Multilevel Phase Analysis

WEIHUA ZHANG, Shanghai Key Laboratory of Data Science, Software School, and State Key Laboratory of ASIC & System, Fudan University, Shanghai, China
JIAXIN LI and YI LI, Parallel Processing Institute, Fudan University, Shanghai, China
HAIBO CHEN, School of Software, Shanghai Jiao Tong University, Shanghai, China

Phase analysis, which classifies the set of execution intervals with similar execution behavior and resource requirements, has been widely used in a variety of systems, including dynamic cache reconfiguration, prefetching, race detection, and sampling simulation. Although phase granularity has been a major factor in the accuracy of phase analysis, it has not been well investigated, and most systems usually adopt a fine-grained scheme. However, such a scheme can only take account of recent local phase information and could be frequently interfered by temporary noise due to instant phase changes, which might notably limit the accuracy.

In this article, we make the first investigation on the potential of multilevel phase analysis (MLPA), where different granularity phase analyses are combined together to improve the overall accuracy. The key observation is that the coarse-grained intervals belonging to the same phase usually consist of *stably distributed* fine-grained phases. Moreover, the phase of a coarse-grained interval can be accurately identified based on the fine-grained intervals at the beginning of its execution. Based on the observation, we design and implement an MLPA scheme. In such a scheme, a coarse-grained phase is first identified based on the fine-grained intervals at the beginning of its execution. The following fine-grained phases in it are then predicted based on the sequence of fine-grained phases in the coarse-grained phase. Experimental results show that such a scheme can notably improve the prediction accuracy. Using a Markov fine-grained phase predictor as the baseline, MLPA can improve prediction accuracy by 20%, 39%, and 29% for next phase, phase change, and phase length prediction for SPEC2000, respectively, yet incur only about 2% time overhead and 40% space overhead (about 360 bytes in total). To demonstrate the effectiveness of MLPA, we apply it to a dynamic cache reconfiguration system that dynamically adjusts the cache size to reduce the power consumption and access time of the data cache. Experimental results show that MLPA can further reduce the average cache size by 15% compared to the fine-grained scheme.

Moreover, for MLPA, we also observe that coarse-grained phases can better capture the overall program characteristics with fewer of phases and the last representative phase could be classified in a very early program position, leading to fewer execution internals being functionally simulated. Based on this observation, we also design a multilevel sampling simulation technique that combines both fine- and coarse-grained phase analysis for sampling simulation. Such a scheme uses fine-grained simulation points to represent only the selected coarse-grained simulation points instead of the entire program execution; thus, it could further reduce both the functional and detailed simulation time. Experimental results show that MLPA for sampling simulation can achieve a speedup in simulation time of about 8.3X with similar accuracy compared to 10M SimPoint.

Categories and Subject Descriptors: C.3 [**Computer Architecture**]: Phase Analysis; D.3 [**Programming Languages**]: Processors—*Runtime environments*

This work was funded by the China National Natural Science Foundation under grant 61370081 and the National 863 Program of China under grant 2012AA010901.
Authors' addresses: W. Zhang, J. Li, and Y. Li, Room 319, Software Building, 825 Zhangheng Road, Shanghai, China, 201203; email: whzhang.fd@gmail.com; H. Chen, Room 3402, Software Building, 800 Dongchuan Road, Shanghai, China, 200240.

DOI: http://dx.doi.org/10.1145/2629594

General Terms: Design, Dynamic Prediction, Performance

Additional Key Words and Phrases: Multilevel, phase, dynamic prediction, sampling, simulation

ACM Reference Format:
Weihua Zhang, Jiaxin Li, Yi Li, and Haibo Chen. 2015. Multilevel phase analysis. ACM Trans. Embedd. Comput. Syst. 14, 2, Article 31 (March 2015), 29 pages.
DOI: http://dx.doi.org/10.1145/2629594

1. INTRODUCTION

Programs generally have abundant execution intervals with repetitive behavior. Such intervals, which are consecutive portions of program execution, usually exhibit similar performance characteristics and resources requirements. The repetitive behavior in execution intervals exists in multiple levels or granularities: fine-grained ones such as the innermost loop body, coarse-grained ones such as the outermost loop body, and recursively or repetitively invoked functions. Accurately capturing such repetitive behavior could enable many optimizations, such as dynamic cache reconfiguration, power reduction, software debugging acceleration for multicore architectures, data prefetching, and sampling simulation.

Phase analysis has been established as a standard technique that characterizes the set of execution intervals with similar performance behavior into the same phase. To perform phase analysis, the execution of a program is first divided into nonoverlapping execution intervals. Then, intervals with similar behavior (e.g., similar IPC and/or cache miss rates) are classified into the same phase (i.e., phase classification). Since intervals in the same phase have similar performance characteristics and resource requirements, many optimizations can be applied through predicting the results of future phase classification (i.e., phase prediction) without expensive detailed modeling or analysis.

The prediction accuracy of phases, which is the proportion of correctly predicted intervals (i.e., the predicted phase ID of the interval equals the actual phase ID) in the total execution intervals, directly influences the effectiveness of dynamic systems using phase analysis. In phase analysis, the prediction accuracy is mainly affected by two key factors [Hind et al. 2003]: *phase granularity*, which is the size (e.g., instruction counts) of an execution interval composing a phase[1]; *phase metrics*, which are metrics identifying which behavior is repetitive, including control flows such as basic-block vectors (BBVs) [Sherwood et al. 2002; Perelman et al. 2003]; and/or data accesses such as memory reuse distance [Shen et al. 2004].

Although phase granularity has been one of the key parameters in phase analysis, there is little research on how it could affect the accuracy of phase analysis. Instead, most prior research usually partitions program execution into fine-grained intervals and applies a fine-grained strategy. Such a strategy, called *fine-grained phase analysis*, exploits most recent local history information to predict future phase behavior. The main advantage of such a scheme is the flexibility in timely adjustment of the optimization strategies on the fly according to dynamic program execution behavior. To guarantee prediction accuracy, they usually use a confidence counter and set a threshold for the counter to filter irrelevant intervals. However, although fine-grained phase analysis is flexible, it can only take account of recent local phase information and lacks *global phase history* information. Thus, the analysis could be frequently disturbed by temporary noise due to instant phase changes, which might notably limit the prediction accuracy.

[1]According to our measurement, the average size of iteration in outermost loops of all SPEC2000 benchmarks is about 2,000M (million) instructions. Hence, we consider an interval size in the range of 1M to 999M instructions as fine grained, whereas an interval size of more than 1,000M instructions as coarse grained.

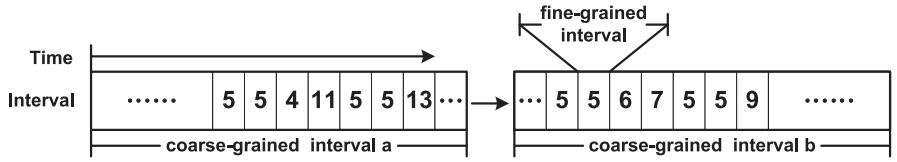


Fig. 1. An example phase sequence from facerec in SPEC2000.

To illustrate the limited prediction accuracy in a fine-grained prediction scheme, we use the intervals of an execution segment from the facerec benchmark in SPEC2000 as an example. As shown in Figure 1, interval *a* and interval *b* are two consecutive execution instances of an outermost loop, which belong to two different coarse-grained phases *A* and *B*, respectively. These two coarse-grained phases are formed by different sequences of fine-grained phases *4*, *5*, *6*, *7*, *9*, *11*, and *13*. If predicting the fine-grained phases in interval *b* using the execution history in interval *a*, the prediction accuracy could be lower than 50% due to the diverse phase sequences. This example shows that for two consecutive coarse-grained intervals of different phases, the prediction results tend to be inaccurate if the fine-grained phase behavior in one coarse-grained phase is predicted based on the execution history in the previous phase. However, simply using coarse-grained phase prediction will lose the flexibility of dynamically adjusting the optimization strategies in time during program execution.

In this article, we analyze the phase behavior of different granularities and observe some important characteristics in coarse-grainedphases: (1) for different fine-grained intervals belonging to the same coarse-grained phase, both the sequences and the distributions of these intervals are usually very similar and stable; (2) a coarse-grained phase could be accurately identified according to the execution of a few fine-grained intervals at the beginning of its execution. Based on the preceding observations, we design and implement a multilevel phase analysis (MLPA) scheme, which first characterizes some fine-grained phases and identifies a coarse-grained phase using some already executed fine-grained intervals. Then, the phases of the following fine-grained intervals to be executed can be predicted based on the sequences of fine-grained phases in a coarse-grained phase. Hence, MLPA could result in notable improvement in prediction accuracy, due to the consideration of global phase behavior, yet still retain the flexibility of fine-grained phase analysis.

To demonstrate the effectiveness of MLPA, we have implemented three phase prediction schemes: next phase (*NP*), phase change (*PC*), and phase length (*PL*). *NP* predicts the phase ID to which the next interval belongs, *PC* predicts which phase ID will occur after the next phase change, and *PL* predicts the length of next phase. Experimental results show such a framework can notably improve the prediction accuracy. When using a Markov fine-grained phase predictor as the baseline, MLPA can improve the prediction accuracy by 20%, 39%, and 29% for *NP*, *PC*, and *PL* prediction, respectively. It only incurs about 2% time overhead and 40% space overhead (about 360 bytes in total). To demonstrate the effectiveness of MLPA, we apply it to a dynamic cache reconfiguration system that dynamically adjusts the cache size to reduce the power consumption and access time of the data cache. Experimental results show that MLPA can further reduce the average cache size by 15% compared to the fine-grained scheme.

Moreover, for MLPA, we also observe that coarse-grained phases can better capture the overall program characteristics with fewer phases and the last representative phase could be classified in a very early program position, leading to fewer execution internals being functionally simulated. Based on this observation, we also design a multilevel sampling simulation technique that combines both fine- and coarse-grained phase analysis for sampling simulation. Such a scheme uses fine-grained simulation

points to represent only the selected coarse-grained simulation points instead of the entire program execution; thus, it could further reduce both the functional and detailed simulation time. Experimental results show that MLPA for sampling simulation can achieve a speedup in simulation time of about 8.3X with similar accuracy compared to 10M SimPoint.

In summary, this article makes the following contributions:

—The observation that the characteristics of coarse-grained phase can be used to improve dynamic prediction accuracy and accelerate sampling simulation.
—The MLPA scheme that combines both coarse- and fine-grained phase analysis, and its novel applications to dynamic cache reconfiguration and sampling-based simulation.
—The experimental evaluation that demonstrates the notable improvement in prediction accuracy and effectiveness of MLPA to dynamic cache reconfiguration and sampling-based simulation.

The remainder of this article is organized as follows. Section 2 describes the related work. Section 3 motivates our approach by illustrating the limitation of fine-grained phase analysis and analyzing the characteristics of coarse-grained phases. Section 4 describes our multilevel phase classification architecture. Section 5 presents our phase prediction algorithm. Section 6 evaluates the effectiveness of dynamic multilevel phase prediction and its applications. Section 7 presents a sampling simulation scheme based on MLPA. In Section 8, we conclude with a brief remark on future work.

2. RELATED WORK

In this section, we discuss prior work in two areas related to this article: phase analysis and applications of phase analysis. We summarize them in Table I according to their metrics, granularity, and usages.

2.1. Applications of Phase Analysis

Since many programs exhibit repetitive behavior over many different metrics, phase analysis has been widely used to identify repetitive program behavioral patterns for power reduction [Isci and Martonosi 2003, 2006; Huang et al. 2003], cache optimization [Shen et al. 2004; Lu et al. 2003], simulation acceleration [Sherwood et al. 2002; Perelman et al. 2003], and software debugging [Marino et al. 2009].

Power reduction. There have been many proposals for adaptive hardware mechanisms targeted at energy optimization. They dynamically adapt different aspects of the processors, including cache organization, issue width, voltage, and frequency. Isci and Martonosi [2003, 2006] used the power breakdowns to identify the power phase behavior and discussed how to use the control flow and the event counter for power behavior analysis. Huang et al. [2003] proposed an adaptive hardware method to partition program execution into fine-grained phases at the procedure level for power reduction.

Cache optimization. A research hotspot in a memory system is cache optimization based on the usage pattern of a running program. Balasubramonian et al. [2000] designed a phase-based system that can dynamically change cache configurations to improve performance and save power. Shen et al. [2004] proposed to collect information on data reuse distance for phase analysis and adaptive cache optimizations. Lu et al. [2003] proposed a runtime data cache prefetching scheme based on online phase analysis. Dhodapkar and Smith [2002] proposed a phase detection mechanism according to changes in instruction working sets [Denning and Schwartz 1972] and discussed their applications for cache reconfiguration.

Table I. A Classification and Comparison of Various Phase Analysis Algorithms

Algorithm	Metrics		Interval			Usage
	Inst	Data	Type	Grained	Phase Analysis	
Huang [Huang et al. 2003]	√	√	Procedure	Fine	Single Level	Power reduction
Georges [Georges et al. 2004]	√	√	Procedure	Fine/Coarse	Single Level	Phase analysis
Shen [Shen et al. 2004]		√	Loop	Fine	Single Level	Cache
Huffmire [Huffmire and Sherwood 2006].		√	Fixed length	Fine	Single Level	Phase analysis
Balasubramonian [Balasubramonian et al. 2000]		√	Fixed length	Fine	Single Level	Cache
Dhodapkar [Dhodapkar and Smith 2002]	√		Fixed length	Fine	Single Level	Cache reconfiguration
[Isci and Martonosi 2003, 2006]	√	√	Fixed length	Fine	Single Level	Power reduction
Duesterwald [Duesterwald et al. 2003]	√	√	Fixed length	Fine	Single Level	Simulation point
SimPoint [Sherwood et al. 2002]	√		Fixed length	Fine	Single Level	Simulation point
SPM [Lau et al. 2006]	√		Loop & Procedure	Fine	Single Level	Simulation point and Cache
HardwareBBV [Sherwood et al. 2003; Lau et al. 2005b]	√		Fixed length	Fine	Single Level	Phase analysis
Lu [Lu et al. 2003]	√		Fixed length	Fine	Single Level	Prefetching
Debugger tool [Marino et al. 2009]	√		Procedure	Fine	Single Level	Debugger tool
Cho [Cho and Li 2006]	√		Fixed-length	Fine	Single Level	Phase analysis
Lau [Lau et al. 2005a]	√		Fixed-length	Fine/Coarse	Single Level	Phase analysis
Our approach	√		Loop & Procedure	Both	Multilevel	Simulation point and Cache resizing

Simulation acceleration. Representative sampling has been one of most efficient techniques to accelerate the architecture simulation speed. As one of the most representative phase techniques, SimPoint [Sherwood et al. 2002; Perelman et al. 2003] uses BBVs as the metric for identifying phases and simulation point selection. The software phase marker (SPM) [Lau et al. 2006] method is also a BBV-based technique. Instead of using fixed-length intervals, it selects fine-grained phases according to loop or procedure boundaries for simulation point selection and cache reconfiguration. With the popularity of parallel applications, how to select simulation points for parallel applications is discussed in Perelman et al. [2006], Van Biesbrouck et al. [2004], and Genbrugge et al. [2010] raises the level of abstraction to do interval simulation in multicore architectural simulation, and the method of estimating programs' multicore

performance based on single-core simulation is discussed in Van Craeynest and Eeckhout [2011].

Software debugging. Dynamic software debugging tools for multithreaded programs have been widely used due to their accuracy and immense help to programmers. However, a significant impediment to their adoption is their runtime overhead. Marino et al. [2009] applied function-level phase analysis to reduce the runtime overhead of those software debugging tools.

2.2. Optimizing Phase Analysis

Due to the importance of phase analysis, there has been a considerable amount of research aimed at optimizing the accuracy of phase analysis. Whereas most previous work uses a fine-grained phase prediction scheme, this work is the first to use a multilevel scheme that combines both coarse- and fine-grained phase analysis, resulting in extremely good prediction accuracy. In the following, we briefly discuss the previous literature in optimizing phase analysis.

Lau et al. [2005a] are the first to discuss the existence of different granularities of phase behavior (i.e., coarse- and fine grained) in programs. However, they did not discuss the relationship between different phase granularities and how to exploit those characteristics to improve prediction accuracy of phase prediction. Hence, in their later work (e.g., SPM [Lau et al. 2006]), they still applied fine-grained strategies. In contrast, this work is the first to make some key observations on the relationship between a coarse-grained phase and its fine-grained phases (i.e., the combination and the distribution of fine-grained phases in a coarse-grained phase), and describes how to apply a multilevel scheme for phase analysis and prediction.

Sherwood et al. [2003] and Lau et al. [2005b] showed that BBV-based phase analysis is efficient in dynamic systems. Although we use the same signature (BBV) as that in prior systems, the prediction strategy and transition phase identification of MPLA are new and are the major contributions of this article, which lead to the notable improvement over prior work [Sherwood et al. 2003; Lau et al. 2005b]. Dhodapkar and Smith [2003] made a performance comparison with instruction-execution–related metrics and showed that BBV performs better than other metrics. Lau et al. [2004] showed that using loop frequency vectors as a metric performed comparably with BBV in accuracy and could further yield fewer distinct phases. Huffmire and Sherwood [2006] used hashed memory accesses as the metric to partition program execution into fine-grained phases. Georges et al. [2004] used an offline implementation of a method-level phase detection algorithm to characterize phase behavior of Java programs. Duesterwald et al. [2003] used hardware counters for phase prediction to find phase behavior. Cho and Li [2006] used a wavelet method to characterize phase complexity and the changes of program behavior.

2.3. Statistical Sampling

Statistical sampling is another common sampling simulation technique. Unlike representative sampling, statistical sampling selects simulation points according to statistical theories instead of analyzing the repetitive behaviors in programs. As shown in Yi et al. [2005], the representative sampling represented by SimPoint [Sherwood et al. 2002] achieves a better trade-off than statistical sampling represented by SMARTS [Wunderlich et al. 2003] in the sense of reaching satisfactory accuracy with less simulation time. Therefore, we compare our approach only with SimPoint in the experiments.

3. MOTIVATION OF MULTILEVEL PHASE ANALYSIS

Phase analysis has been established as a standard technique that characterizes the set of execution intervals with similar performance behavior into the same phase.

Table II. Basic Configuration Used Throughout This Article

Base Configuration (Config A)	
Parameter	*Value*
Out-of-order issue	8-way decode, issue, commit width
ROB/LSQ entries	128/64
Registers	32 integer, 32 floating point
Functional units	8-integer ALU, 4-load/store units, 2-FP adders
	2-integer MULT/DIV, 2-FP MULT/DIV
Instruction cache	8k 2-way associative, 32 byte blocks, 1 cycle latency
Data cache	16k 4-way associative, 32 byte blocks, 2 cycle latency
Unified L2 cache	1Meg 4-way associative, 32 byte blocks, 20 cycle latency
Branch predictor	Combined, 8K BHT entries
Memory latency	150, 10 cycle access (first, following)

Due to its effectiveness, phase analysis has been widely used for many optimizations. The dynamic optimizations include dynamic cache reconfiguration, power reduction, software debugging acceleration for multicore architectures, and data prefetching. The major application of static phase analysis is sampling simulation.

Although phase granularity has been one of the key parameters in phase analysis, there is little research on how it could affect the accuracy of phase analysis. Instead, most prior research usually partitions program execution into fine-grained intervals and applies a fine-grained strategy. This section first analyzes the motivation for dynamic MLPA. Then we will present the motivation for MLPA for sampling simulation.

3.1. Evaluation Methodology

We performed our analysis using several SPEC2000 programs with reference inputs. The programs include gzip, mgrid, gcc, equake, facerec, lucas, bzip2, ammp, and mcf. The major reasons that we chose these programs are as follows. First, they were widely used in prior similar research, such as [Sherwood et al. 2003; Lau et al. 2005b, 2006; Shen et al. 2004]. Moreover, Aashish et al. [Phansalkar et al. 2005] demonstrated that the programs in SPEC2000 can be clustered into eight subsets based on similarity analysis. Our selected programs primarily cover those subsets, which can effectively represent the entire suite. Finally, as analyzed in Nair and Joh [2008], the programs in SPEC2006 have similar phase behavior and phase distribution to those in SPEC2000. Hence, to make a comparable study with prior literature, we choose SPEC2000 instead of SPEC2006 in our study. The statistics of phase behavior in these workloads were measured using SimpleScalar. The base simulation configuration is detailed in Table II, which is the same as that in Lau et al. [2005b], Sherwood et al. [2003, 2002], Perelman et al. [2003], and Lau et al. [2006]. We use arithmetic mean to compute the average results for the following accuracy evaluations. For speedup, the average speedup is the total simulation time of our approach (MLPA) divided by that of SimPoint method. For accuracy results, we use arithmetic mean. The major reason behind it is that it is difficult to assign a weight to each benchmark, which makes it impossible to use the harmonic mean. For the arithmetic mean, it is not reasonable since the speedup values for some benchmarks are very large (more than 4,000).

Fine-grained phase prediction schemes to study. In fine-grained phase prediction, the execution history is usually composed of the most recent phase information, including past phase IDs and phase lengths. Based on history information used for prediction, there are two widely used prediction methods: last value and Markov. As analyzed by Lau et al. [2005b], last value prediction, which simply predicts the phase of the next interval as the same phase of the last executed interval, incurs little hardware

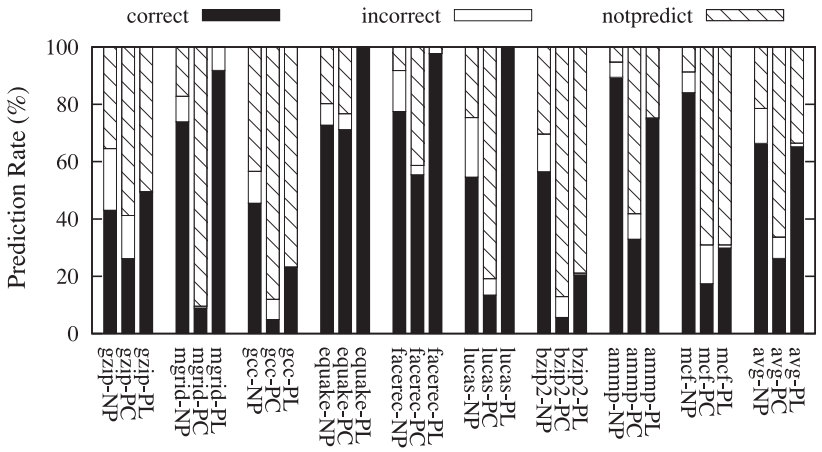


Fig. 2. The prediction accuracy of a fine-grained prediction scheme using a Markov model for three phase prediction types: *NP*, *PC*, and *PL*.

or software overhead. The basic idea of the Markov model is that the next state of a system is only related to the last set of states. Hence, a Markov scheme of order k predicts the next phase based on the phase behavior of k previous phases. The Markov model is a classical predictor, which is easily implemented in hardware or software and has been widely used to predict various events, such as branch prediction [Chen et al. 1996] and prefetching [Joseph and Grunwald 1997].

Since last value strategy cannot be used to predict phase change and phase length, we only use the Markov model as the baseline for the evaluation of those two attributes. A Markov predictor with order 1 (Markov-1) and order 2 (Markov-2) [Lau et al. 2005b; Sherwood et al. 2003] are two of the most widely used models in prior research, and Markov-2 is a more accurate scheme. We thus focus on Markov-2 in this article and use Markov-2 (Markov) with run length information [Sherwood et al. 2003].

3.2. Motivation for Dynamic Multilevel Phase Analysis

In this subsection, we will first analyze the limitations of fine-grained phase analysis for dynamic behavior prediction. Then, we will give out the observation and motivation for MLPA.

3.2.1. Limitations on Dynamic Behavior Prediction. To illustrate the possible inaccuracy in a fine-grained prediction scheme, we evaluated the Markov model using the evaluation methodology described earlier. Figure 2 shows the prediction results of three prediction types: *NP*, which predicts the phase ID of the next interval; *PC*, which predicts the phase ID that will occur after the next phase change; and *PL*, which predicts the length of next phase. The prediction accuracy is defined as the number of correctly predicted intervals divided by the number of totally executed intervals. When predicting the next phase length, it is difficult to predict the exact length (i.e., the number of intervals in the phase). Fortunately, under many conditions, it is enough to know the approximate length of the next phase—that is, whether it is short or long. Therefore, in this article, we also use the classification method by Lau et al. [2005b]. The phase lengths are grouped into four sets: 1–15, 16–127, 128–1023, and intervals longer than 1024, which roughly correspond to the phase lengths of 10–100M instructions, 100M–1B instructions, 1B–10B instructions, and more than 10B instructions, respectively.

As shown in the figure, the average prediction accuracy is 66%, 26%, and 65%, respectively. The major reason behind the low prediction accuracy is that a program

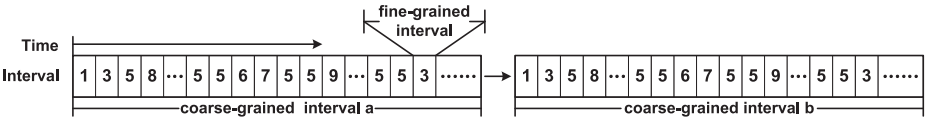


Fig. 3. An example phase sequence of gzip from SPEC2000.

usually contains multilevel repetitive execution behavior, such as nested loops and recursive functions, which leads not only to fine-grained phase behavior but also to coarse-grained phase behavior being exhibited during program execution. Thus, not considering the multilevel execution behavior would lose many prediction opportunities. Consequently, 21%, 66%, and 33% of phases are not predicted for *NP*, *PC*, and *PL*, respectively.

After a detailed analysis of the phase distribution of typical program execution, we found that different coarse-grained intervals (e.g., outermost loop) usually consist of different sequences of fine-grained intervals. Hence, the prediction tends to be inaccurate if using the fine-grained phase information in one coarse-grained phase to predict the fine-grained phase behavior in another coarse-grained phase, which has been shown in Figure 1.

Even if two consecutive coarse-grained intervals belong to the same coarse-grained phase, it might not be accurate to use the fine-grained phase information at the end of the previous interval to predict the fine-grained phase behavior at the beginning of the next interval. Figure 3 illustrates such a situation by showing the phase behavior and execution sequence of the gzip benchmark from SPEC2000. As shown in the figure, *1*, *3*, *5*, *6*, *7*, *8*, and *9* are fine-grained phase IDs, and both coarse-grained interval *a* and coarse-grained interval *b* belong to the same coarse-grained phase *A*. Since the fine-grained phase behavior at the end of coarse-grained interval *b* is dissimilar with that at the beginning of coarse-grained interval *a*, both last value prediction and the Markov prediction achieve poor prediction accuracy (lower than 45% based on our evaluation results).

3.2.2. Observation and Motivation for Dynamic Behavior Prediction. To gain insight into possible solutions to increase the accuracy of phase prediction, we studied the distribution of phases in SPEC2000 and found that the sequences of fine-grained phases are similar and stable for different coarse-grained intervals that correspond to the same phase. According to our measurement of SPEC2000, the identical sequences of fine-grained phases are more than 80% for different coarse-grained intervals in the same phase.

Observation of phase classification. Based on the characteristics of phase distribution in SPEC2000, we found that the phase of a coarse-grained interval can be accurately identified based on the executed fine-grained intervals at the beginning of its execution. Hence, instead of identifying a coarse-grained phase by executing all fine-grained intervals within the coarse-grained interval, we can use a few fine-grained intervals at the beginning of a coarse-grained interval to identify the coarse-grained phase to which it belongs. Figure 4 shows the accuracy of coarse-grained phase identification based on a different number of fine-grained intervals at the front of them. As the results show, when using five fine-grained (10M) intervals to predict the coarse-grained phase, we can correctly identify more than 95% of the coarse-grained phases, which is close to the results for 10 intervals of 10M. Hence, the result would be quite accurate if using five fine-grained intervals for phase identification. By contrast, the average number of fine-grained intervals is about 200 in a coarse-grained phase (as described in footnote 1).

Observation of phase prediction. The pervasive existence and stable distribution of coarse-grained phases in many programs provide opportunities to exploit the coarse-grained phase behavior of phase prediction. Hence, not only the fine-grained phase

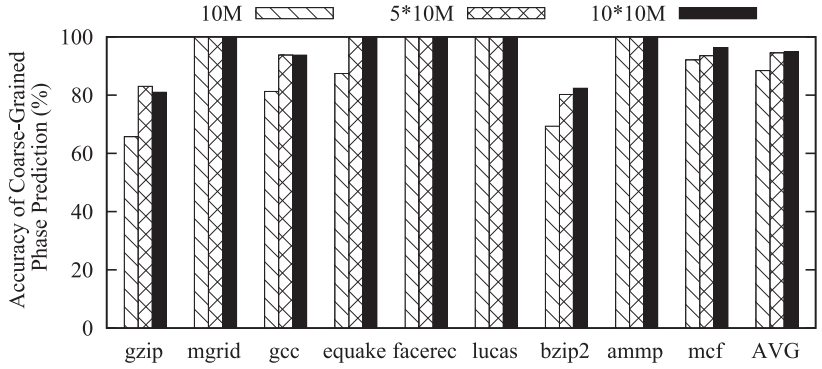


Fig. 4. The prediction accuracy of coarse-grained phase based on the number of fine-grained intervals at the beginning of their execution.

behavior but also the coarse-grained phase information could be leveraged to predict both the fine- and coarse-grained phases. Specifically, we can first identify the (coarse-grained) phase to which the current coarse-grained interval belongs based on the beginning phase sequence of already executed fine-grained intervals. Afterward, the rest of the fine-grained phase sequence can be predicted based on the recorded phase sequence in the coarse-grained phase that corresponds to the current interval.

Figure 3 also shows an example of using coarse-grained phases to predict fine-grained phases. As shown in the figure, a and b are two coarse-grained intervals that will be classified into the same coarse-grained phase A. Interval a is executed before interval b. Once the execution of interval a is completed, the sequence of fine-grained phases in it will be recorded. When interval b is being executed, it will be identified as phase A after five fine-grained intervals at the beginning of its execution are finished, because the beginning five fine-grained phases match those of interval a. As interval b is identified as phase A, the remaining sequence of fine-grained phases can be accurately predicted according to the recorded phase sequence in phase A.

3.3. Motivation for Multilevel Phase Analysis for Sampling Simulation

Besides applied in dynamic optimizations, phase analysis can also be used in static analysis. One of the most representative applications is sampling simulation. In this subsection, we will first analyze the limitations of fine-grained phase analysis for sampling simulation. Then we will give out the observation and motivation of MLPA for sampling simulation.

3.3.1. Limitations on Sampling Simulation. To reduce *detailed* simulation time, prior sampling techniques, such as SimPoint [Sherwood et al. 2002] and the SPM method [Lau et al. 2006], tend to choose fine-grained phases and to constrain the maximum interval size. However, when a program is spilt into finer-grained intervals, more sensitive changes in program behavior will be exposed, and more phases would be identified even if their overall proportion in the entire program were very low (i.e., not very important), as the example introduced in Section 3.3.2. Consequently, more simulation points will be selected. Some of them will usually be located close to the end of the program's execution. Hence, the simulation of program execution will need to be extended to a substantially larger portion of the program execution to cover the end phases. In such cases, even though using finer-grained phases could reduce the detailed simulation time of each simulation point, the total amount of simulated instructions could be

Table III. Coarse-Grained Phase Number in SPEC2000

Phase Number	Benchmark Number
1	9
2	7
3	1
Larger than 3	3

substantially increased, which consequently increases the total simulation time (both functional and detailed).

3.3.2. Observation and Motivation for Sampling Simulation. To gain insight into possible solutions to further improve the efficiency of the sampling simulation method, we studied the behavior of coarse-grained phases in SPEC2000 and found some interesting characteristics, which could be exploited to further improve the efficiency of sampling simulation.

Reducing functional simulation time. Based on the characteristics of coarse-grained phase distribution in SPEC2000, we found that the number of coarse-grained phases is very small. The detailed data are shown in Table III. For SPEC2000, the average coarse-grained phase number is three, and only three benchmarks' coarse-grained phase numbers are larger than three (four for gzip, six for equake, and five for fma3d). Furthermore, the position[2] of last coarse-grained simulation points is very small (i.e., early). For example, the average position for SPEC2000 is about 17%, and only three benchmarks are larger than 30% (86% for gcc, 47% for art, and 36% for bzip2). Based on this observation, we can select coarse-grained simulation points to optimize the functional simulation time in a sampling simulation approach.

Here we use two benchmarks in SPEC2000, lucas and facerec, as examples. We respectively collect the BBVs of each fixed-length interval of 10M instructions and that of our coarse-grained approach. Since BBVs are a kind of multidimensional data, we use principal component analysis (PCA) [Johnson and Wichern 2002] to extract their first principal component. The PCA results have decreasing variance, with the first principal component containing the most information and the last one containing the least information. To illustrate the problem more clearly, we retain the top principal component containing more than 90% information. Those PCA values are shown on the y-axis in Figure 5. The interval numbers of each program are numbered according to their execution order shown on the x-axis. As shown in the figure, the curves of the fine-grained method are very chaotic, with violent changes. These lead to more phases being identified and more simulation points being selected close to the end of program execution (shown as the check marks in Figures 5(b), 5(c), 5(d), and 5(e)). In contrast, the curves of the coarse-grained approach are very smooth. Thus, far fewer simulation points at very early stages of program execution are selected (shown in Figures 5(a) and 5(d), and determining the number of coarse phases will be introduced in Section 7). As a result, a very large portion of program execution needs to not be functionally simulated, and the overall simulation time could be reduced.

Reducing detailed simulation time. Furthermore, as shown in Figure 6, after selecting the coarse-grained simulation points, we could apply a fine-grained sampling method again to those coarse-grained simulation points. Since those fine-grained simulation points are only used to represent the selected coarse-grained simulation points instead of the entire program execution, less fine-grained simulation points will be required, which can further optimize the detailed simulation time compared to pure

[2]We define the position of an interval in a program to be the instruction number before its last instruction dividing the total instruction number in the program.

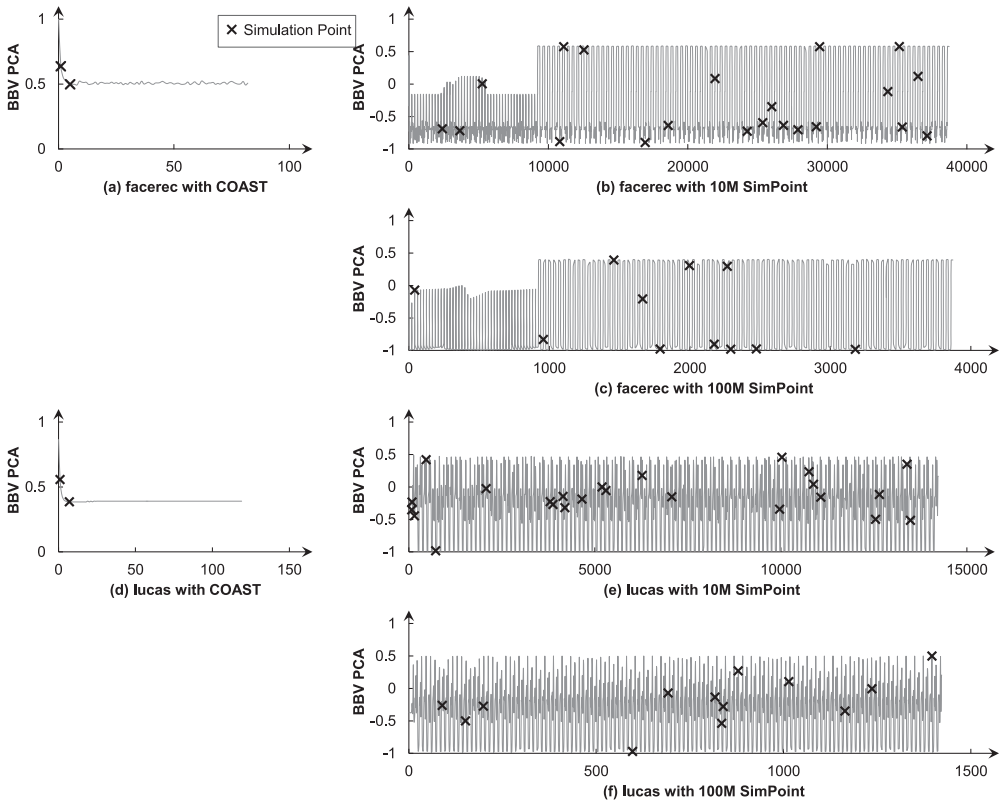


Fig. 5. This example illustrates how different granularities influence the selection of simulation points. The x-axis is the interval number (numbered according to its execution order), and the y-axis is the PCA eigenvalue of BBV in each interval. The check marks are the positions of the selected simulation points.

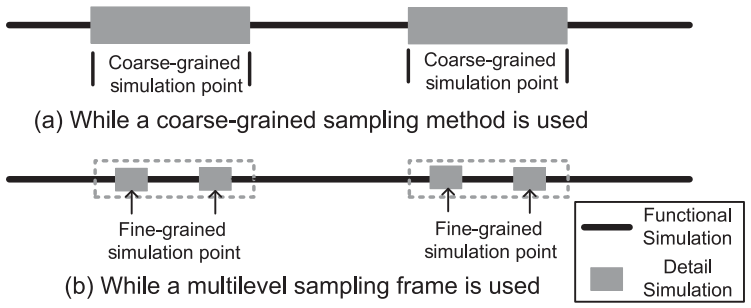


Fig. 6. This example illustrates how the MLPA framework works.

fine-grained methods. Therefore, in a representative sampling simulation method, such a multilevel method could have the advantages of both coarse- and fine-grained approaches.

Since sampling methods tend to select the most representative parts to represent the entire execution of a program, it can guarantee the simulation accuracy to a certain level. Therefore, sampling twice in our multilevel sampling approach will not lead to

more deviation. Our evaluation results confirm that the accuracy of our multilevel scheme is comparable to a fine-grained scheme.

4. MULTILEVEL PHASE CLASSIFICATION

In this section, we introduce the basic phase classification algorithm, our multilevel phase classification architecture, and our transition phase identification algorithm.

4.1. Basic Steps in Phase Classification

In MLPA, we use 10 million (10M) instructions as the length for a fine-grained interval. The reason is that 10M is roughly the time between OS context switches and is small enough to adjust different optimizing strategies. In fact, such an interval length is widely used in prior research, which makes the comparison with other work feasible. Furthermore, we use the outermost loop boundaries to form the coarse-grained intervals. The basic steps in phase classification can be summarized as follows.

Interval partition. The first step is to partition the execution of a program into multiple intervals. We identify a coarse-grained interval according to the outermost loop or frequently invoked functions. The boundary information of coarse-grained intervals is collected with the dynamic method similar to prior work [Lau et al. 2006; Huang et al. 2003]. In each coarse-grained interval, we then classify each 10M instruction stream into a fine-grained interval. For the fine-grained phase analysis, such as last value or Markov, we also use the same partition method.

Signature collection. A signature is a metric to represent the characteristics of program execution. When the intervals are executed, some metrics, such as memory access or control flow information, are collected in this step. Afterward, these metrics form a signature to represent a fine-grained interval. Finally, the signatures of fine-grained intervals in a coarse-grained interval will be summarized as the signature of the coarse-grained interval. In this work, we use BBV [Sherwood et al. 2002] as the signature to classify phases because it provides a higher sensitivity and can produce more stable phases as analyzed in Dhodapkar and Smith [2003].

Phase classification. To perform phase classification, the signature will be compared with past signatures through computing Manhattan distances between them to determine whether the current interval belongs to an existing phase or is a new one. If a match occurs, the phase ID for the matched signature table entry is returned. Otherwise, the signature is marked as a new phase and will be inserted into the signature table. The detailed algorithm will be presented in next section.

4.2. Multilevel Phase Classification

Instead of classifying phases by only considering fine-grained intervals, we combine fine- and coarse-grained phase classification together, which forms multilevel phase classification. Fine-grained phases and coarse-grained phases are presented, respectively, and they are combined to identify transition phases and perform phase prediction as described in the next section. Our phase classification architecture is shown in Figure 7, which consists of two parts: coarse-grained phase classification and fine-grained phase classification.

The lower portion of the figure is composed of a BBV queue with two vectors, which records the program counter of every committed branch and the instruction number committed between two consecutive branches, respectively. In the upper portion of the figure, there are two arrays with N saturating counters (accumulators) holding signatures for both the current fine-grained interval and the coarse-grained interval, respectively. Each program counter in the BBV queue will be hashed into an item of

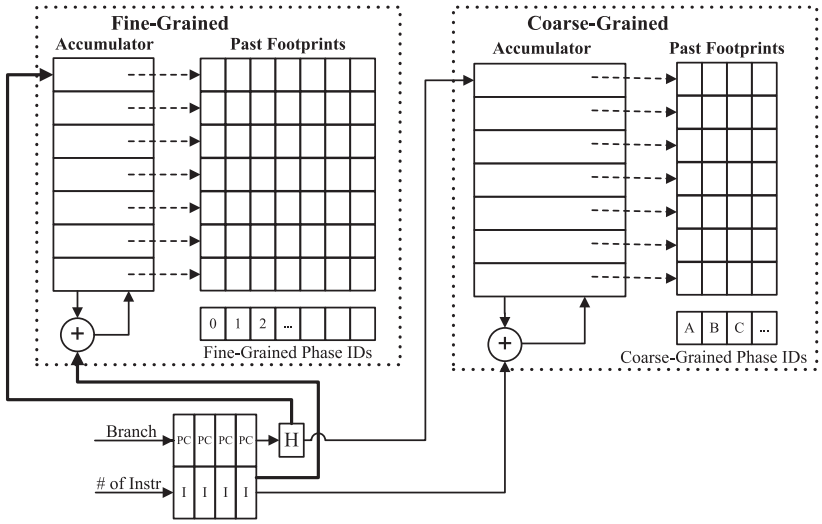


Fig. 7. The architecture of multilevel phase classification. PC, program counter; I, instruction number.

each accumulator, with the corresponding counter being incremented by I. The counter in the accumulator will be used to track the proportion of the program executed.

The past footprint arrays in the upper portion of Figure 7 are used to store the signatures for existing phases. The size of both arrays is configured as 32 entries, which is the same as that in Lau et al. [2005b]. For the fine-grained signature table, each item stores the signature of the entire fine-grained interval. For the coarse-grained signature table, the signatures of the beginning fine-grained intervals of a coarse-grained phase are stored for later phase identification. As shown in Figure 4, five 10M intervals (50M) at the front of a coarse-grained interval can accurately represent the corresponding coarse-grained phase. Therefore, we use the signature of 50M at the front of a coarse-grained interval as its signature, and the signature is compared after 50M instructions are executed.

To determine whether the current signature is similar to past phases, we search the signature table. If there is an entry in the table within a similarity distance (i.e., Manhattan distance [Sherwood et al. 2002]) to the current signature, the interval of the current signature is classified as an existing phase. In this work, the threshold of similarity distance is 25%, which is also the same as that in Lau et al. [2005b]. If a match occurs, the value in the signature table will be replaced with the current signature. Otherwise, the current signature will be inserted into the table as the representative of a new phase. Multiple signatures in the signature table may satisfy the similarity threshold. In this case, we choose the phase whose signature is most similar to the current signature.

To decide the number of entries for coarse-grained phase classification, we measured the coarse-grained phase data. The data are shown in Table IV. *CPhase Num* is the number of coarse-grained phases in each benchmark, and *CInterval Num* is the average interval number belonging to each coarse-grained phase. According to our tests, the average phase number of coarse-grained phases in SPEC2000 is three, and the largest phase number is eight. Since the phase number of most benchmarks is not larger than three, we set three entries to store past signatures for coarse-grained classification. When the table is full, a First-In-First-Out strategy is applied for replacement.

Table IV. Statistics of Coarse-Grained Phase

Benchmark	CPhase Num	CInterval Num
gzip	3	47
mgrid	1	25
gcc	3	9
equake	8	19
facerec	2	63
lucas	2	61
bzip2	3	43
ammp	2	86
mcf	5	9
AVG	3	40

4.3. Transition Phase

Generally, consecutive phases, where the same phase appears consecutively for a relatively long period, are more suitable for optimizations. However, there might be some transition phases [Lau et al. 2005b] between two consecutive phases. Since such transition phases are usually short and may happen rarely, the optimizing strategies should not be adjusted when a transition phase occurs. Therefore, it is necessary to identify transition phases, which can improve prediction accuracy and reduce pressure on signature table.

In a prior fine-grained phase classification scheme [Lau et al. 2005b], the transition phases are identified by checking whether the frequency of a phase has been larger than a threshold (e.g., eight). However, such an approach has difficulties in some conditions. For example, consider phase sequence a, a, a, a, b, a, a, a, a, and b, where a and b are two phase IDs. Phase a will be identified as a transition phase if the threshold is larger than four. This leads to the waste of some opportunities for optimizing for phase a. Even worse, it is difficult to choose a uniform threshold for different benchmarks.

In our multilevel phase classification scheme, we identify transition phases in each coarse-grained interval by checking if the fine-grained phase appears with low frequency in it. If so, the fine-grained phase is not important and can be considered to be a transition phase. Therefore, transition phases can be identified by checking the proportion of a newly appeared fine-grained phase in the prior coarse-grained interval. Such a design can lead to more accurate results and more opportunities for optimization. As analyzed in Lau et al. [2005b], the transition phases on average accounts for about 6% of program execution. Therefore, we set the threshold for the proportion of fine-grained phases at 6%. If the proportion of a newly appeared fine-grained phase is lower than 6% in the prior coarse-grained interval, it will be identified as a transition phase.

5. MULTILEVEL PHASE PREDICTION

This section describes how to use MLPA to predict future phase behavior, including *NP*, *PC*, and *PL*.

As shown in Section 3, coarse-grained phases are stable in the composition and distribution of fine-grained phases. Further, a coarse-grained interval can be accurately identified according to the fine-grained intervals at the beginning of its execution. Therefore, based on such features of coarse-grained phases, we design a multilevel scheme to improve the accuracy of phase prediction. The basic idea is to first identify a coarse-grained phase based on the sequences of its beginning fine-grained intervals. Then, the remaining fine-grained intervals will be predicted based on the history information in the corresponding coarse-grained phase.

Table V. A Comparison of Space Requirements of Two Methods for the Largest Coarse-Grained Intervals in Different Benchmarks

Benchmark	Uncompressed (Byte)	Compressed (Byte)
gzip	108	95
mgrid	588	132
gcc	700	140
equake	117	43
facerec	230	117
lucas	151	52
bzip2	194	75
ammp	702	50
mcf	1,927	117
AVG	524	91

5.1. Saving History Information

In multilevel phase prediction, the sequence of fine-grained phases in the just-finished coarse-grained interval will be saved for future uses. There are two methods to save this information. The first one is to sequentially save the corresponding phase ID of each fine-grained interval in a coarse-grained interval in an array. The second method compresses the space to store consecutive phases by using a triple <*id*, *beginpos*, *length*>. In the triple, *id* is the phase ID of a fine-grained interval, *beginpos* is the beginning position of this phase, and *length* is the phase length that fine-grained intervals repeat. Since a stable fine-grained phase generally lasts for a long execution time, the second method is more space saving. Table V is a comparison of the space overhead of these two methods for the longest coarse-grained intervals in different benchmarks. As the data show, the second method can significantly reduce the space overhead, and such a scheme saves around 83% of space compared to saving the phase ID sequentially in an array. Since the largest space requirement is 140 bytes, we use it as the length of history table. In the phase classification architecture, each coarse-grained phase item corresponds to such a history table item. While the phase signature is updated, this table is also updated.

5.2. Multilevel Phase Prediction

Before a coarse-grained phase is identified, the fine-grained intervals at the beginning of its execution are predicted based on the fine-grained phase prediction. After there are enough executed fine-grained intervals (i.e., five fine-grained intervals in this article), the coarse-grained phase is identified. If this coarse-grained phase is a new one, the following fine-grained intervals in it will also be predicted based on the fine-grained phase prediction. Otherwise, the following fine-grained intervals are predicted based on the history information in this coarse-grained phase. Since the fine-grained phase sequence of two coarse-grained instances classified into one phase will not always be totally the same, the position of a fine-grained phase in the history information must be located. After the execution of the current fine-grained interval is finished, its phase ID is identified. Based on its position in the sequence (interval number) in the coarse-grained interval, its phase ID, and the phase ID adjacent to it, we search the history table for the history information of the corresponding coarse-grained phase. Specifically, the corresponding position in the current interval and history table are first searched. If the phase ID matches, then the history information is used for the subsequent prediction. Otherwise, several intervals adjacent to the current position are compared to find a match. In this work, we use the first matched interval as the final position. Figure 8 is the comparison of different search distances where *d-n* is the

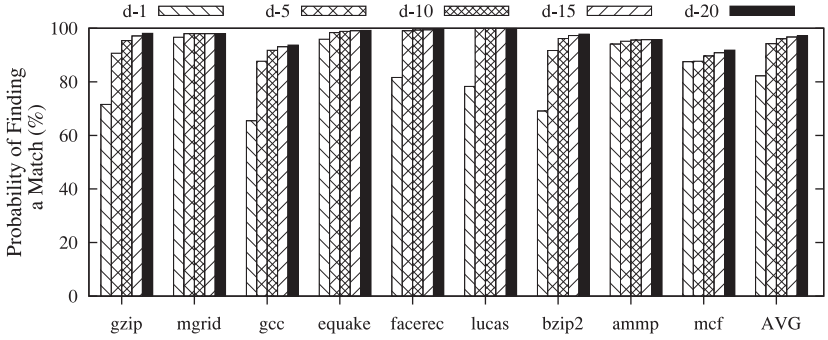


Fig. 8. The probability of finding a match with various search distances (1, 5, 10, 15, and 20).

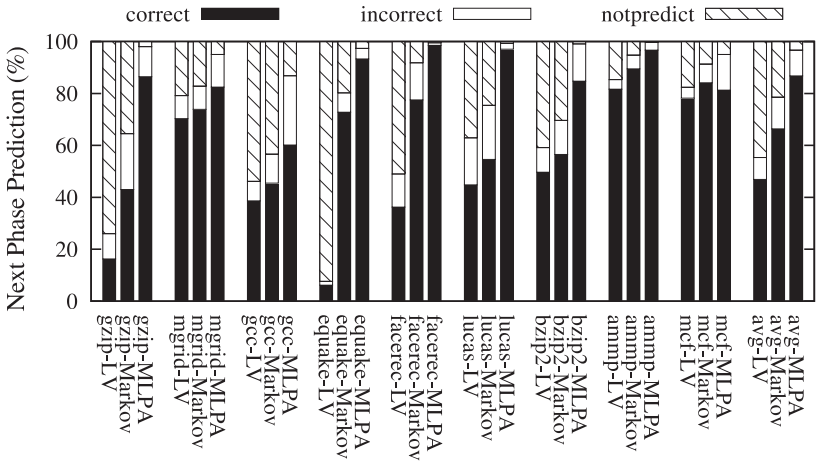


Fig. 9. Accuracy of *NP* prediction.

search distances, with n. As the data show, a distance of five can achieve a good trade-off between time and accuracy. Therefore, we use it as the parameter for searching.

6. EFFECTIVENESS OF MULTILEVEL PHASE PREDICTION

To demonstrate the effectiveness of our MLPA, we evaluated the improvement of prediction accuracy in *NP* prediction, *PC* prediction, *PL* prediction, and the overall overhead in space and time. We also described an example application of MLPA—dynamic cache reconfiguration—to demonstrate the effectiveness of our approaches.

6.1. Effectiveness of Multilevel Phase Analysis

Next phase prediction. *NP* prediction predicts the phase ID to which the next interval belongs, which is done for every execution interval. The results are shown in Figure 9. As the results show, MLPA gets more accurate prediction results for most benchmarks and achieves 20% accuracy improvement on average compared to the fine-grained method. The major reason is that the combination and distribution of fine-grained phases in different intervals of a coarse-grained phase is stable in most cases. Moreover, a coarse-grained phase can be accurately identified according to the execution of a few fine-grained intervals at the beginning of its execution. Therefore, compared to the fine-grained methods, more accurate history information is available, which makes the

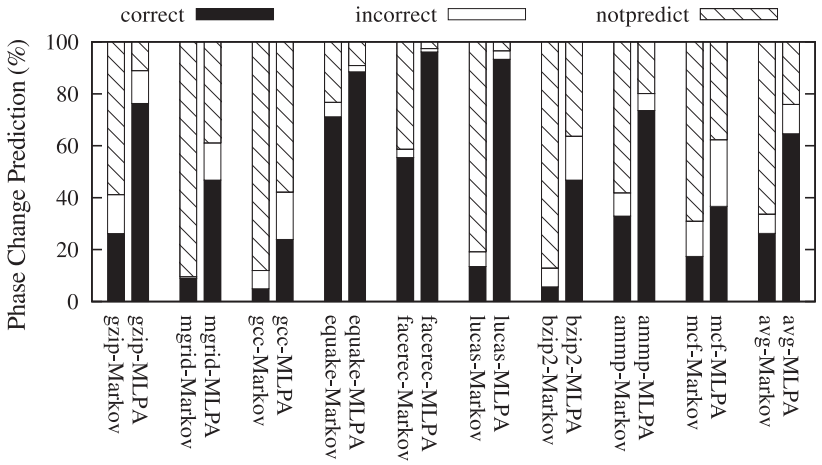


Fig. 10. Accuracy of *PC* prediction.

prediction of the next phase more accurate. For mcf, the prediction accuracy degrades a little for MLPA. The reasons are twofold. First, the regular fine-grained phases lead to a very accurate prediction result for fine-grained Markov-2 prediction. Second, the number of coarse-grained intervals is small (about 45), and the coarse-grained phase behavior is not very stable as in other benchmarks. In such a case, most coarse-grained intervals are identified as different phases. Therefore, the related fine-grained phases are essentially predicted using the fine-grained strategies.

Phase change prediction. *PC* prediction predicts the outcome of the next phase change—that is, predicting which phase ID will occur after the next phase change. When such a change occurs, the optimizing strategies can be adjusted accordingly. Figure 10 is the comparison of the results of multilevel phase prediction and those of Markov-2. As the results show, multilevel phase prediction achieves about 39% accuracy improvement over fine-grained Markov-2, resulting in 65% accuracy on average.

Phase length prediction. Besides predicting the next phase ID, it is also useful to know how long the next phase will repeat. Because the execution of programs generally consists of long-term stable periods and short transition periods, knowing the length of the next phase will avoid some unnecessarily expensive reconfigurations for phases that will not execute long enough. The length of the next phase will be predicted on the completion of the current phase.

Figure 11 shows the results of comparing multilevel phase prediction with fine-grained phase prediction. As the results show, MLPA achieves 94% prediction accuracy, which has a 29% improvement over fine-grained phase prediction (65%).

Overhead evaluation. MLPA must analyze and predict phase behavior based on the information of both fine- and coarse-grained phases. Therefore, there will be additional space and time overhead, and we will evaluate the associated time overhead and space overhead using our MLPA.

Our MLPA can be implemented in hardware, in software, and in a hybrid manner. For hardware implementation, the coarse-grained phase analysis can be implemented in parallel with that of fine-grained phase analysis. Therefore, it will not involve any additional time overhead. For software implementation, we measure the time overhead in our software implementation. Based on the measurement, the average software time overhead compared to the Markov fine-grained method is about 2%, and the largest one is less than 3%.

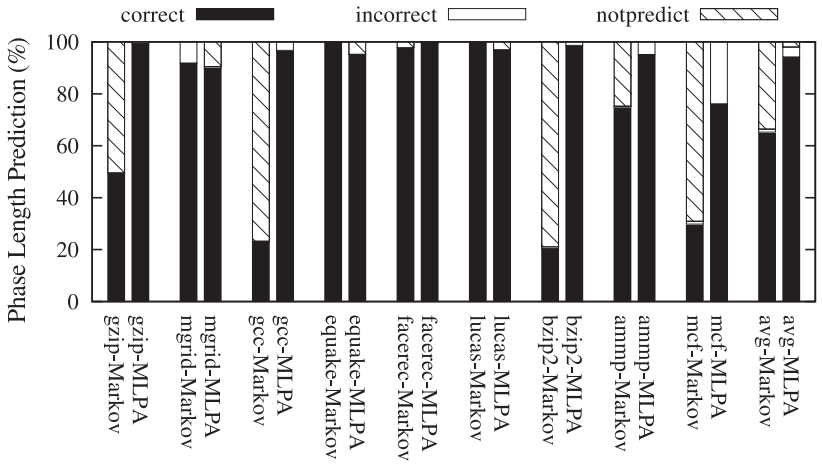


Fig. 11. Accuracy of *PL* prediction.

The space overhead is fixed after the phase classification, and the prediction algorithm is designed to support both software implementation and a hardware design. In our current design, the space overhead is about 40%. Since the total space requirement for fine-grained phase prediction is about 900 bytes, such a space overhead is not a problem for current hardware and software platforms.

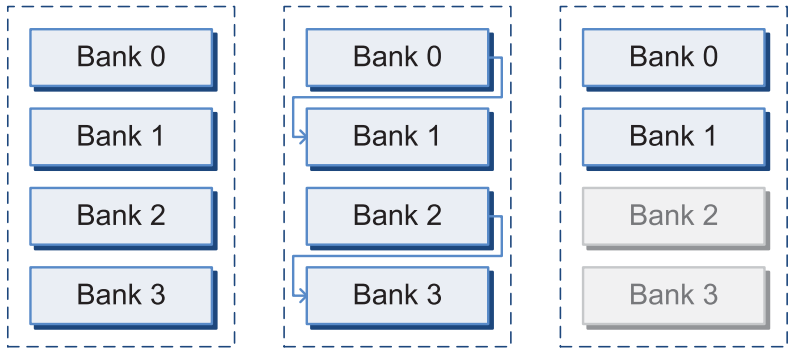
6.2. Cache Reconfiguration

Energy consumption has been a major concern in embedded computing systems, which usually are powered by battery.

The ever-increasing gap between the processing unit and memory results in the prevalence of on-chip caches. On-chip cache hierarchy has become default parts in embedded processors and is responsible for a significant part of the total system energy consumption. It may occupy more than 50% available die area and power consumption in modern embedded processors [Choi and Yeung 2013; Malik et al. 2000; Borkar 2001] due to its large on-chip area and high access frequency. Therefore, among the components in an embedded processor, on-chip cache hierarchy has been a key place to look for energy savings.

Applications generally have different cache requirements. Therefore, specializing the cache to an application's needs can save power consumption by 62% on average [Gordon-Ross et al. 2004]. To utilize such application characteristics, many reconfigurable cache architectures have been designed, such as the works in Modarressi et al. [2006], Settle et al. [2006], and Zhang et al. [2005]. Moreover, adaptive cache reconfiguration techniques [Lau et al. 2006, 2005b] are also proposed to reduce physical cache size for energy reduction without increasing the miss rate.

To illustrate the effectiveness of our MLPA, we apply it to dynamic cache reconfiguration. The design proposed in Zhang et al. [2005] is a reconfigurable cache architecture. It uses a very small custom hardware to manage cache configurations and controls the configurations via special registers. Due to no modification on the critical path, the cache latency does not increase. Therefore, we choose it as the basic cache architecture in our design. Figure 12 is an example of such a reconfigurable cache architecture. As shown in Figure 12(a), it consists of four separate banks, each of which acts as a separate way, and the size of each bank is 2KB. The basic configuration functions include way concatenation and way shutdown. Way concatenation logically concatenates



(a) Base cache bank (b) Way concatenation (c) Way shutdown

Fig. 12. Configurable cache architecture.

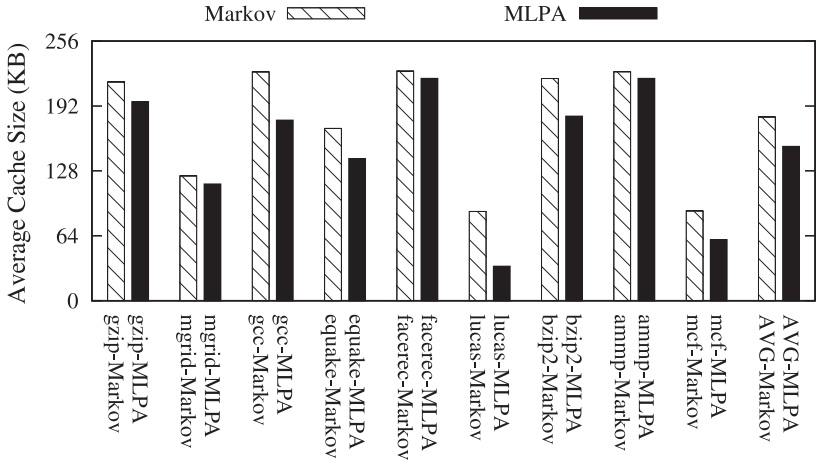


Fig. 13. Cache size for Markov and MLPA.

ways together, enabling configurable associativity. As show in Figure 12(b), it can be concatenated into a two-way 8KB cache. It can also be concatenated as a one-way 8KB cache. Way shutdown shuts down ways to vary cache size. As shown in Figure 12(c), two ways are shut down to implement two-way 4KB cache.

The cache is configured to be a 256KB cache with eight-way associativity (eight banks), and each 32KB bank can be dynamically powered on or off. The cache reconfiguration is achieved through powering off some banks when some intervals are executed. To obtain the resizing information, the best cache configuration of a phase is first collected when its first two intervals are executed. The best cache configuration of a phase is the smallest bank number that yields the same miss rate as that of the 256KB cache. After the configuration of a phase is decided, its best cache configuration is applied when the phase is predicted.

We compared our MLPA against the method in Lau et al. [2005b]. Figure 13 shows the average cache size for each approach. The data show the average data cache size used over the execution of the program. As the data show, our MLPA approach can reduce cache size by about 15% (i.e., about 26KB) compared to the fine-grained method [Lau et al. 2005b]. The major reason is that MLPA can achieve better phase prediction

accuracy. Although MLPA achieves more accurate prediction results for mgrid, the reconfigured cache size of the phases correctly predicted by MLPA is very close to that of the intervals executed before it. Therefore, the average cache size of MLPA is similar to that of Markov-2 for mgrid.

7. MULTILEVEL SAMPLING SIMULATION

In this section, we design a sampling-based simulation approach by applying MLPA to further illustrate the effectiveness of our MLPA.

7.1. Design of Multilevel Sampling Simulation

In this section, we will describe the design of our multilevel sampling framework. In this framework, we first select coarse-grained simulation points based on our coarse-grained sampling algorithm, which is referred as to COASTS. Then, we resample those coarse-grained simulation points to perform multilevel sampling via a fine-grained sampling algorithm.

7.1.1. Coarse-Grained Sampling (COASTS). For presentation clarity, we will refer to one occurrence of the phase as a phase instance. It could consist of multiple intervals. As they could lead to periodic behavior when the program is in a loop or in a recursive procedure, we refer to them as cyclic program structures in the rest of the article.

In the first-level sampling stage, we try to obtain larger iteration sizes (or interval sizes). Hence, we tend to choose outer loops or shallow recursive calls to form coarse-grained intervals instead of using inner ones with constrained interval sizes as in the SPM method [Lau et al. 2006]. Although a cyclic program structure exhibits some repetitive behavior, its iterations cannot be simply classified into a single phase because branches and memory accesses could lead to different dynamic behavior. Therefore, after choosing the appropriate level of a cyclic program structure, we classify the different iterations in the cyclic program structure into different phases based on the metrics. In this work, we still choose BBVs [Sherwood et al. 2002] as the metrics for phase identification because BBV performs better than other instruction-execution–related metrics, such as the working set as analyzed in Dhodapkar and Smith [2003]. Briefly, our approach requires the following three steps:

—*Collection of boundary information*: Currently, we use a method similar to that of Lau et al. [2006] and Huang et al. [2003] to collect the boundary information from dynamic profiling. Based on the profiling information, we first discard cyclic program structures with coverage of less than 1%, as they contribute little to the final simulation results.

—*Collection of metrics information*: Metrics information, such as BBVs, is collected for each *iteration interval* of the selected cyclic program structures during a profiling stage. After the original information is collected, BBVs are randomly projected onto their respective 15-dimension vectors. Such projections reduce computation complexity and storage requirements for the trace file. They also preserve behavior information for phase selection. Such a projection is widely used in prior techniques, such as SimPoint [Sherwood et al. 2002]. Then, BBV from each iteration interval is concatenated to form a signature vector. Such signature vectors are then normalized by having each element divided by the sum of all elements in the vector.

—*Coarse-grained sampling*: After the metrics information is collected, we apply the k-means clustering method [MacQueen 1967; Sherwood et al. 2002] for coarse-grained phase classification. For SPEC2000, the average coarse-grained phase number is three, and only four benchmarks are larger than three (four for gzip, six for equake, and five for fma3d). Therefore, the default K_{max} parameter for our coarse-grained phase clustering is three. Once the coarse-grained phases are classified, we choose

Table VI. Deviation Comparison under Different K Values for Fine-Grained Sampling

		Config A			Config B		
CPI	K	AVG	Worst	>5%	AVG	Worst	>5%
COASTS	—	1.98%	6.42%	4	1.47%	4.79%	0
Multilevel sampling	5	4.23%	14.36%	7	4.21%	14.87%	6
	10	3.37%	23.91%	5	3.63%	22.23%	3
	15	3.15%	20.54%	5	1.92%	7.82%	1
	20	3.70%	23.58%	6	2.46%	8.20%	2
	25	3.57%	20.56%	6	2.39%	8.29%	2
	30	3.46%	20.60%	6	2.61%	8.29%	3

the earliest interval of each coarse-grained phase as its representative (i.e., coarse-grained simulation point).

7.1.2. Fine-Grained Sampling. Although the coarse-grained sampling method used in the first-level sampling can effectively reduce the functional simulation time by decreasing the number of simulation points, the number of instructions in each simulation point can be increased. To further optimize the detailed simulation time in chosen coarse-grained simulation points, our multilevel sampling framework further resamples those coarse-grained simulation points in the second-level sampling via a fine-grained sampling method. If the size of a coarse-grained simulation point is larger than a threshold, we apply a fine-grained sampling method to resample it. Through selecting finer-grained simulation points within the coarser-grained simulation point, we can gain the advantages of both coarse- and fine-grained approaches.

In our current implementation, we use the SimPoint method as the second-level sampling method to select fine-grained simulation points to make a more reasonable comparison with prior fine-grained sampling methods. In the original SimPoint method, the default K_{max} parameter for the k-means clustering method [Sherwood et al. 2002] is 30. However, since the fine-grained simulation points are mainly used to represent the coarse-grained simulation points, a smaller value for the K_{max} parameter is sufficient. To choose a proper parameter for finer-grained simulation points, we compare the CPI deviations in different K values. The results are shown in Table VI. Besides the average results (AVG column of Table VI), we also count the worst results and the numbers of benchmarks whose CPI deviations compared to real CPI are higher than 5% (*>5%* column of Table VI). As the data show, with the k value increasing, all results improve. However, after K is equal to 15, all results change little regardless of the use of configuration *A* or *B*. Therefore, to achieve relatively lower deviations and avoid some extremely bad cases, we choose 15 as the default K value for the fine-grained sampling in our multilevel sampling simulation.

7.2. Evaluation

We evaluate our approach based on the SimpleScalar tool set 3.0 [Burger and Austin 1997] and SPEC2000 benchmarks with reference inputs for evaluation.

To compare our results with those from SimPoint [Sherwood et al. 2002], the base machine configuration of the simulation is the same as that in Perelman et al. [2003] and Lau et al. [2006], which is shown in Table II. To test the architecture sensitivity of our approach, we employ another architecture configuration shown in Table VII. This configuration includes a larger cache size and longer memory latency. CPI, L1 cache hit rate, and L2 cache hit rate are used to measure the accuracy. The baseline data are collected from complete execution of each benchmark for the original version of sim-outorder. We then collect results by simulating the points selected by our approach and those selected by the current SimPoint version (10M fixed length with $K_{max} = 30$

Table VII. Sensitivity Analysis Configuration

Sensitivity Analysis Configuration (Config B)	
Parameter	Value
Out-of-order issue	8-way decode, issue, commit width
ROB/LSQ entries	128/64
Registers	32 integer, 32 floating point
Functional units	6-integer ALU, 2-load/store units, 6-FP adders 4-integer MULT/DIV, 4-FP MULT/DIV
Instruction cache	32k direct mapping, 32 byte blocks, 1 cycle latency
Data cache	128k 2-way associative, 32 byte blocks, 1 cycle latency
Unified L2 cache	1Meg 4-way associative, 64 byte blocks, 23 cycle latency
Branch predictor	Combined, 16K BHT entries
Memory latency	330, 20 cycle access (first, following)

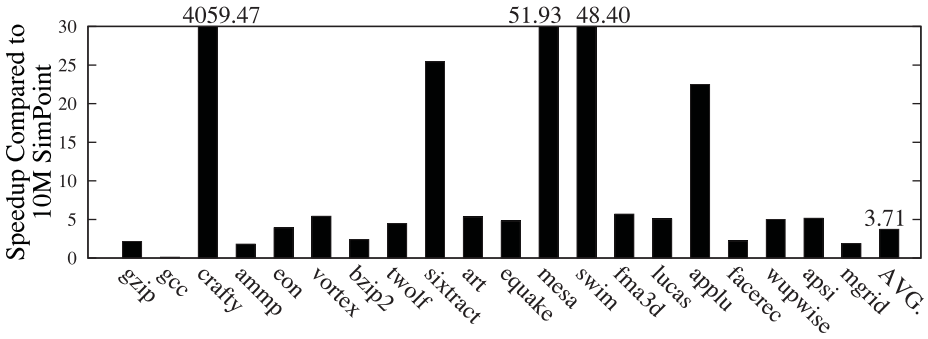


Fig. 14. Speedup of COASTS over 10M SimPoint.

and 100M fixed length with $K_{max} = 10$). The reasons we select 10M/100M SimPoint are as follows. First, 10M/100M SimPoint is the recommended interval length and is used in most of the SimPoint research. Second, although the SPM method uses VLIs, we cannot compare our results to it directly because it is not implemented in the current release of SimPoint. However, comparing with 10M SimPoint will not influence the final conclusions, as the SPM method has about the same simulation time as 10M SimPoint with a comparable error rate [Lau et al. 2006]. Since the default K_{max} of 10M SimPoint is 30, we use 300M as the resampling threshold as described in Section 7.1.2 (which is calculated as 10M * 30 = 300M.)

In the following sections, we will first present the experimental results of first-level sampling using COASTS alone and then illustrate the effect of our multilevel sampling framework by including second-level sampling.

7.3. Evaluation for COASTS

Figure 14, Figure 15, and Table VIII show the speedup and the deviation, respectively, using the COASTS approach and 10M/100M SimPoint. Average speedup (AVG) is the total simulation time of our approaches divided by that of 10M/100M SimPoint method. Due to space considerations, we only give the average deviation results and the deviation results in the worst case. In Table VIII, the AVG is the average deviation results, and the *Worst* is the worst deviation results. In Figure 14, some benchmarks (crafty, sixtract, mesa, swim, and applu) have very high speedup, as their cyclic program structure is very simple, selecting only quite a few coarse-grained points is enough. From the average results, our COASTS approach (first-level sampling) showed a speedup

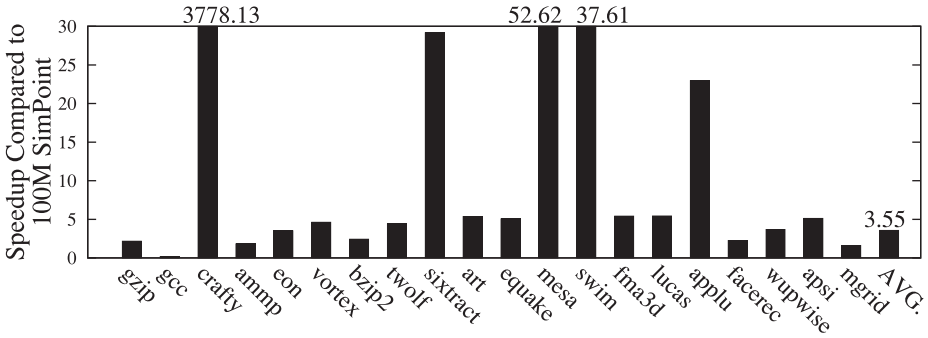


Fig. 15. Speedup of COASTS over 100MSimPoint.

Table VIII. Deviation Comparison under Different Configurations

	Config A		Config B	
CPI	AVG	Worst	AVG	Worst
COASTS	1.98%	6.42%	1.47%	4.79%
100M SimPoint	1.12%	8.43%	1.05%	2.94%
10M SimPoint	2.23%	10.03%	4.21%	17.86%
Multilevel sampling	3.15%	20.54%	1.92%	7.82%
L1 Cache Hit	AVG	Worst	AVG	Worst
COASTS	0.16%	0.89%	0.17%	0.89%
100M SimPoint	0.17%	1.09%	0.17%	0.98%
10M SimPoint	0.07%	0.25%	0.08%	0.26%
Multilevel sampling	0.36%	1.89%	0.35%	1.90%
L2 Cache Hit	AVG	Worst	AVG	Worst
COASTS	0.61%	2.51%	0.61%	2.62%
100M SimPoint	0.89%	2.72%	0.82%	2.92%
10M SimPoint	3.62%	23.32%	5.13%	16.22%
Multilevel sampling	2.35%	13.65%	3.84%	16.09%

Table IX. Simulation Points Statistics

Algorithm	Mean Sample Number	Mean Detail (inst. %)	Mean Function (inst. %)
COASTS	1.6	0.37%	2.21%
100M SimPoint	7.7	0.26%	88.28%
10M SimPoint	20.1	0.09%	93.76%
Multilevel sampling	7.3	0.05%	5.06%

of about 3.7X over 10M SimPoint and 3.6X over 100M SimPoint while maintaining comparable accuracy.

To identify the root cause of the faster simulation, we compare a couple of metrics in Table IX. In this table, *Mean Sample Number* is the average number of selected simulation points, *Mean Detail* is the total instructions simulated in detail divided by the total instructions, and *Mean Function* is the total instructions functionally simulated divided by the total instructions. Based on the preceding data, we can conclude the following:

—Since the coarse-grained samples can better catch the overall characteristics and hide instant fine-grained changes, fewer coarse-grained phases occur, and the last representative phase could be classified in a very early program position. In our study, even though the benchmarks in SPECINT2000 are quite complex, fewer than

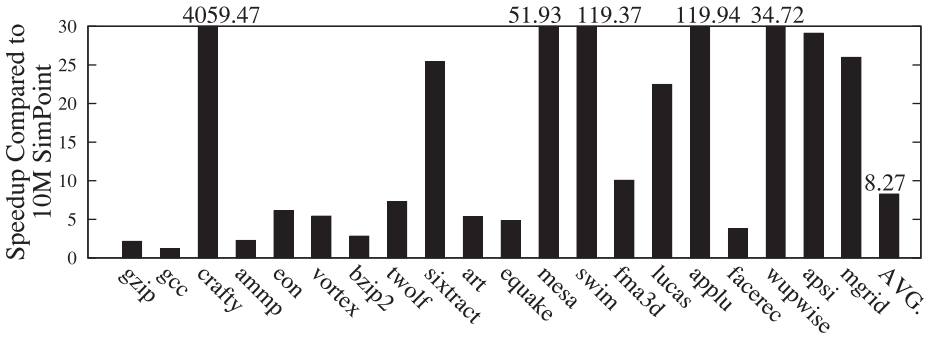


Fig. 16. Speedup of the multilevel sampling framework over 10M SimPoint.

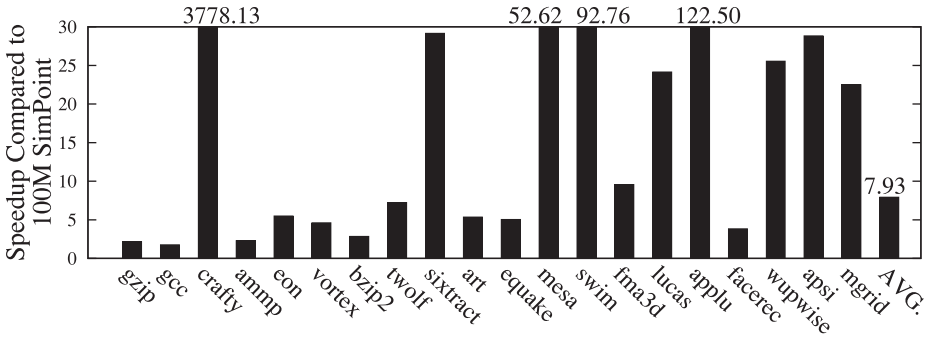


Fig. 17. Speedup of the multilevel sampling framework over 100M SimPoint.

three simulation points on average are needed. Coarse-grained samples lead to less total simulation time. As illustrated in Table IX, our mean functional part is larger, and it leads to longer detailed simulation time (0.37% of total instructions simulated vs. 0.09% in 10M SimPoint). However, the functional simulation time is significantly reduced because fewer simulation points near the end of program execution are selected (2.21% of total instructions simulated vs. 93.76% in 10M SimPoint). A larger reduction of the proportion of the simulated instructions leads to a higher performance improvement.

—Regardless of the choice of a coarse- or a fine-grained sampling method, the variable length interval (VLI) only makes the phase boundaries more natural but does not improve performance. For example, although the SPM method applies the VLI, the dominant functional simulation time is not reduced.

While only applying first-level sampling, more simulation time will be needed for gcc than that of 10M SimPoint. The reason is that gcc is a complex benchmark [Sherwood et al. 2002; Shen et al. 2004]. There are 56 iterations in its outermost loop using the reference input set, and their instruction counts vary significantly. Although we only selected two simulation points for gcc, the instruction count of one selected simulation point accounts for 60% of the total number of instructions executed in gcc. As a result, the COASTS approach needs to simulate more instructions in detail, and the total simulation time is significantly increased.

7.4. Evaluation for Multilevel Sampling

The detailed results of speedup and deviation when K = 15 are shown in Figure 16, Figure 17, and Table VIII. As the results show, fewer instructions are simulated in

detail in such a multilevel sampling framework, and it can achieve a speedup of about 8.3X over 10M SimPoint and 7.9X over 100M SimPoint while maintaining comparable accuracy. Even for gcc, our framework achieves 115% of the performance of the 10M SimPoint method.

The percentage of detailed simulation instructions for each benchmark is shown in Table IX. The results show that fewer instructions are simulated in detail in such a multilevel sampling framework. The underlying reason is that the total instructions in selected coarse-grained samples are much less than that of the entire program. As a result, fewer fine-grained simulation points are needed compared to those used to represent the entire program. Although two-level sampling could lead to a larger accumulation of errors, the increments of error rates are slight, as shown in Table VIII. Therefore, they are still comparable to those for COASTS and 10M SimPoint. The results in Table VIII also illustrate that the framework is not sensitive to different architectural configurations.

7.5. Discussion and Future Work

In this section, we will discuss the other common issues related to sampling techniques for architecture simulation.

7.5.1. Checkpoint-Based Techniques. In prior research, checkpoint techniques, such as the method in Wenisch et al. [2006], are proposed to optimize the simulation time for those fine-grained sampling methods. Moreover, they are also believed to provide an opportunity to simulate multiple simulation points in parallel. However, even without checkpoint techniques, different configurations or different applications can be simulated in parallel. Therefore, the major effect of checkpoint techniques is their ability to optimize the functional simulation time, which is similar to the effect achieved by the coarse-grained method used in our first-level sampling. However, our multilevel sampling framework can achieve two extra benefits that cannot be achieved by checkpoint-based techniques. First, besides reducing functional simulation time, our approach can also optimize the detailed simulation time. Second, our approach is more efficient in time and space overhead. Those checkpoint techniques require saving and maintaining a large dataset. For example, the method in Wenisch et al. [2006] has to collect and maintain 36TB of checkpoint data. Even when compressed, the data are still more than 10GB. In contrast, the metrics of coarse-grained intervals can be computed from those of the fine-grained intervals comprising it. Therefore, our multilevel sampling approach does not involve any additional space overhead.

Furthermore, checkpoint techniques are orthogonal to our multilevel sampling framework. They can be combined with our framework if necessary, which will be our future work.

7.5.2. Extension for Parallel Benchmarks. Simulating parallel benchmarks for multicore architectures is becoming increasingly important. In general, parallel benchmarks can be classified into two categories: multiprogram applications and multithreaded applications. For a multiprogram application, multiple independent single-threaded applications are simultaneously executed on different logical cores. Prior research [Van Biesbrouck et al. 2004, 2006] has proposed techniques that extend the existing representative serial sampling methods to multiprogram applications for multicore architectures. The sampling steps are as follows:

—The phase behavior in each program is analyzed, and a phase-ID trace is collected to represent the complete execution of a single program. Each phase is represented by a unique ID determined by serial phase analysis techniques, such as SimPoint or our COAST approach.

—A co-phase matrix is constructed to represent the possible combination of all of the phase-IDs from each program in the workload that can execute simultaneously on multicore platforms.
—When these programs are simulated, the phase-ID trace at a given point is used to look up the co-phase matrix. If a sample does not exist, the detailed simulation is performed and the per-thread IPC values are stored into the co-phase matrix for later use. If a sampling exists, the per-program IPC values are retrieved from the co-phase matrix. Through using each program's phase-ID trace, the number of instructions is calculated until the next phase change for each program. Based on IPC values, the fast-forward distance, which is the smallest number of cycles until the next phase change, is determined.

For a multithreaded application, different threads in a multithreaded program are executed on different logical cores. Different from multiprogram applications, the threads in an application need to interact and cooperate to finish a task through synchronization operations. Ignoring these synchronization events will lead to large accuracy loss [Carlson et al. 2013]. Therefore, for multithread sampling, all synchronization points need to be simulated in detail [Carlson et al. 2013]. When fast-forwarding distance is calculated, it is determined by two factors: the smallest cycles for the next phase change or the nearest synchronization point.

Based on the preceding analysis, it is easy to apply our multilevel approach for multiprogram sampling. We only need to replace a fine-grained method with our multilevel approach for the phase analysis of each program in the first step. One of potential advantages for our approach is that it can be used to reduce the co-phase matrix scale because less fine-grained phases are selected. However, for multithread sampling, it will be an open problem of how to deal with the relation between synchronization operations and MLPA. Our future work will investigate the potential of our current method to simulate multithread applications for multicore architectures.

8. CONCLUSION

In this article, we presented a comprehensive study of phase granularity and observed that a coarse-grained interval consists of stably distributed fine-grained intervals, which led to the design and implementation of our MLPA system. Experimental results showed that our system can improve the prediction accuracy by 20%, 39%, and 29% for *NP*, *PC*, and *PL* prediction, respectively, yet with little time and space overhead. To demonstrate the usefulness of our approach, we also applied it to dynamic cache reconfiguration and sampling simulation. Experimental results showed that our system can reduce the average cache size by 15% compared to the fine-grained strategies and achieve a speedup in simulation time of about 8.3X with similar accuracy compared to 10M SimPoint. For future work, we intend to apply our MLPA framework to other systems, such as power reduction and race detection, to reduce the performance overhead associated with current tools.

ACKNOWLEDGMENTS

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

REFERENCES

Rajeev Balasubramonian, David H. Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. 2000. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 245–257.

Shekhar Borkar. 2001. Low power design challenges for the decade (invited talk). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. ACM, New York, NY, 293–296.

Doug Burger and Todd M. Austin. 1997. *The SimpleScalar Tool Set, Version 2.0*. Technical Report 1342. Computer Sciences Department, University of Wisconsin, Madison, WI.

Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2013. Sampled simulation of multi-threaded applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.

I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. 1996. Analysis of branch prediction via data compression. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 128–137.

Chang-Burm Cho and Tao Li. 2006. Complexity-based program phase analysis and classification. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. 105–113.

Inseok Choi and Donald Yeung. 2013. *Symbiotic Cache Resizing for CMPs with Shared LLC*. Technical Report UMIACS-TR-2013. University of Maryland, College Park, MD.

Peter J. Denning and Stuart C. Schwartz. 1972. Properties of the working-set model. *Communications of the ACM* 15, 3, 191–198.

Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the International Symposium on Computer Architecture*. 233–244.

Ashutosh S. Dhodapkar and James E. Smith. 2003. Comparing program phase detection techniques. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 217.

Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. 2003. Characterizing and predicting program behavior and its variability. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 220.

Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*. 1–12.

Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. 2004. Method-level phase behavior in Java workloads. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 270–287.

Ann Gordon-Ross, Frank Vahid, and Nikil Dutt. 2004. Automatic tuning of two-level caches to embedded applications. In *Proceedings of the Conference on Design, Automation, and Test in Europe—Volume 1*. IEEE, Los Alamitos, CA, 10208.

Michael J. Hind, Vadakkedathu T. Rajan, and Peter F. Sweeney. 2003. *Phase Shift Detection: A Problem Classification*. Technical Report. IBM, Armonk, NY.

Michael Huang, Jose Renau, and Josep Torrellas. 2003. Positional adaptation of processors: Application to energy reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. 157–168.

Ted Huffmire and Tim Sherwood. 2006. Wavelet-based phase classification. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 95–104.

Canturk Isci and Margaret Martonosi. 2003. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 93.

Canturk Isci and Margaret Martonosi. 2006. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 133–144.

Richard A. Johnson and Dean A. Wichern. 2002. *Applied Multivariate Statistical Analysis (5th ed.)*. Prentice Hall.

Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. In *Proceedings of the 2nd Annual International Symposium on Computer Architecture*. 252–263.

Jeremy Lau, Erez Perelman, and Brad Calder. 2006. Selecting software phase markers with code structure analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*. 135–146.

Jeremy Lau, Erez Perelman, Greg Hamerly, Timothy Sherwood, and Brad Calder. 2005a. Motivation for variable length intervals and hierarchical phase behavior. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 135–146.

Jeremy Lau, Stefan Schoenmackers, and Brad Calder. 2004. Structures for phase classification. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 57–67.

Jeremy Lau, Stefan Schoenmackers, and Brad Calder. 2005b. Transition phase classification and prediction. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 278–289.

Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. 2003. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 180–190.

James MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. 281–297.

Afzal Malik, Bill Moyer, and Dan Cermak. 2000. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, Los Alamitos, CA, 241–243.

Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective sampling for lightweight data-race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 134–143.

Mehdi Modarressi, Shaahin Hessabi, and Maziar Goudarzi. 2006. A reconfigurable cache architecture for object-oriented embedded systems. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, Los Alamitos, CA, 959–962.

Arun A. Nair and Lizy Joh. 2008. Simulation points for SPEC 2006. In *Proceedings of the IEEE International Conference on Computer Design*. 38–46.

Erez Perelman, Greg Hamerly, and Brad Calder. 2003. Picking statistically valid and early simulation points. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 244.

Erez Perelman, Marzia Polito, Jean-Yves Bouguet, John Sampson, Brad Calder, and Carole Dulongh. 2006. Detecting phases in parallel applications on shared memory architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. 148–157.

Aashish Phansalkar, Ajay Joshi, Lieven Eeckhout, and Lizy Kurian John. 2005. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 10–20.

Alex Settle, Dan Connors, Enric Gibert, and Antonio Gonzalez. 2006. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing* 2, 2, 221–233.

Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality phase prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 165–176.

Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 45–57.

Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase tracking and prediction. In *Proceedings of the International Symposium on Computer Architecture*. 336–349.

Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder. 2006. Considering all starting points for simultaneous multithreading simulation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Los Alamitos, CA, 143–153.

Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2004. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 45–56.

Kenzo Van Craeynest and Lieven Eeckhout. 2011. The multi-program performance model: Debunking current practice in multi-core simulation. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 26–37.

Thomas Wenisch, Roland Wunderlich, Babak Falsafi, and James Hoe. 2006. Simulation sampling with livepoints. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 2–12.

Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. 84–97.

Joshua J. Yi, Sreekumar V. Kodakara, Resit Sendag, David J. Lilja, and Douglas M. Hawkins. 2005. Characterizing and comparing prevailing simulation techniques. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 266–277.

Chuanjun Zhang, Frank Vahid, and Walid Najjar. 2005. A highly configurable cache for low energy embedded systems. *ACM Transactions on Embedded Computing Systems* 4, 2, 363–387.

Received January 2013; revised February 2014; accepted April 2014