We simulated SPEC CPU2000 benchmarks compiled for the Alpha ISA.

Each co-phase was executed until a combined 50M instructions were committed by both threads. Since our target workload (all co-phases) is constant and each phase is homogeneous, we calculate performance using throughput in instructions per cycle. Due to the long simulation period, warmup effects correspond to less than 0.5% variation in throughput. Nonetheless, we ignore the first 5M combined instructions to remove error due to warmup effects.

4.2 Cluster and Principal Components Analysis

We analyzed the benchmarks and microarchitecture-independent co-phase features using SimPoint 3.0 [9]. When analyzing benchmarks with SimPoint we found

	64kB 2-way set-associative, 64-byte blocks, 1-cycle latency
D-Cache	32kB 8-way set-associative, 64-byte blocks, 3-cycle latency *or* 64kB 8-way set-associative, 64-byte blocks, 3-cycle latency
Unified L2	1 MB 8-way set-associative, 128-byte blocks, 10-cycle latency *or* 4 MB 16-way set-associative, 128-byte blocks, 14-cycle latency
Memory	250-cycle latency
Branch Pred	21264-style hybrid predictor with 13-bit global history indexing a 8k-entry global PHT and 8k-entry choice table; 2k 11-bit local history entries indexing a 2k-entry local PHT A: 4kB, 4-way set-associative BTB; 3-cycle misprediction recovery *or* B: 4kB, 2-way set-associative BTB; 2-cycle misprediction recovery
OOO Issue	out-of-order issue, 256-entry re-order buffer
Width	8 instructions per cycle (Fetch, Decode, Issue and Commit)
Func Units	6 Integer, 2 Integer Multiply, 4 FP Add, 2 FP Multiply

Table 2: SMT processor configuration.

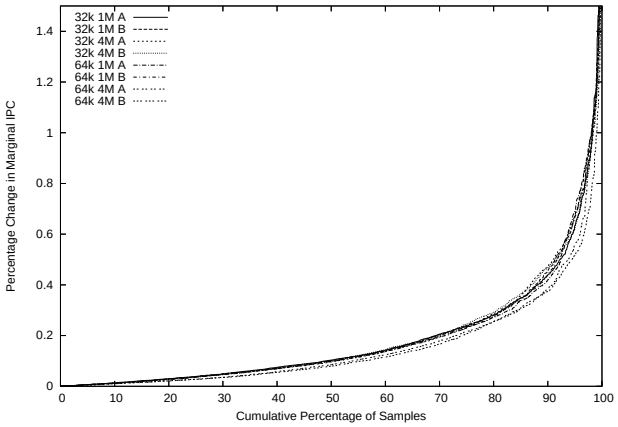

Figure 1: Cumulative distributive function for marginal change in IPC.

up to 10 phases per benchmark. We selected an average of 5 phases per program by removing phases that corresponded to less than 2.5% of program execution.

The microarchitecture-independent analysis was performed using a modified version of SimpleScalar. We analyzed each of the 50M-instruction simulation points found in the previous step.

From the PCA step we selected the 4 most-significant dimensions, which were sufficient to explain over 44% of the variance. Thus clustered 8-dimensional data. Increasing the number of dimensions used leads to poorer cluster analysis as clustering treats all of the dimensions as equally significant — this leads to the curse of dimensionality problem.

5 Experimental Evaluation

5.1 Homogeneous Intervals

The accuracy of our simulations depends on homogeneous behavior within each 50M-instruction interval. If the pattern of execution for one program deviated significantly near the end of the simulation interval, this would affect simulations that execute the different code, but some simulations might make faster progress with the second program and thus never execute the different code. It would be misleading to compare the results of the two experiments. Thus, we need to verify that the intervals contain homogeneous behavior. To do this, we examine the execution of all co-phases on our baseline processor and observe the effects of varying the length of simulation between 45M and 50M instructions, in 0.5M instruction increments. At each increment we compare the IPC with the IPC prior to the increment and determine the relative difference caused by the slightly longer execution. We plot these values in Figure 1 as a cumulative distributive function (CDF) for eight microarchitectures. For 80% of samples the variation in throughput is at most 0.3% and less than 1% of samples cause a variation of more than 1.2%. Thus we can expect that our simulations will provide stable, reliable results that are not sensitive to the exact point at which simulation terminates. Furthermore, the error rate is not particularly sensitive to the machine configuration.

Variation in the middle of a simulation could also lead to incomparable executions provided that both programs have significant variation, as we demonstrated in our previous work [17, 18]. For the homogeneous intervals in this paper, the degree of variation in the middle of the intervals is similar to that at the end of the execution intervals. The use of 50M-instruction intervals ensures that the natural fine-grain program variation is insignificant on the scale that we sample.

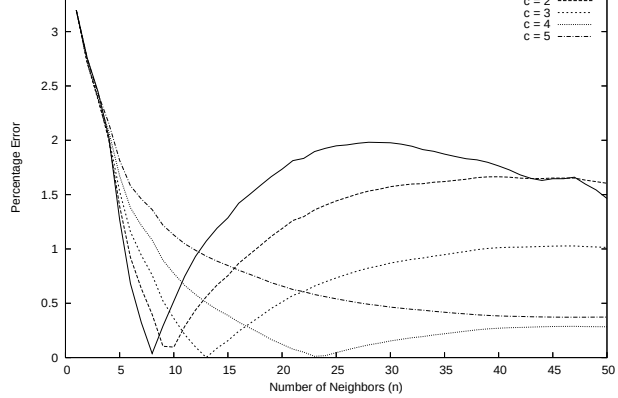

Figure 2: Error using different interpolation parameters (configuration 32k 4M A).

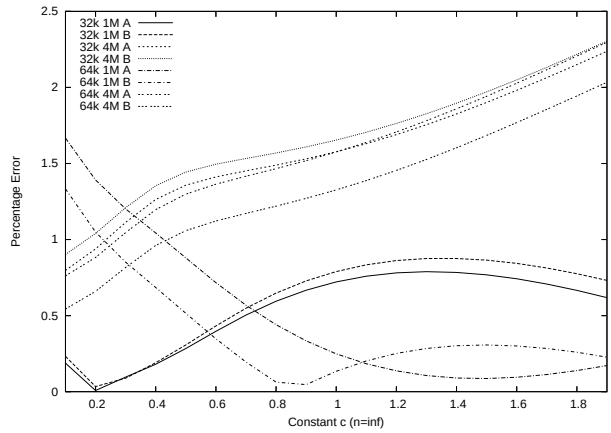

Figure 3: Error varying c using all configurations.

5.2 Interpolation

In Figure 2 we examine the effects on estimating throughput that changing the parameters to the interpolation algorithm has, as described in Section 3.6. For each combination of parameters we use 50 representative points. We use one line for each choice for constant c. The x-axis is the number of neighbors used to compute throughput, n. Two values of n are of particular note. When $n = 1$ the algorithm is equivalent to the standard SimPoint algorithm that selects a single representative simulation point (thus c has no effect). Although this method is reasonably accurate (3.2% error), it performs worse than any other combination of parameters. At $n = 50$, all representative points are used. Their weights are dependent upon their distances and c. The larger values of c lead to more accurate results because they give negligible weight to distant representative points. Small values of c and n combine to get excellent results but the sharp inflection points indicate the need for fine-tuning. All methods give excellent results (under 2% relative error) as long as at least 5 neighbors are used, but using all the points is the most robust option.

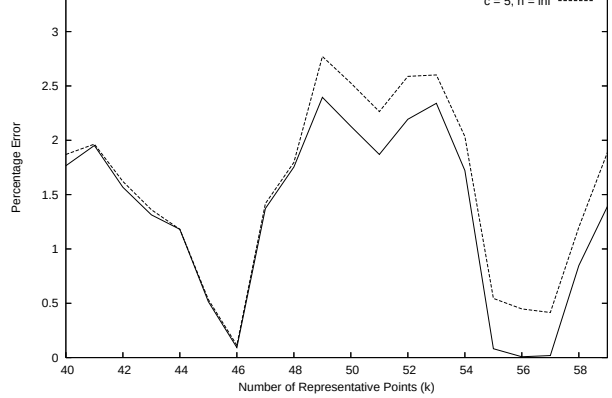

Figure 4: Error using different numbers of randomly chosen representative points (configuration 32k 4M A).

In Figure 3 we examine the effects on error of using different values of c with all eight microarchitecture configurations. We see that the error rates are low in all cases, but different depending on machine configuration. The configurations with 4M L2 caches have similar changes in error rates for all c. The remaining configurations are also distinguishable by L1 cache size. Since the error rates are low, any value of c in this range can be used to accurately compare different microarchitectural configurations.

5.3 Random Representative Points

We also investigated using random selection of representative points rather than cluster centers. Without interpolation, the results were significantly worse than using clustering. Interpolation, however, leads to similar results whether centers are chosen randomly or by using clustering. The main difference that we found were slightly higher error rates and a preference for slightly fewer neighbors. As we can see in Figure 4, we consistently get at most 2.5% error when using varying numbers of randomly chosen cluster centers. Since we can get accurate results despite randomly selecting centers, we can scale the algorithm to large numbers of cores and threads. Randomly sampling the set of possible cophases and clustering the results should give good results even though the 'best' cluster centers might not be in the sample sets. Clustering ideally uses thousands of iterations of cluster center movement with distance comparisons to all points in every step. Should our distance metric be too expensive for huge numbers of cores or threads, the random center results suggest that we could reasonably reduce the number of iterations to compensate or eliminate clustering altogether.

Architecture studies of multithreaded processor need to balance the performance requirements of every combination of benchmarks. Simulating all of the benchmark combinations is excessively time-consuming, even when using sampling techniques such as the Co-Phase Matrix.[18, 17] We demonstrate a technique for analyzing a benchmark suite and finding all of the distinct co-phase behaviors that can occur when pairs of benchmarks run together. By clustering the co-phases behaviors we are able to find representative *co-simulation points* that can be simulated as substitute for simulating all of the co-phases. We demonstrate that less than 50 co-simulation points provide results differing by less than 2.5% from simulating all co-phases.

This set of co-simulation points can be used to compare the performance of different microarchitectural configurations by executing 2.5 billion instructions per configuration. Our simulation point selection technique simplifies the simulation procedure by ensuring that each co-simulation point has homogeneous behavior.

Acknowledgments

We would like to thank the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by NSF grant No. CCF-0342522, NSF grant No. CCF-0311710, a UC MICRO grant, and a grant from Intel and Microsoft. Lieven Eeckhout is a Postdoctoral Fellow with the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen) and is also supported by Ghent University, IWT, HiPEAC and the European SARC project No. 27648.

References

[1] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded commercial workloads. In *Annual International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[2] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Feb. 2003.

[3] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[4] I. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 128–137, Oct. 1996.

[5] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. *sium on Workload Characterization (IISWC)*, pages 2–12, Oct. 2005.

[6] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload design: Selecting representative program-input pairs. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 83–94, Sept. 2002.

[7] M. Ekman and P. Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–99, Mar. 2005.

[8] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 22nd Annual International Symposium on Microarchitecture (MICRO-22)*, pages 236–245, Dec. 1992.

[9] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[10] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.

[11] 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 a nd Software*, Mar. 2005.

[12] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *Proceedings of the 2004 International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 57–67, Mar. 2004.

[13] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, pages 10–20, Mar. 2005.

[14] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2003.

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

[16] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *2005 International Conference on High Performance Embedded Architectures and Compilation (HiPEAC)*, pages 47–67, Nov. 2005.

[17] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaneous multithreading simulation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 143–153, Mar. 2006.

[18] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of the 2004 IEEE International Symposium on Performanc e Analysis of Systems and Software (ISPASS'04)*, Mar. 2004.