Exploiting Program Microarchitecture Independent Characteristics and Phase Behavior for Reduced Benchmark Suite Simulation

Lieven Eeckhout
ELIS Department
Ghent University, Belgium
Email: leeckhou@elis.UGent.be

John Sampson
CSE Department
University of California, San Diego
Email: jsampson@cs.ucsd.edu

Brad Calder
CSE Department
University of California, San Diego
Email: calder@cs.ucsd.edu

***Abstract*— Modern architecture research relies heavily on detailed pipeline simulation. Simulating the full execution of an industry standard benchmark can take weeks to complete. Simulating the full execution of the whole benchmark suite for one architecture configuration can take months. To address this issue researchers have examined using targetted sampling based on phase behavior to significantly reduce the simulation time of each program in the benchmark suite. However, even with this sampling approach, simulating the full benchmark suite across a large range of architecture designs can take days to weeks to complete.**

The goal of this paper is to further reduce simulation time for architecture design space exploration. We reduce simulation time by finding similarity between benchmarks and program inputs at the level of samples (100M instructions of execution). This allows us to use a representative sample of execution from one benchmark to accurately represent a sample of execution of other benchmarks and inputs. The end result of our analyis is a small number of sample points of execution. These are selected across the whole benchmark suite in order to accurately represent the complete simulation of the whole benchmark suite for design space exploration. We show that this provides approximately the same accuracy as the SimPoint sampling approach while reducing the number of simulated instructions by a factor of 1.5.

I. Introduction

Understanding the cycle level behavior of a processor during the execution of an application is crucial to modern computer architecture research. To gain this understanding, researchers typically employ detailed simulators that model each and every cycle. Unfortunately, this level of detail comes at the cost of speed, and simulating the full execution of an industry standard benchmark can take weeks or months to complete, even on today's fastest simulators running on the fastest machines. Exacerbating this problem further is the need of architecture researchers to simulate each benchmark over a variety of different architectural configurations and design options, to find the set of features that provides the appropriate tradeoff between performance, complexity, area, and power. The same program binary, with the exact same input, may be run hundreds or thousands of times to examine how, for example, the effectiveness of a given architecture changes with cache size. Researchers need techniques which can reduce the number of machine-months required to estimate the impact of an architectural modification without introducing an unacceptable amount of error or excessive simulator complexity.

Sampled simulation is a well known approach for speeding up simulation runs [1], [2], [3]. One tool for guiding sampled simulations is called SimPoint [1], [2] which reduces the simulation time of a single program by exploiting the phase behavior found in applications. Executing programs have behaviors that change over time in ways that are not random, but rather are often structured as sequences of a small number of reoccurring behaviors, which are called phases. This structured behavior is a great benefit to simulation. SimPoint allows very fast and accurate sampling by identifying each of the repetitive behaviors and then taking only a single sample of each repeating behavior to represent that behavior. All of these samples together represent the complete execution of the program. SimPoint intelligently chooses a very small set of samples called *Simulation Points* that, when simulated and weighted appropriately, provide an accurate picture of the complete execution of the program. In addition, simulating only these carefully chosen simulation points can save hours of simulation time.

SimPoint's focus was on finding a small set of representative samples to represent a single program's execution. What we have found in this paper is that two samples from two different applications can have very similar behavior across many different microarchitecture independent metrics, and these samples end up also having very similar architecture metrics. This means that we can reduce the simulation time of a complete benchmark suite by picking a representative reduced set of samples from the programs across the benchmark suite. That is the focus of this paper.

We use SimPoint's model of breaking each program's execution into a set of contiguous non-overlapping intervals of 100 million instructions of execution. For each interval of execution we collect a set of architecture independent characteristics such as instruction mix, data and instruction working set sizes, and more. We then perform Principal Component Analysis (PCA) across a set of all program and input pairs for a benchmark suite to reduce the dimensionality of the data set into non-correlated principal components. These principal components are then represented as a signature for each interval of execution (sampled) for all benchmarks and we use cluster analysis to pick the samples to use across all of the benchmarks. This results in approximately the same accuracy as SimPoint and a reduction of a factor of 1.5 in the number of instructions that need to be simulated.

The contributions of this paper are:

- We propose reducing the simulation time for a whole benchmark suite by finding cross program and cross input similarities based upon representing a program's phase behavior through a set of microarchitecture independent characteristics.
- We show that using microarchitecture independent characteristics based on instruction mix, working set sizes, etc, can find a set of representative simulation points independent of the microarchitecture. These simulation points can then be used to represent the complete benchmark suite to accurately guide design space exploration.
- We show that performing benchmark suite reduction on a per phase level across all of the benchmark suite reduces simulation time by a factor of 1.5 over using SimPoint with roughly the same error rate.
- We show that when performing benchmark suite reduction it is better to select representative samples from all program-input pairs instead of performing benchmark suite reduction by focusing on which program-input combinations can be completely removed from simulation as was done in [4], [5].

II. Prior Work

In this section we summarize the SimPoint approach that we improve upon and prior work in benchmark suite reduction.

A. Program Phase Behavior and SimPoint

Sherwood *et al.* [2] proposed using code signatures to break a program's execution into phases. To identify phases, they broke a program's execution into contiguous non-overlapping intervals. An *interval* is a continuous portion of execution (a slice in time) of a program. A *phase* is a set of intervals within a program's execution with similar behavior, regardless of temporal adjacency. This means that a phase may appear many times as a program executes. Phase classification partitions a set of intervals into phases with similar behavior. For our paper we use an interval size of 100 million instructions as in [2].

Sherwood *et al.* [2] created a tool called SimPoint that groups together intervals with similar code signatures into the same phase. The code signature for each interval is represented by a Basic Block Vector (BBV) [6] to capture information about changes in a program's behavior over time. A *Basic Block Vector* is a one dimensional array, where each element in the array corresponds to one static basic block in the program. The BBV for each interval represents the frequency of execution of each static basic block in the program for that interval. SimPoint then performs clustering on BBVs, because each vector contains the frequency distribution of code executed in each interval. By comparing BBVs of two intervals during clustering, SimPoint can evaluate the similarity of two intervals. If the distance between the two BBVs is small (close to 0), then the two intervals spend about the same amount of time in roughly the same code, and therefore we expect the performance of those two intervals to be similar. Code signatures grouped into the same cluster have been shown to exhibit similar CPI, numbers of branch mispredictions, numbers of cache misses, etc. [6].

After this phase classification algorithm is done, intervals with similar code usage will be grouped together into the same phase. Then from each phase, they choose one representative interval that will be simulated in detail to represent the behavior of the whole phase. Therefore, by simulating *only* one representative interval per phase, SimPoint can extrapolate and capture the behavior of the entire program. This set of intervals, one chosen from each phase, is called the set of *Simulation Points*. This same set of simulation points for a given program-input pair are simulated to create an estimated CPI (and other architecture metrics) for each architecture configuration during design space exploration.

In this paper, we build on SimPoint to try to find cross program and cross input similarities. To achieve this we examine finding similar behavior using signature vectors that are both microarchitecture and code signature independent. SimPoint's vectors (BBVs) are code dependent, since they are based upon the code signatures when executing that program. This is an important distinction, because little similarity can be found across programs and only minor similarity across inputs if one was to look at only code signatures.

B. Benchmark Suite Reduction with PCA

Eeckhout *et al.* [4] 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 III-B 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.

The set of program characteristics used by Eeckhout *et al.* [4] is a collection of microarchitecture-independent and *microarchitecture-dependent* characteristics. The microarchitecture-dependent characteristics include for example miss rates for specific cache configurations and specific branch predictors. The main disadvantage of using microarchitecture-dependent characteristics is that it is unclear whether the results are directly applicable for other microarchitectural configurations. In this paper, we follow the proposal by Phansalkar *et al.* [5] to only use a set of important *microarchitecture-independent* characteristics. The main contribution made in this paper over these previously proposed benchmark suite reduction techniques which considered aggregate metrics only (averaged over the complete benchmark execution), is that we reduce the benchmark suite based on a phase level analysis. We break up a complete benchmark execution into intervals in order to collect a number of microarchitecture-independent characteristics per phase. As we will show, by exploiting the phase behavior of a benchmark execution the reduced workload is more accurate than when aggregate behavior is considered.

Several additional studies have been performed using this workload analysis approach, for example comparing the memory behavior of SPEC CPU95 and SPEC CPU2000 [7], subsetting benchmark suites [8], studying the interaction between the virtual machine and the Java application [9], and validating reduced input sets [10]. Yi *et al.* [11] use the Plackett-Burman design to quantify the similarity between benchmarks. The Plackett-Burman design requires that a number of simulations are run for a number of microprocessor configurations. All of this prior work on quantifying benchmark similarity considered microarchitecture-dependent characteristics measured from complete benchmark executions. Our paper on the other hand, uses only microarchitecture-independent characteristics at the phase level.

III. Workload analysis

This section discusses our workload analysis. We identify two major issues: (i) the microarchitecture-independent characteristics that need to be collected for each interval of execution for the various benchmarks under consideration, and (ii) the data analysis to extract useful information from this large data set.

A. Microarchitecture-independent characteristics

In order to be able to measure cross-input and cross-program similarity at the phase level, we consider microarchitecture-independent characteristics measured over intervals of program execution. The program execution intervals we use in this paper are fixed-length intervals of 100 million dynamically executed instructions. This does not affect the generality of our approach—the approach presented in this paper could also be used for fixed-length intervals of other sizes as well as for variable-length intervals. The reason why we use 100M instruction intervals is that 100M intervals are easy to use because they are fairly insensitive to warmup issues, i.e. simple warmup strategies are sufficient [12]. Note this is also the reason why Intel's PinPoint uses large instruction intervals [13].

The reason we consider microarchitecture-independent characteristics instead of microarchitecture-dependent 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 simulation points found across all the different architecture configurations during design space explorations. Table I 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 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 [14]. 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.

Data stream strides. The data stream is characterized with respect to local and global data strides [15]. 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.* [16], 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 $(m-1)$th order Markov predictor. In case the search misses again, the PPM predictor indexes the $(m-2)$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

category	no.	characteristic
instruction mix	1	percentage loads
	2	percentage stores
	3	percentage control transfers
	4	percentage arithmetic operations
	5	percentage integer multiplies
	6	percentage fp operations
ILP	7	32-entry window
	8	64-entry window
	9	128-entry window
	10	256-entry window
register traffic	11	avg. number of input operands
	12	avg. degree of use
	13	prob. register dependence $= 1$
	14	prob. register dependence ≤ 2
	15	prob. register dependence ≤ 4
	16	prob. register dependence ≤ 8
	17	prob. register dependence ≤ 16
	18	prob. register dependence ≤ 32
	19	prob. register dependence ≤ 64
working set size	20	I-stream at the 32B block level
	21	I-stream at the 4KB page level
	22	D-stream at the 32B block level
	23	D-stream at the 4KB-page level
data stream strides	24	prob. local load stride $= 0$
	25	prob. local load stride ≤ 8
	26	prob. local load stride ≤ 64
	27	prob. local load stride ≤ 512
	28	prob. local load stride ≤ 4096
	29	prob. local store stride $= 0$
	30	prob. local store stride ≤ 8
	31	prob. local store stride ≤ 64
	32	prob. local store stride ≤ 512
	33	prob. local store stride ≤ 4096
	34	prob. global load stride $= 0$
	35	prob. global load stride ≤ 8
	36	prob. global load stride ≤ 64
	37	prob. global load stride ≤ 512
	38	prob. global load stride ≤ 4096
	39	prob. global store stride $= 0$
	40	prob. global store stride ≤ 8
	41	prob. global store stride ≤ 64
	42	prob. global store stride ≤ 512
	43	prob. global store stride ≤ 4096
branch predictability	44	GAg PPM predictor
	45	PAg PPM predictor
	46	GAs PPM predictor
	47	PAs PPM predictor

TABLE I

MICROARCHITECTURE-INDEPENDENT CHARACTERISTICS.

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—it attains upper-limit performance—rather than an actual predictor that is to be built in hardware.

B. Statistical data analysis

Our statistical data analysis consists of two major steps: principal components analysis and cluster analysis.

1) Principal components analysis: Principal components analysis (PCA) [17] 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 variables (these correlated variables result from the 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, the cases are the various 100M intervals from the various benchmarks; the columns are the 47 microarchitecture-independent characteristics presented in the previous section. PCA computes new variables, called *principal components*, which are *linear combinations* of the original variables, such that all principal components are uncorrelated. PCA tranforms 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 '60%, 70% or 80% 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.

In this study the p original variables are the microarchitecture-independent characteristics mentioned in section III-A. 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 implies a strong impact of the original characteristic X_j 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 cases or the rows are the intervals for the various benchmarks and the variables or 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—correlation in a data set skews the distance measure. We will call the obtained multidimensional space, the rescaled PCA space.

2) Cluster analysis: The second step in our workload analysis is cluster analysis (CA) [17]. 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 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, i.e. cluster membership ceases to change between iterations. For this paper, we use the SimPoint software[1]. The SimPoint software evaluates and compares clustering results for *k* varying from 1 to max_K. And for each *k*, the SimPoint software also evaluates a number of randomly choosen initial cluster centers; in our experiments, we evaluated 7 random seeds for each *k*. For each clustering, the SimPoint software also computes the *Bayesian Information Criterion (BIC)* score which measures the goodness of fit of the clustering for the given data set. When all clustering experiments are done, the SimPoint software then picks the smallest clustering (value for k) for which the BIC score is within 90% of the maximum observed BIC over all clusterings [2].

IV. Experimental setup

In this paper, we use the SPEC CPU2000 benchmarks, see Table II. The binaries were taken from the SimpleScalar website[2]; they are compiled for the Alpha ISA. We used the reference inputs for all benchmarks. For most benchmarks we used multiple reference inputs; for the others we only used those inputs for which we could run all our simulation tools. The dynamic instruction count for each of them is shown in the right column of Table II.

Measuring the microarchitecture-independent characteristics discussed in section III-A is done using ATOM [18]. ATOM is a binary instrumentation tool that allows for instrumenting functions, basic blocks and instructions. The instrumentation itself is done offline, i.e. an instrumented binary is stored on disk. While running the instrumented binary, the microarchitecture-independent characteristics get collected.

In the evaluation section of this paper, we will use detailed architectural simulation using SimpleScalar/Alpha v3.0 [19] for computing CPI (number of committed instructions per cycle) numbers for each benchmark. We used Wattch [20] for computing EPI numbers (energy consumption per instruction). The baseline processor configuration is an 8-issue machine; this is the same configuration as in [2].

V. Evaluation

We now present the results from our reduced workload approach. We first discuss workload analysis through principal components analysis and show that the principal components correlate well with the phase behavior of a program's execution. Second, we present the reduced workload as obtained from our per-phase uarch-independent workload reduction approach and discuss where the reduction comes from. We subsequently compare this per-phase reduced workload to prior work, i.e. SimPoint [2] and reduced workloads obtained from aggregate program characterization [4], [5]. Finally, we evaluate the accuracy of the reduced workload for design space explorations.

A. Workload analysis

As mentioned before, the input to our workload analysis is a vector signature for each interval of execution. The elements in the vector signature are the microarchitecture-independent characteristics—the dimensionality of a vector signature is 47. There are as many vector signatures as there are 100M-instruction intervals of execution over all program-input pairs in the workload. For SPECint2000, we have 32,964 vector signatures; for SPECfp2000, we have 38,997 vector signatures. Note that we perform separate analyses for SPECint and SPECfp because it is to be expected that there will be little similarity between integer and floating-point benchmarks.

The first step in our workload analysis, PCA, reduces the dimensionality of the vector signatures from 47 to 4 and 2 for SPECint2000 and SPECfp2000, respectively, where the dimensions are the principal components. Through various analyses we studied the required number of principal components. We varied the number of retained principal components and verified the accuracy of the reduced workload for performance prediction. We concluded from our detailed analysis that $q = 4$ for SPECint2000 and $q = 2$ for SPECfp2000, accounting for 60% of the total variance, makes a good balance between accuracy and simulation speedup. Adding more principal components usually increases the number of

[1] `http://www.cs.ucsd.edu/~calder/simpoint`
[2] `http://www.simplescalar.com`

program		input	I-cnt (B)
bzip2	int	graphic	143.5
		program	124.9
		source	108.8
crafty	int	ref	191.8
eon	int	cook	80.6
		rush	57.8
gap	int	ref	256.9
gcc	int	166	46.9
		200	108.6
		expr	12.0
		integrate	13.1
		scilab	62.0
mcf	int	ref	61.8
parser	int	ref	546.7
gzip	int	graphic	103.7
		log	39.5
		program	168.8
		random	82.1
		source	84.3
perlbmk	int	split.957	110.8
		diff	39.9
		makerand	2.0
		perfect	29.0
twolf	int	ref	346.4
vortex	int	ref1	118.9
		ref2	138.6
		ref3	133.0
vpr	int	route	84.0
ammp	fp	ref	326.5
applu	fp	ref	223.8
apsi	fp	ref	347.9
art	fp	110	41.7
		470	45.0
equake	fp	ref	131.5
facerec	fp	ref	268.2
fma3d	fp	ref	268.3
galgel	fp	ref	409.3
lucas	fp	ref	142.3
mesa	fp	ref	281.6
mgrid	fp	ref	419.1
sixtrack	fp	ref	470.9
swim	fp	ref	225.8
wupwise	fp	ref	349.6

TABLE II

THE SPEC CPU2000 BENCHMARKS USED IN THIS PAPER.

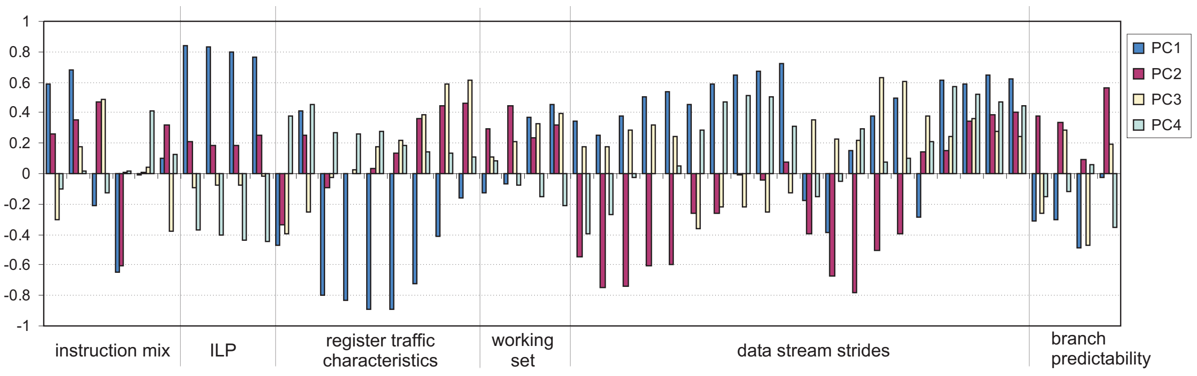

Fig. 1. Factor loadings for the first four principal components for SPECint2000.

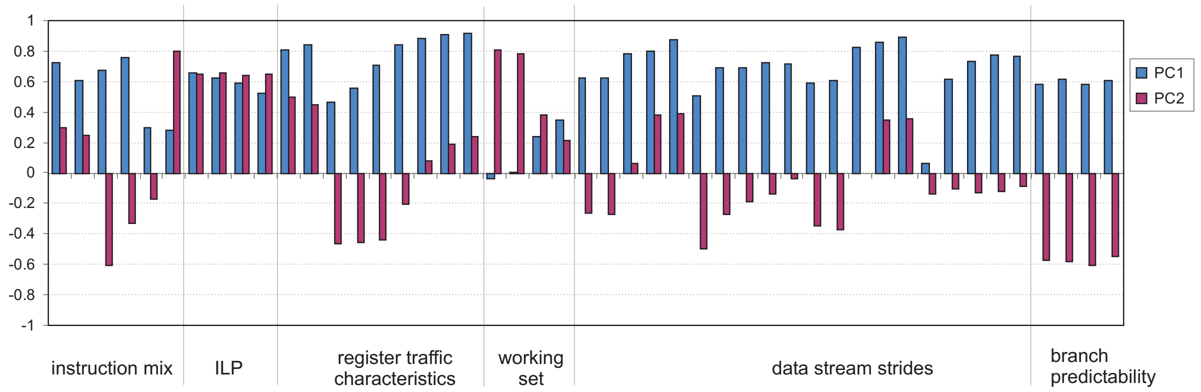

Fig. 2. Factor loadings for the first two principal components for SPECfp2000.

representative samples—which increases simulation time—while not significantly improving accuracy. Reducing the number of principal components usually reduces the accuracy making the reduced workload no longer representative of the complete workload. We do not include this detailed analysis on the number of required PCs here because of space constraints.

Figures 1 and 2 show the factor loadings of the various principal components in terms of the original microarchitecture-independent characteristics for SPECint2000 and SPECfp2000, respectively. The order of the microarchitecture-independent characteristics along the X axis in these figures is the same as in Table I. Figure 1 shows how the principal components are computed by combining the various metrics. For example, computing the value of the first principal component for a given interval of execution is calculated (starting with the first 4 PC1 bars on the left hand side of Figure 1) as: $PC1 = 0.58 \times percentage\_loads + 0.68 \times percentage\_stores - 0.21 \times percentage\_ctrl - 0.65 \times percentage\_arithmetic + \ldots$. Recall that the microarchitecture-independent characteristics appearing in this formula are the ones after normalization. This means for example that a SPECint2000 execution interval with a high first principal component tends to have a relatively large percentage load/store operations and a low number of arithmetic operations, few short register dependences, a high ILP, a large number of local and global store strides, and a high branch predictability. This also means that for the given data set there appears to be a strong correlation between these microarchitecture-independent characteristics. Similar interpretations can be given along the

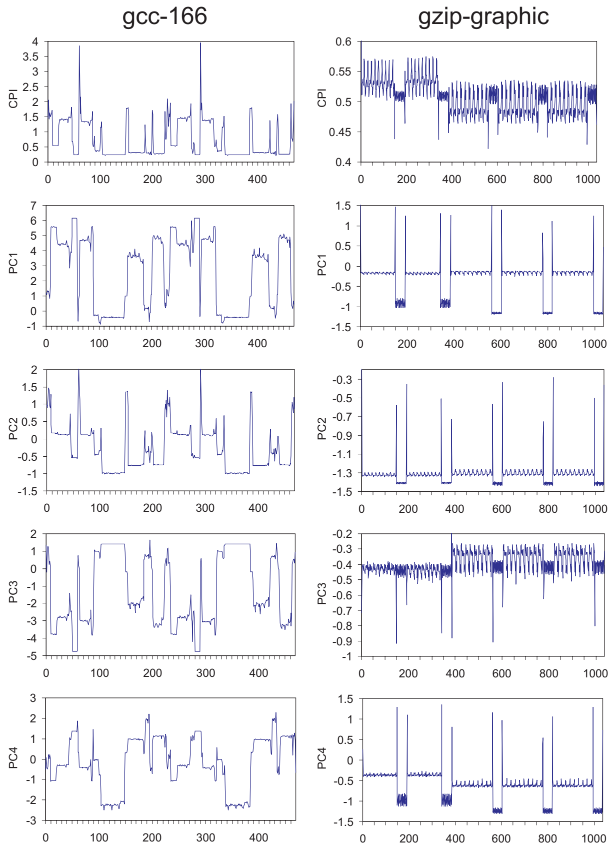

Fig. 3. Graphs showing how well the principal components track the phase behavior for gcc-166 on the left and gzip-graphic on the right: top graph in each column shows CPI as a function of the number dynamically executed instructions (in 100 millions); the remaining graphs show the values for the four principal components as a function of time.

other principal components.

In order for our workload reduction technique to work, we need to verify that the principal components correlate well with performance. This is illustrated in Figure 3 where CPI and the first four principal components are shown as a function of time for two benchmarks gcc-166 and gzip-graphic. These graphs clearly show there exists correlation between the principal component value and overall performance. Another way of looking at this data is to look at the receiver operating characteristic (ROC) [21] which evaluates the relationship between significant changes in CPI versus significant changes in the principal components. A true positive in the ROC is defined as a significant (5%) change in CPI that is identified by a significant change in the principal component's value. A false positive is defined as a significant change in the PC's value without a significant (5%) change in CPI. Example ROCs are shown in Figure 4 for gcc-166 and gzip-graphic—similar graphs are obtained for other benchmarks; the various dots in the graph show varying thresholds for defining a significant change in PC value. The higher the percentage of true positives and the lower the percentage of false positives, the better. The ROC clearly demonstrates that the principal components correlate well with performance. In other words, looking at the time-varying behavior of the principal components reveals the phase behavior of an application's execution. This reinforces that the principal components built from our collection of microarchitecture-independent characteristics will be useful for workload reduction.

An important application of PCA is that it allows for visualizing the workload space. Figures 5 and 6 show such a visualization for the SPECint and SPECfp benchmarks, respectively. These workload space plots show the 100M instruction intervals for all the benchmarks as a function of the first two principal components. The axes on all these figures are equal to each other which allows comparison between the benchmarks (at least within SPECint and SPECfp; we do not compare SPECint against SPECfp, since their principal components were computed separately). There are several interesting observations to be made from these graphs. First, some benchmarks, e.g. crafty and eon, do not show a big diversity across the 100M instruction intervals. Other benchmarks on the other hand, e.g. gcc, show a widely diverse behavior across the instruction intervals. This gives some intuition about the time varying behavior of these applications. Second, when interpreting the meaning of the principal components, we can gain insight into how benchmarks differ from each other according to the microarchitecture-independent characteristics. For example, a SPECint benchmark such as mcf positioned along a negative value of PC1 and a high value along PC2 tends to have relatively low ILP, a small number of local load strides, a high branch predictability, a large data stream working set and a small instruction stream working set.

B. *Workload reduction*

The data as they are obtained from PCA—the coefficients for the various execution intervals in the rescaled PCA space—now serve as input to the second step in our workload reduction method. The purpose of this second step is to apply cluster analysis and to determine a set of representative execution intervals *across all benchmarks and inputs*. The obtained representative 100M instruction execution intervals then are the simulation points of our reduced workload. Applying the cluster analysis on the SPECint2000 data set yields 148 simulation points. The error that is observed by comparing the CPI through complete benchmark simulation versus the CPI through the simulation of the obtained simulation points is only 1.11%. A similar result is obtained for estimating EPI (Energy Per Instruction): 1.05% error. This is shown as the per-phase uarch-independent results in Table III. For SPECfp2000 we obtain fewer simulation points than for SPECint2000, namely 84. The CPI and EPI prediction errors are only 0.87% and 0.90%, respectively.

Where do these simulation reductions come from? There are three major sources: (i) phase behavior within the execution of a single program-input pair, (ii) cross-input similarity for a given program, and (iii) cross-program similarity. Cross-input similarity comes from the fact that different inputs to a single program often result in significant parts of the entire execution to be common over various inputs. Cross-program similarity refers to execution intervals in different programs that exhibit similar execution characteristics. Figure 7 quantifies these three types of similarity for SPECint2000 and SPECfp2000. These pies show the fraction of simulation points that represent a set of execution intervals from (i) a single program-input

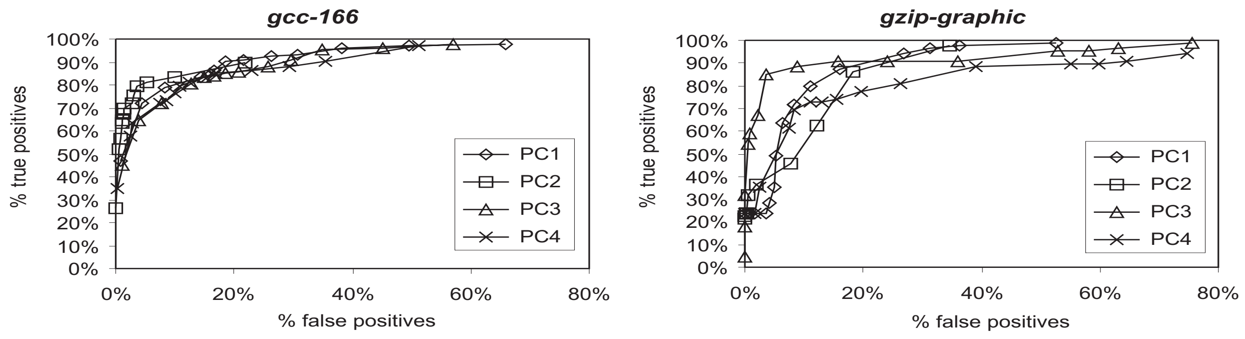

Fig. 4. ROC curve for gcc-166 and gzip-graphic: % true positives on the Y axis vs. % false positives on the X axis.

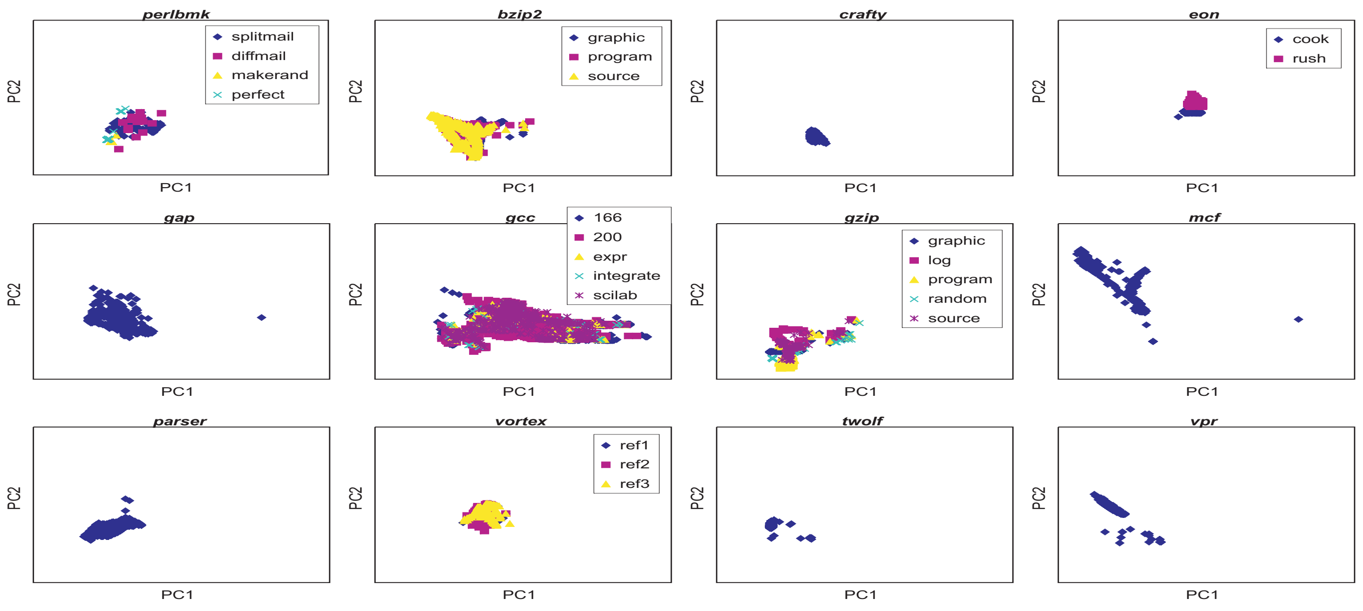

Fig. 5. The workload space for the SPECint2000 benchmarks.

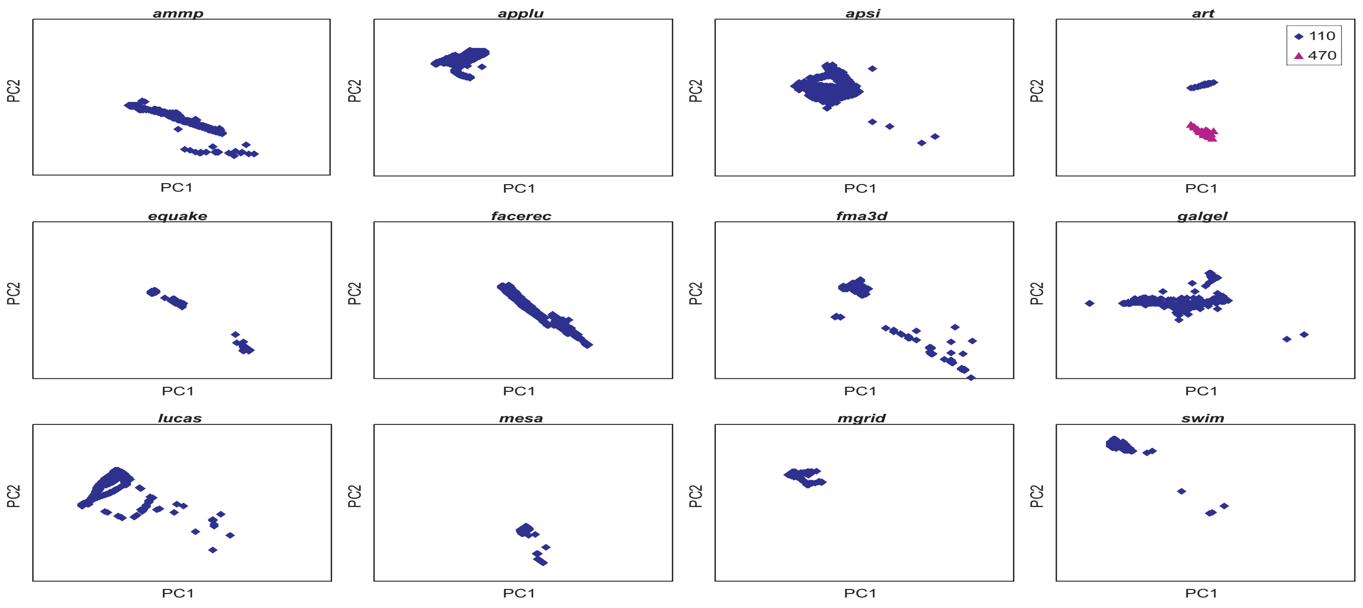

Fig. 6. The workload space for the SPECfp2000 benchmarks.

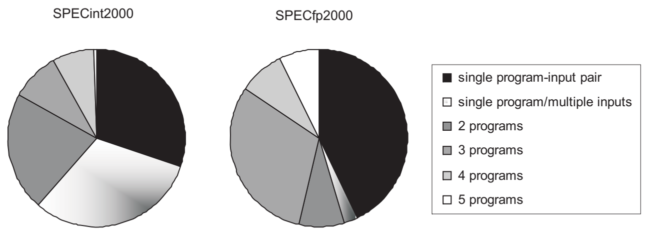

Fig. 7. Similarity classification of the SPECint2000 (left) and SPECfp2000 (right) simulation points.

pair, (ii) a single program and multiple inputs, (iii) two different programs, (iv) three programs, (v) four programs and (vi) five or more programs. For SPECint2000 and SPECfp2000, the fraction simulation points representing intervals from a single program-input pairs equals 30% and 43%, respectively. For SPECint2000, 31% of all simulation points represent a single program with multiple inputs—this is the fraction cross-input similarity. For SPECfp2000, this is only 2.4% of all simulation points—the reason is that only one floating-point benchmark comes with multiple reference inputs, namely art, see Table II. Cross-program similarity is observed at the phase level in a significant fraction of the simulation points, 38.5% for SPECint2000 and 54.8% for SPECfp2000. These results show that there exists an important fraction of cross-input and cross-program similarity. This paper exploits both cross-input and cross-program (next to intra-program) similarity to reduce the workload. SimPoint only exploits intra-program similarity. The reasons for cross-input and cross-program similarity are that the same parts of a program may be exercised across different inputs; similar initialization code, similar library codes and similar programming constructs and code generation schedules are being executed across programs.

C. Comparison to SimPoint

We now compare our reduced workload with the simulation points obtained using SimPoint [1], [2] which is described in section II-A—the maximum number of simulation points per benchmarks was set to 10 (MaxK=10) in the SimPoint software. The SimPoint results represent taking the simulation points for each program-input pair in the benchmark suite and simulating them to find the average performance of the overall workload. Comparing SimPoint against the newly proposed per-phase microarchitecture-independent workload reduction technique presented in this paper is somewhat difficult. Both approaches result in a different number of simulation points and a different accuracy guided by SimPoint. Therefore, comparing the number of simulation points for exactly the same accuracy, or vice versa, comparing the accuracy for exactly the same number of simulation points is not meaningful. SimPoint identified 233 simulation points for SPECint2000 and 126 simulation points for SPECfp2000. The CPI and EPI errors for SimPoint are very low for both SPECint2000 and SPECfp2000, see Table III. The level of accuracy for SimPoint is comparable to the level of accuracy obtained through our workload reduction approach.

Comparing now the number of simulation points between SimPoint and our reduced workload, we conclude that our workload is smaller with a reduction of a factor of 1.57 for SPECint2000 and a factor of 1.5 for SPECfp2000. The reason why our workload reduction approach attains fewer simulation points is due to the fact that we exploit cross-program, cross-input, and intra-program similarity. SimPoint on the other hand, only considers phase behavior for a single program-input pair (intra-program similarity).

D. Comparison to aggregate workload analysis

Previous work [4], [5] proposed a method to determine a set of representative program-input pairs from a large set of program-input pairs. In their analysis, the authors use aggregate characteristics (i.e. by averaging over the complete program execution) instead of the per-phase characteristics used in this paper. The characteristics used are either a mix of uarch-dependent and uarch-independent characteristics [4], or are a set of solely uarch-independent characteristics [5]. A reduced workload is then obtained by applying PCA and clustering on this data set.

We now compare our reduced workload to what can be obtained from an aggregate workload analysis. The results for the aggregate microarchitecture-dependent approach of [4] and the microarchitecture-independent approach of [5] are shown in Table III. To gather these results, the analysis in [4], [5] was used to reduce the number of program-input pairs to be simulated to represent the workload. Then for each program-input pair we used SimPoint to estimate the overall benchmark's execution. This was to provide a simulation time (number of instructions to simulate) that is comparable to our approach. The results show that the errors obtained from an aggregate analysis are typically higher (3.5% and 2% for SPECint and SPECfp, respectively) than for the per-phase analysis (1.1% and 0.8% for SPECint and SPECfp, respectively).

We find that by selecting samples from a selected set of representative program-input pairs in the aggregate analysis, the resulting reduced workload lacks important phase behavior from the other (non-selected) program-input pairs. In our approach, which takes a collection of representative samples from the complete workload, all important phase behavior is represented which results in a higher accuracy. In addition, a single set of simulation points for a workload using our approach can be used to calculate performance estimates for many different architecture configurations, since they are microarchitecture-independent.

E. Design space exploration

As mentioned before, an important motivation for using uarch-independent characteristics is the fact that the obtained reduced workload can be used for design space exploration. We now evaluate the accuracy of our reduced workload for design space exploration. To this end, we use our reduced workload approach to examine eight different architecture configurations for the SPECint2000 benchmarks. We examine varying the processor core resources, the branch predictor configuration and the cache sizes. We only examine eight configurations because we had to run each benchmark to completion for each configuration to get the baseline results—note this is exactly the problem we address through our reduced workload approach. For the small processor core

	Workload reduction method	CPI	EPI	#insns
SPECint2000	Per-phase uarch-independent workload reduction	1.11%	1.05%	14.8B
	SimPoint	0.80%	0.86%	23.3B
	Aggregate uarch-dependent workload reduction w/ SimPoint	3.65%	3.50%	13.1B
	Aggregate uarch-independent workload reduction w/ SimPoint	4.01%	3.52%	13.3B
SPECfp2000	Per-phase uarch-independent workload reduction	0.87%	0.90%	8.4B
	SimPoint	1.37%	1.34%	12.6B
	Aggregate uarch-dependent workload reduction w/ SimPoint	1.97%	0.11%	8.6B
	Aggregate uarch-independent workload reduction w/ SimPoint	2.00%	2.04%	8.5B

TABLE III

COMPARING PER-PHASE WORKLOAD ANALYSIS ON UARCH-INDEPENDENT CHARACTERISTICS VERSUS SIMPOINT VERSUS AGGREGATE WORKLOAD ANALYSIS ON UARCH-DEPENDENT AND MICROARCHITECTURE-INDEPENDENT CHARACTERISTICS FOR SPECINT2000 AND SPECFP2000.

resource machine we consider a 4-issue machine with a 32-entry window; the large processor core resource machine is 8-issue with a 128-entry window. The small branch predictor has 2K-entry tables; the large one 8K-entry tables. The small cache hierarchy consists of a 16KB L1 D-cache, an 8KB L1 I-cache and a 1MB L2 cache; the large cache hierarchy consists of a 64KB L1 D-cache, a 32KB L1 I-cache and a 4MB L2 cache. Obviously, we also assume different access latencies for the small and large cache configurations.

Our reduced workload approach computes one set of simulation points using the micro-architecture independent characteristics. Since this set of simulation points was chosen independent of the underlying microarchitecture we can use it for all 8 architecture configurations we simulated for our design space exploration.

Figure 8 (top graph) compares the CPI numbers obtained from complete benchmark suite simulation versus the ones obtained from the per-phase uarch-independent reduced workload. The error for the reduced workload is very small over all processor configurations (no more than 6%). These results were gathered without warmup of the caches and branch predictors—this explains the somewhat higher error rates compared to the results presented before which assumed perfect warmup (the focus in the previous sections was on the error derived from sampling). This does not affect the overall conclusions that can be taken from these graphs though.

More importantly than absolute accuracy, these CPI results show that during design space exploration the relative performance between the various processor configurations is estimated very well by the reduced workload. We believe in early stages of a microprocessor design process, relative accuracy is even more important than absolute accuracy in order to make early design tradeoff decisions. Indeed, computer architects want to quantify how overall performance increases or decreases as a function of a microarchitectural parameter.

To quantify the relative accuracy of the reduced workload we first define relative error. We define the relative error of the reduced workload for predicting the performance between two processor configurations A and B as: $RE_{A,B} = |\frac{CPI_{A,entire}}{CPI_{B,entire}} - \frac{CPI_{A,reduced}}{CPI_{B,reduced}}|$. This means we measure the relative difference in quantifying performance speedup or degradation between the entire workload simulation and the reduced workload simulation. With the results from Figure 8 we can compute the relative accuracy along the three microarchitectural dimensions that we considered: processor core resources, branch predictor sizes and cache sizes. These results confirm that the reduced workload achieves a high relative accuracy with

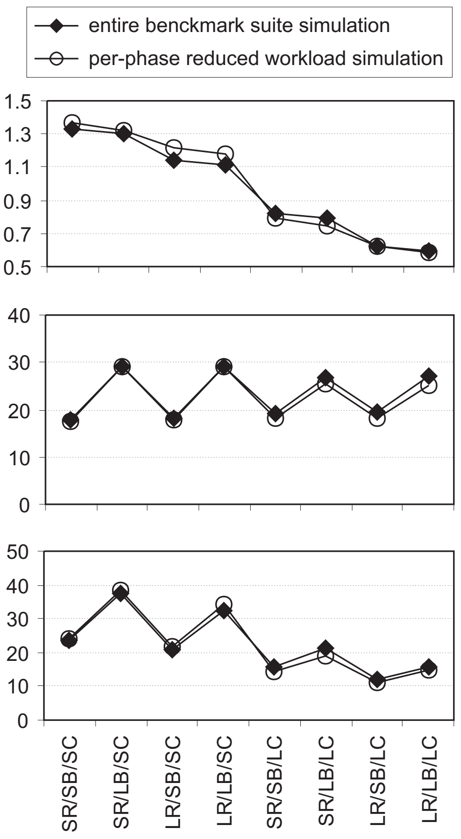

Fig. 8. Design space exploration using the per-phase uarch-independent reduced workload: top graph shows CPI, middle graph shows EPI and bottom graph shows EDP (Energy Delay Product). 'SR' and 'LR' stands for small vs. large processor core resource; 'SB' and 'LB' mean small vs. large branch predictor configuration; 'SC' and 'LC' mean small vs. large cache hierarchy. Configurations are sorted by decreasing CPI.

relative errors below 4.16% along the cache size dimension, 3.54% along the processor core resource dimension and 1.79% along the branch predictor dimension.

We can do a similar analysis while focusing on energy consumption, see Figure 8 (middle graph). The relative errors for the EPI numbers are below 5.51% along the cache size dimension, 2.66% along the processor core resource dimension and 2.12% along the branch predictor dimension. When combining the CPI and EPI estimates, we can compute the energy-delay product (EDP): $EDP = EPI \cdot CPI$. Figure 8 (bottom graph) shows EDP when simulating the entire benchmark suite versus when simulating the per-phase reduced workload. This graph clearly shows that the reduced workload tracks the entire workload fairly well. Both per-phase reduced workload simulation and entire workload simulation identify the same optimal microarchitecture with the minimal EDP metric, namely the large processor core configuration with the large cache and small branch predictor configuration. We thus conclude that the reduced workload can be used for accurate and efficient processor design studies.

VI. Conclusion

In this paper we have applied taking a set of microarchitecture independent characteristics to find similar intervals of execution across different programs and inputs. We examined finding similar behaviors between different programs at the level of intervals (phases) whereas the prior SimPoint approach only found similarities between intervals within a single program. To do this we had to look at microarchitecture-independent characteristics and code-independent characteristics, we could not just take the basic block vectors used in SimPoint. Therefore, the characteristics we focused on finding similarities for are instruction mix, register dependencies, working set sizes, etc. These were used to find a set of representative simulation points independent of the microarchitecture across a set of benchmarks and inputs. These simulation points can then be used to represent the complete benchmark suite to accurately guide design space exploration.

We showed that performing benchmark suite reduction on a per phase level across all of the benchmark suite reduced simulation 1.5 times over using SimPoint with roughly the same error rate. We showed that when performing benchmark suite reduction that we were able to achieve a lower error rate by exploiting per-phase interval behavior over the prior aggregate approach [4], [5]. Since the reduced workload simulation points were derived using the microarchitecture-independent metrics we showed that they can be used across different architecture designs, which is needed to guide design space exploration.

Overall, the reduced workload simulation points generated from our approach provide a nice reduction in simulation time over SimPoint for full workload simulation, and can be used for design space exploration.

Acknowledgements

We would like to thank the anonymous reviewers for providing helpful comments on this paper. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research – Flanders (Belgium) (F.W.O. Vlaanderen). This work was also supported by Ghent University, IWT and HiPEAC. 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] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT-2003)*, Sept. 2003, pp. 244–256.

[2] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 2002, pp. 45–57.

[3] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-30)*, June 2003, pp. 84–95.

[4] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Quantifying the impact of input data sets on program behavior and its applications," *Journal of Instruction-Level Parallelism*, vol. 5, Feb. 2003, `http://www.jilp.org/vol5`.

[5] 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)*, Mar. 2005, pp. 10–20.

[6] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, Sept. 2001, pp. 3–14.

[7] H. Vandierendonck and K. De Bosschere, "Fragile and eccentric benchmarks," in *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2004, pp. 2–11.

[8] ——, "Experiments with subsetting benchmark suites," in *Proceedings of the Seventh Annual IEEE International Workshop on Workload Characterization (WWC)*, Oct. 2004, pp. 55–62.

[9] L. Eeckhout, A. Georges, and K. De Bosschere, "How Java programs interact with virtual machines at the microarchitectural level," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, Applications and Systems (OOPSLA)*, Oct. 2003, pp. 169–186.

[10] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Designing workloads for computer architecture research," *IEEE Computer*, vol. 36, no. 2, pp. 65–71, Feb. 2003.

[11] J. J. Yi, D. J. Lilja, and D. M. Hawkins, "A statistically rigorous approach for improving simulation methodology," in *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Feb. 2003, pp. 281–291.

[12] G. Hamerly, E. Perelman, and B. Calder, "How to use SimPoint to pick simulation points," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 25–30, Mar. 2004.

[13] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of larhe Intel Itanium programs with dynamic instrumentation," in *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO-37)*, Dec. 2004, pp. 81–93.

[14] 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)*, Dec. 1992, pp. 236–245.

[15] 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)*, Mar. 2004.

[16] 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)*, Oct. 1996, pp. 128–137.

[17] R. A. Johnson and D. W. Wichern, *Applied Multivariate Statistical Analysis*, 5th ed. Prentice Hall, 2002.

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

[19] D. C. Burger and T. M. Austin, "The SimpleScalar Tool Set," Computer Architecture News, 1997, see also `http://www.simplescalar.com` for more information.

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

[21] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, Dec. 2003, pp. 217–227.