Discovering the Runtime Structure of Software with Probabilistic Generative Models

Anonymous Author(s)
Affiliation
Address
email

Abstract

Modern computer systems have become so complex that understanding and predicting the performance of programs is a significant challenge. We address the problem of estimating the efficiency of a program on a hypothetical hardware architecture, typically measured by cycles per instruction executed (CPI). Simulating the entire program at the hardware level is extremely computation intensive. Consequently, researchers have resorted to picking portions of the program execution that are considered typical, called *simulation points*, and extrapolating from detailed simulation of these points to estimate CPI of the program as a whole. Simulation points are chosen by running the program at the software level and collecting statistics such as counts of basic code blocks executed, resulting in a time-varying sequence of basic-block vectors. The state-of-the-art algorithm for selecting simulation points uses k-means clustering of basic-block vectors. We propose an alternative class of probabilistic models, which include hidden Markov models (HMMs) and multinomial mixture models. We also explore techniques for reducing the dimensionality of the basic-block vectors, picking simulation points, and estimating CPI given the simulation points. After exploration of an enormous search space, we found no model that significantly outperforms k-means for picking simulation points, a result we find particularly surprising given that we expected time to provide a useful additional constraint to the HMM. Nonetheless, the HMM makes several potentially important contributions for understanding program behavior—by parsing the execution trace into discrete phases and discovering a finite-state model that characterizes phase-transition dynamics—and for run-time program optimization—by phase prediction.

Designing modern computer systems is a difficult optimization problem. Engineers must assess the trade-offs involved when dividing the on-die real estate (i.e. transistors) among L1 cache, branch prediction logic, arithmetic-logic, floating-point, and vector processing functional units, etc., in order to maximize a certain objective function. Typically, the objective is to maximize instructions executed per second; however, maximizing performance per watt is also very common. Because no analytic means is available to estimate the relationship between a particular microprocessor architecture and performance, it is necessary to *profile* a program running on a proposed architecture either through hardware instrumentation or software simulation. Profiling involves collecting runtime metrics—such as clock cycles per executed instruction (CPI), power consumption, cache miss rate, or branch prediction accuracy—for a particular suite of programs, such as the Standard Performance Evaluation Corporation (SPEC) benchmarks—a standard benchmarking suite designed to approximate the demands of various of modern computing tasks. The runtime metrics for a given program are dependent on both the functional specifications of the microarchitecture and the program input.

Although profiling could be done in near real time by polling hardware counters, an experimental microarchitecture would need to be re-fabricated every time an engineer tweaked the design.

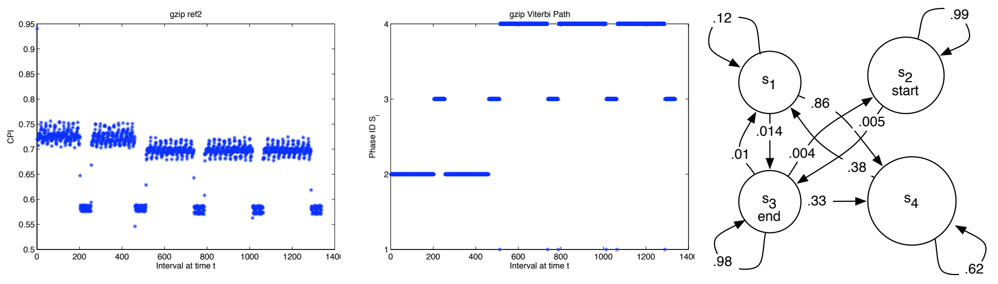

Figure 1: (left) graph of CPI as a function of time interval for the program 164.gzip with input ref2, (center) phase behavior inferred by an HMM with four states for 164.gzip, (right) Finite state machine representation of the phase behavior

Because re-fabrication is impractical, profiling is performed by simulating a program on a software-implemented microarchitecture. Simulation can increase the runtime of the program by many orders of magnitude. Consequently, the time cost of simulation can seriously restrict the exploration of experimental architectures and profiling some modern programs is completely impractical. For instance, simulating the SPEC 2000 suite on a cycle-accurate out-of-order execution core (i.e. a software implemented version of a microprocessor's functional units and resources) can take months of computing time. The lack of a practical source of runtime profile data retards programming and microarchitectural design.

Fortunately, there is a viable alternative; a profile can be estimated by performing detailed simulation for only certain brief intervals of the executing program. SMARTS [11] is a commonly used undirected scheme that profiles a random, unbiased sample of the program/input. However, SMARTS is not very efficient because it requires a large number of samples to achieve estimates that accurately reflect true program behavior.

In response to SMARTS and other early attempts to simulate an abridged version of the program, a technique called SimPoints [8, 9, 3] was developed. SimPoints attempts to obtain samples of the program—*simulation points*—that are characteristic of larger portions of program execution by exploiting the *phase* structure of a program. The idea underlying phase structure is that the execution dynamics of a large program can be understood in terms of a relatively small set of distinct patterns of program behavior. The temporal execution trace is broken into fixed-duration intervals, and the goal is to assign a phase label to each interval such that the program behavior across all intervals with the same label is similar (See Figure 1). The notion of 'similarity' depends on the particular metrics used to describe program behavior. However, when different metrics are used to infer phase structure, phase boundaries coincide across the metrics [1]. Dividing a program into phases allows a simulation point to be chosen for each phase that represents the phase as a whole. The goal is to extrapolate an accurate profile of the entire program's execution from a relatively small set of simulation points.

Two approaches have been proposed for discovering phases. *Graph-based* algorithms utilize the structure of a program—as represented by the compiler-generated call-loop graph—to identify coherent blocks of code. *Execution-based* algorithms define phases dynamically via observation of the time-varying dispatched instruction execution *trace* of a program (i.e. a log of the sequence of instructions executed by the microprocessor). The dispatched instruction trace reflects key aspects of the program execution, such as branch executions and load/store operations. It is hardware independent given the instruction set (e.g. x86, PowerPC). Thus, in contrast to performance metrics such as CPI or cache miss rates, the instruction trace can be collected at little cost using fast software instrumentation techniques. This trace is typically divided into disjoint, consecutive, fixed-duration *intervals*. Each interval is classified as belonging to one of k unique phases. Often, the dynamically defined phases will have a close correspondence with phases defined in terms of the program source. For instance, the execution of a loop unrolls in the program trace to a sequence of intervals that may exhibit periodic behavior. Thus, by profiling only a single period of the loop (the simulation point of the phase), both graph- and execution-based approaches can provide accurate runtime metrics at a significantly reduced execution time. However, graph-based algorithms have not been extensively studied because they require consideration of a combinatorial (recursive) set of parses of the program structure. Graph-based algorithms also require the program's source code. State-of-the-art algorithms are all execution based.

The predominant statistic used in execution-based algorithms is a *Basic Block Vector* (BBV). A BBV is a frequency vector of the basic blocks within an interval of instructions. A *basic block* is a sequence of instructions that has a single entrance and exit point. For instance, if a program contains five unique basic blocks, then each interval is processed into a BBV that has five dimensions, each of which counts the number of times a specific basic block is observed during an interval. Research has shown that program behavior on metrics such as CPI, cache misses rate, etc., are closely related to the basic blocks being executed [8].

SimPoints, the standard execution-based algorithm in the field and in common use by many groups [6] [ANOTHER], uses *k*-means to cluster BBVs to identify phases and simulation points. Considering that *k*-means is a primitive clustering algorithm relative to the state of the art in machine learning, several recent papers have attempted to cast phase discovery in a probabilistic context. Because an interval of program execution is represented by a basic-block frequency vector, it is sensible to treat a BBV as a phase-conditional sample from a multinomial probability distribution. This generative model can be used to infer the underlying phases from BBV observations. [7] and [2] attempted to use such a multinomial-clustering model. However, because of the high dimensionality of the BBVs (for large programs, on the order of 50,000), the authors performed a random projection of the BBVs to obtain a lower-dimensionality representation. Unfortunately, this projection yielded vectors that were impossible to interpret (e.g., negative values and no preservation of vector sum), and therefore the authors incorporated an ad hoc normalization scheme before applying the clustering model. Consequently, the results were difficult to interpret in terms of the original formulation, which is what gave the probabilistic model its original appeal. In the next section we outline a more principled application of two probabilistic generative models.

1 Probabilistic generative models

We explore a family of models that diverge from and improve on previous work in three respects. (1) We avoid the problematic dimensionality reduction scheme of [7] and [2] by using the BBV without projection. The costs and benefits of dimensionality reduction have not been evaluated with respect to model performance. We also propose a random projection technique that allows for an interpretation of the projected vectors as counts. (2) We consider a model that takes into account the temporal structure of the program trace. Each BBV is an observation in a *sequence*; thus, there is potential for hidden Markov models (HMMs) to discover and exploit the sequential structure. In the HMM formulation, the current phase is represented by the hidden state. The additional constraints on phases imposed by modeling phase transitions may yield a better representation of the phases themselves. We compare a multinomial HMM against an atemporal multinomial mixture model (MMM) to evaluate the benefit of incorporating temporal structure. (3) Given a model of the phase structure of a program, one still has the challenge of picking representative samples of each phase for detailed simulation (i.e., the simulation points). We explore several execution-based schemes for picking simulation points that exploit the probabilistic nature of our models.

1.1 Models: multinomial HMM and multinomial mixture model

An HMM is a probabilistic generative model of a sequence of observations. A naïve approach to modeling phase behavior is to identify a unique symbol of language Σ with each basic block, and treat the HMM as emitting symbols. However, a single basic block contains only weak information about the state of the program. To provide stronger evidence, we define an HMM that emits not one but a *set* of u symbols at each time step. This set of symbols is equivalent to a BBV, and thus one time step in the HMM models an interval of program execution. With this formulation, we obtain an HMM with a multinomial observation distribution. Given a sequence of BBVs, inference on this HMM yields a hidden state representation that clusters BBVs. It is sensible to identify the hidden state with a phase because all BBVs assigned to the same state spend about the same amount of time in the same sections of code. Although we lose some temporal information by treating the observations as BBVs—the multinomial probability disregards the ordering of basic blocks within an interval—coarser temporal structure exists via the sequence of intervals, and the HMM may well exploit this coarser structure to constrain the representation of phases.

We also implement a multinomial mixture model (MMM) similar to that proposed by [7, 2]. The MMM can be viewed as a special case of our HMM with uninformative transition probabilities. The states of the MMM are based solely on clustering of BBVs, whereas the states of the HMM also incorporate a temporal-context constraint: BBVs are clustered not only by their similarity but by the similarity of the temporal context in which they occur. It remains an empirical question as to whether this additional constraint serves to obtain phase representations that are more useful for estimating program profiles.

1.2 Dimensionality reduction

Both [3] and [2] apply a random projection to the BBVs in order to reduce the dimensionality of the data and speed up computations. Although there are other techniques to reduce the dimensionality of the dataset, such as PCA, a [nearly orthogonal?] random projection is both computationally efficient and reinforced by the Johnson-Lindenstrauss Lemma [4], which states that an orthogonal projection of a set of points into some low dimensional subspace will approximately preserve the relative distance between the points [4]. Unfortunately, such projections through the origin can yield negative values, which precludes interpreting the elements of the projected BBV as counts. Furthermore, projection does not preserve the total count—L_1 norm—of the BBV. Because the total count reflects the amount of evidence in the BBV, this information should be retained in the projected BBV.

Previous attempts to model reduced-dimensionality BBVs via a multinomial distribution have used ad hoc random projection schemes that preserve positive counts and the L_1 norm. [7] uses an orthogonal projection matrix R chosen sparsely from $\pm$. Then, in order to recover an interpretation of the projection as a vector of counts, the vector elements are additively shifted so that the minimum element is zero. [7] avoids this problem by choosing R from a uniform [0 1] distribution. However, both schemes require renormalizing the reduced-dimensionality BBV, or *RBBV*, such that the L_1 norm of the BBV is preserved. Because the renormalization is different for each BBV, the RBBV can not be interpreted as a projection of the BBV.

We propose a novel projection that satisfies the constraints of non-negative elements and conserving the L_1 norm. We achieve these constraints with a random projection matrix R whose elements are chosen from the uniform [0, 1] distribution, and whose rows are normalized to sum to one (assuming that R is a $d \times d'$ where d is the dimensionality of the BBV and d' is the desired reduced dimensionality, and R is premultiplied by the BBV). This projection matrix effectively redistributes the counts in the d-bin BBV to the *d'*-bin reduced-dimensionality vector, denoted *RBBV*. The row constraint on *R* ensures that the total evidence of each observation is preserved, and the non-negative elements ensures no negative counts.

In our simulation studies, we compare using no dimensionality reduction (FULLDIM), our novel projection scheme (PROJREWEIGHT), and the random scheme used by [7] (PROJRAND).

1.3 Using model to estimate runtime profile metrics

Given a model that clusters observations according to the hidden state (phase), we wish to choose representative observation intervals on which to perform detailed simulation (the simulation points), and then estimate the runtime metrics of the un-simulated intervals using these simulation points and the state posteriors, $P(S_t|\mathbf{O})$, where S_t denotes the hidden state at interval t, and $\mathbf{O}$ is the observation sequence.

We have explored eight different techniques for determining simulation points, each of which involves finding a time t for each state s that satisfies some criterion. The techniques and criteria are as follows MAXLIKELIHOOD1: $t = \text{argmax}_\tau P(o_\tau|v_\tau = s)$; MAXLIKELIHOOD2: $t = \text{argmax}_\tau P(o_\tau|S_\tau = s)$; OBSMODELDIST1: $t = \text{argmin}_\tau ||n_\tau - P(o_\tau|S_\tau = v_\tau)||_1$; OBSMODELDIST2: $t = \text{argmin}_\tau ||n_\tau - P(o_\tau|S_\tau = v_\tau)||_2$; OBSMODELDIST3: $t = \text{argmin}_\tau \text{KL}(n_\tau||P(o_\tau|v_\tau = s))$; OBSMODELDIST4: $t = \text{argmin}_\tau \text{KL}(n_\tau||P(o_\tau|v_\tau = s)) + \text{KL}(P(o_\tau|v_\tau = s)||n_\tau)$; CLUSTERCENTER: $t = \text{argmin}||o_\tau - c_s||_2$, where $c_s = 1/|T_s| \sum_{i \in T_s} o_i$; MAXPOSTERIOR: $t = \text{argmax}_\tau P(S_\tau = s|o_\tau)$. In this notation, v_τ denotes the state assigned by

Viterbi for interval τ, S_τ denotes the state at τ, o_τ denotes the observation at τ, $n_\tau = o_\tau/||o_\tau||$ denotes the normalized observation, and T_s is the set of all time steps t for which $v_t = s$.

Given the choice of a simulation point, the runtime profile is obtained for that interval, producing a metric M_s for each state s. We use two techniques for estimating the profile metric for all time steps t' that were not simulated: (EPM1) use $M_{v_{t'}}$ (EPM2) integrate out over state uncertainty and use $\sum_s P(o_{t'}|S_{t'} = s)M_s$. Although we were optimistic that the marginalization scheme would yield better results, it did not. Therefore, the results we report use only the maximum likelihood technique EPM1.

2 Methodology

2.1 Data generation

Our data set consists of 19 SPEC2000-int benchmarks, each of which is a particular program running on a particular input, sampled at an interval-size of 100 million instructions for consistency with [3, 7, 2]. Our data consists of two time series of equal length: the observation trace and the baseline metric trace. The observation trace is the training sequence for our models and consists of BBVs. Our models are used to predict a particular metric; in the results we report here, we focus on the metric most commonly used, CPI. The baseline metric trace provides actual CPI values used to evaluate the models. Both traces are generated with the SimpleScalar simulators sim-fast and sim-outorder respectively, modified to output the desired data. SimpleScalar is a popular microarchitecture simulator used by both [3] and [2]. sim-fast counts the number of times each basic block was executed in an interval (the BBV), while sim-outorder outputs the average CPI for each interval. These program traces range from hundreds of billions to trillions of instructions and many industry applications may have more. The language size, $|\Sigma|$, of these programs ranges from 871 unique basic blocks for 164.gzip to 40,066 for 176.gcc.

2.2 Model Implementation

We implement the multinomial HMM by modifying the discrete HMM package developed by [5]. The models are initialized randomly and trained on both projected and unprojected data. The dimensionality of the random subspace is determined by a simple rule proposed by [7]: $|\Sigma| < 2000 \rightarrow d' = 15$, $2000 \leq |\Sigma| \leq 9999 \rightarrow d' = 50$, $10000 \leq |\Sigma| \leq 19999 \rightarrow d' = 75$, $|\Sigma| \geq 20000 \rightarrow d' = 100$. Although somewhat arbitrary, this method reduces the dimensionality of the datasets by a factor of ˜100. To mitigate the effects of local optima during EM training, each model is run for fifteen random restarts and the model with the highest log likelihood is then used to generate simulation points and estimates of CPI. We used an existing implementation of SimPoints [9], which also performs fifteen random restarts of the *k*-means algorithm.

3 Results

We conducted a set of simulation experiments exploring models that vary on five dimensions: (1) whether or not temporal structure is modeled (HMM vs. MMM), (2) representation of data (FULLDIM, PROJREWEIGHT, and PROJRAND), (3) number of phases (latent states, chosen from 4, 8, 16, 32, and 64), (4) techniques for determining simulation points (MAXLIKELIHOOD1, MAXLIKELIHOOD2, OBSMODELDIST1, OBSMODELDIST2, OBSMODELDIST3, OBSMODELDIST4, CLUSTERCENTER, MAXPOSTERIOR), and (5) techniques for estimating the performance metric CPI given the simulation points (EPM1, EPM2). All models were evaluated on the task of estimating CPI for the 20 program-input pairs we selected from the SPECint 2000 benchmark suite. We compare our models to the SimPoints algorithm.

Before reporting results, we address an issue in interpreting the results. In many of the probabilistic models constructed, the full number of latent states was not used in prediction. We explored a variety of techniques for avoiding local optima including reinitializing states that have a small influence on the model likelihood and splitting states (Split State Viterbi [10]). However, the models were resistant to this heuristic assistance: we found very little improvement in log likelihood, suggesting little benefit of using the full complement of states for modeling the data. The challenge goes beyond

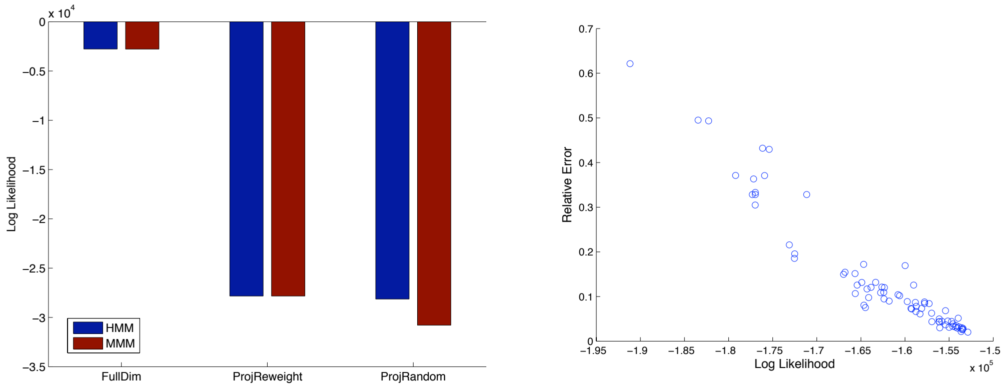

Figure 2: (left) mean log likelihood for the models (HMM, MMM) paired with a data representation (FULLDIM, PROJRAND, and PROJREWEIGHT), averaged over models with different numbers of states; (right) Log likelihood vs. error in the CPI estimate for fifteen random restarts of the HMM and the MMM for 176.gcc.ref2 with the FULLDIM representation, averaged over models with different numbers of states

local optima in training: even when models are trained to use all states, the techniques for selecting simulation points and estimating the performance metric may not exploit all states. That is, the number of simulation points may be fewer than the number of states in the model. Consequently, we simply decided to report performance as a function of the average number of unique simulation points used for a given size model.

Figure 2 (left) shows mean log likelihood on the training data for six classes of models (HMM and MMM, each with the three data representations), averaged over number of states available. As expected, log likelihood scores on the training data increase for all models as a function of the number of states allowed. The Figure shows that log likelihood scores are higher for the models without dimensionality reduction (FULLDIM) than with (PROJREWEIGHT and PROJRAND), not surprising considering that the FULLDIM models have far more free parameters to fit. The Figure also shows that log likelihood scores for the HMM and MMM are comparable, suggesting that there may not be much benefit in modeling the temporal structure of the data.

We train our models to maximize data likelihood, but ultimately, we are interested in using the models to accurately estimate the CPI performance metric. Generally, it is not sensible to train on one objective and evaluate on a different objective, but our domain is sufficiently complex that we have not succeeded in developing an algorithm that directly optimizes CPI estimates. Consequently, we need to provide some empirical evidence that optimizing data likelihood also tends to improve CPI estimates. Figure 2 (right) is a scatterplot of data log likelihood versus error in the CPI estimate (to be defined below), where each point in the scatterplot is an instantiation of a particular model with CPI estimated using OBSMODELDIST1 and EPM1. A strong correlation is observed between log likelihood and the quality of the CPI estimate. We see this correlation with every CPI-estimation technique. This strong correlation rationalizes the use of data likelihood for model selection. We do not observe overfitting in the scatterplot, nor should one necessarily expect overfitting given the two distinct criteria.

To evaluate model-based estimates of CPI, past researchers [3, 7, 2] have used the overall relative error, $E^R = |\sum_{t=1}^T c_t - \sum_{t=1}^T \hat{c}^t| / \sum_{t=1}^T c_t$, where c_t is the true CPI value for time t, and $\hat{c}_t$ is the estimated value from an algorithm. However, this metric could easily obscure local program activity by averaging out overestimates and underestimates. We therefore propose a finer-grained measure, mean interval-wise relative error, $E^I = 1/T \sum_{t=1}^T |c_t - \hat{c}_t| / c_t$. Because $\sum_t |c_t - \hat{c}_t| \geq |\sum_t c_t - \hat{c}_t|$, a small E^I error implies a small E^R error, although the reverse is not necessarily true. Therefore, we use E^I as a more informative measure.

The bottom line of our simulation studies is that no approach we tested significantly outperformed SimPoints. Figure 3 shows the mean and standard error of E^I for all models and data representations, for two different schemes for selecting simulation points and for estimating CPI given the

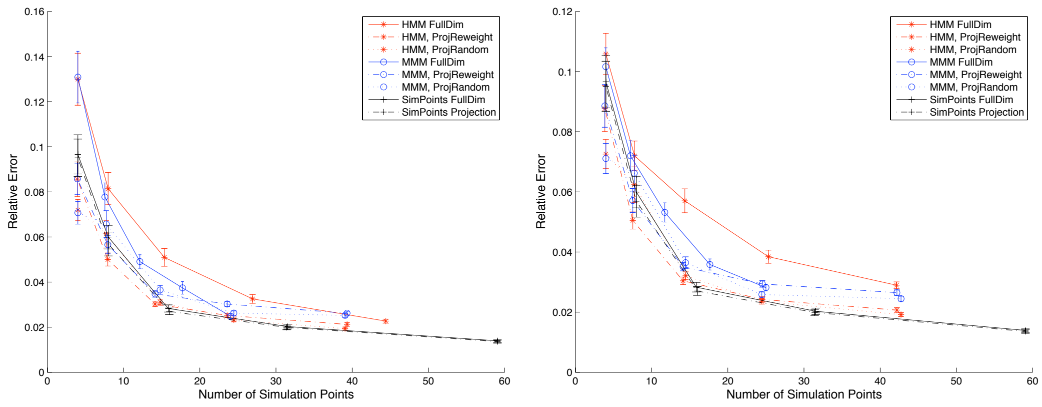

Figure 3: Error in estimate of CPI over the entire test suite as a function of the number of states (simulation points) in a model, for HMM and MMM models coupled with three different data representations. The left panel computes CPI estimates using the combination of CLUSTERCENTER and EPM1; the right panel computes CPI estimates using the combination of MAXPOSTERIOR and EPM2. The results from SimPoints (with both reduced and full dimensionality representations) is shown for comparison. Although the graphs are cluttered, the basic result to observe is that SimPoints outperforms all of the alternatives.

simulation points. The graph on the left estimates CPI using CLUSTERCENTER and EPM1 the graph on the right uses MAXPOSTERIOR and EPM2. (We explored 16 CPI-estimation schemes in total, but the other 14 yield similar outcomes.) Although the graphs are cluttered, the main observation is that none of the models we tested outperform SimPoints, which is the lowest curve in both graphs.

The CLUSTERCENTER-EPM1 combination (left graph) obtains the best error estimates of all 16 CPI-estimation schemes. However, this combination performs no better than SimPoints. Had we outperformed SimPoints, we would have needed to perform additional validation testing: the training data was used for model selection (i.e., choosing CLUSTERCENTER and EPM1) and therefore should not be used as well for comparing performance with SimPoints. However, even our optimistic error estimates suggest that SimPoints outperforms all of the probabilistic models.

The MAXPOSTERIOR-EPM2 combination (right graph) is interesting because it uses a Bayesian method to estimate CPI, via computation of state posteriors and computing an expected CPI by integrating over the uncertainty in the posteriors. Nonethless, this scheme does not yield the best results.

4 Discussion

The template for a machine-learning application paper is to propose a model that is novel to a domain and show that it outperforms existing models in the domain. Our research fails to match the template. We proposed several theoretical ideas that are novel to the computer-systems performance-analysis community, including: (1) the use of hidden Markov models to capture temporal structure in a program's execution, (2) a scheme for dimensionality reduction of a count vector that preserves an interpretation of the vector as a sample of a multinomial random variable, and (3) novel, probability-based schemes for selecting simulation points and estimating CPI.

Although we conducted a systematic exploration of a large model space, we found no model that outperforms SimPoints on the task of estimating CPI. In particular, the HMM models that encode phase transitions performed almost identically to the MMM models that do not infer the phase at the current time with respect to the phase at the previous time.Clearly, the transition constraint on phases offered by the temporal sequence is weak relative to the constraint offered by the observation vector at each time.

Our findings are consistent with other research that supports the robustness of SimPoints [7, 2, 3]. Nonetheless, negative results are important for directing the research efforts of the field, and for posing a challenge to the machine learning community. It is disappointing when novel and sensible approaches do not outperform existing techniques, but we should not be entirely surprised: k-means has a long history of application in machine learning. We suspect the reason has to do with the use of the L_2 norm for estimating the relationships among BBVs. The one technique we used that took advantage of an L_2 norm for determining BBV prototypicality, CLUSTERCENTER, was a component of the best solution we discovered. It is also disappointing that the temporal constraints of the domain could not be exploited by the HMM to yield better phase decompositions. We suspect the reason for the failure of the HMM is that the BBVs clusters are so well separated that temporal constraints play a small role in inferring phase (state) from observations. The separation of the clusters is another explanation for the success of k means.

Our HMM models make a potentially important contribution to the field of computer systems analysis. The HMM's state transition matrix, which defines the state to state transition probabilities, can be viewed as a finite state machine detailing where and how a program spends its time. We illustrate the transitioning behavior of 164.gzip, a relatively simple program that simply compresses and decompresses an input stream. Referring to Figure 1, we can see, the program spends most of the time in states s_2 and s_4. Furthermore, we can see from the state to state transition probabilities that states s_2, s_3, and s_4 are very stable, with high self-transitional probabilities, while state s_1 is either very short lived or is picking up on noise from the phase modeled by s_4. With the knowledge that a phase is stable, the HMM can also be used in an real-time context to predict upcoming phase transitions. Phase prediction has utility in computer systems research for dynamic code optimization. A non-probabilistic algorithm like SimPoints cannot be readily used for modeling and predicting phase structure.

Acknowledgments

This research was supported by NSF BCS 0339103 and NSF CSE-SMA 0509521.

References

[1] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. *International Conference on Parallel Achitecture and Compilation Techniques*, 2003.

[2] G. Hamerly, E. Perelman, and B. Calder. Comparing multinomial and k-means clustering for simpoint. *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*, 2006.

[3] G. Hamerly, E. Perelman, J. Lau, B. Calder, and T. Sherwood. Using machine learning to guide archecture simulation. *Journal of Machine Learning Research*, pages 343–378, 2006.

[4] W. Johnson and J. Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Int Conference in modern analysis and probability*, pages 189–206, 1984.

[5] K. Murphy. Hidden markov model toolbox for matlab. *http://www.cs.ubc.ca/ murphyk/Software/HMM/hmm.html*, 2005.

[6] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. *International Symposium on Microarchitecture*, 2004.

[7] K. Sanghai, T. Su, and D. Kaeli. A multinomial for fast simulation of computer architecture designs. *KDD*, pages 808–813, 2005.

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

[9] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *The International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[10] S. M. Siddiqi, G. J. Gordon, and A. W. Moore. Fast state discovery for hmm model selection and learning. *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2007.

[11] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. Smarts: Accelerating microarchitectural simulation via rigorous statistical sampling. *International Symposium on Computer Architecture*, 2003.