

Phase-Based Miss Rate Prediction Across Program Inputs

Xipeng Shen Yutao Zhong Chen Ding
{xshen, ytzhong, cding}@cs.rochester.edu

Computer Science Department, University of Rochester

Abstract. Previous work shows the possibility of predicting the cache miss rate (CMR) for all inputs of a program. However, most optimization techniques need to know more than the miss rate of the whole program. Many of them benefit from knowing miss rate of each execution phase of a program for all inputs.

In this paper, we describe a method that divides a program into phases that have a regular locality pattern. Using a regression model, it predicts the reuse signature and then the cache miss rate of each phase for all inputs. We compare the prediction with the actual measurement. The average prediction is over 98% accurate for a set of floating-point programs. The predicted CMR-traces matches the simulated ones in spite of dramatic fluctuations of the miss rate over time. This technique can be used for improving dynamic optimization, benchmarking, and compiler design.

1 Introduction

Memory hierarchy provides a way to ameliorate the problem of the widening speed gap between memory and CPU. The cache miss rate (CMR) measures the effect of the program reference patterns on a cache configuration. It determines the effectiveness of the memory hierarchy. Program reference patterns depend on program inputs. Therefore, to fully understand the behavior of a program, it is important to know the cache performance across all input data sets. In [32], Zhong *et. al.* present a method for predicting the average CMR of a whole program across different inputs. However, an average CMR is just a summary. A program's execution may include billions of instructions. Different parts have different computations and sometimes dramatically different behavior. The overall miss rate summarizes the behavior of billions of instructions into one number. The information loss is substantial. The loss becomes important as on-line dynamic techniques are more widely used. It is desirable to accurately predict the behavior of each execution phase of a program across all inputs. This paper reports our preliminary results toward this goal.

We build a locality-trace prediction system based on the locality phase analysis [29] and the miss-rate prediction [11, 28]. In our system, we first cut a program into phases that have regular locality patterns. They are called locality phases. We then build a regression model to predict the reuse signature of each phase on any given input, from which we predict the cache miss rate of each phase. We use a simple but effective method to predict the new phase sequence. The new CMR trace for the execution is

generated by aligning the predicted miss rates along the phase sequence. Our preliminary experiments show that the average prediction accuracy is over 98% for the cache hit rate of each phase¹. The predicted cache behavior matches with the measured behavior, despite the dramatically different cache behavior in different phases.

One use of this technique is for dynamic hardware and software optimization. In recent years, for their high flexibility of on-line adaptation to program executions, dynamic techniques are widely studied in both the hardware design and software optimization. These techniques tune the hardware (e.g. cache size) or optimization parameters (e.g. unrolling level) during a program’s execution. To find the best configuration, most techniques explore various configurations during the execution. The run-time exploration brings much overhead given many options to explore. Furthermore, because they use recent history to predict future behavior, the prediction tends to miss the best configuration when program behavior changes dramatically. Our technique helps to solve those two problems. It accurately predicts the locality and CMR of each section of an execution, which improves the efficiency of run-time exploration and helps to find the best system configuration.

Also, locality trace prediction helps benchmarking and the compiler design. It helps the selection of benchmarks and inputs, thus enables the comprehensive evaluation of dynamic techniques. It also helps to test and design compiler techniques since locality traces reveal the effects of the techniques on different phases of a program.

Our system adds a new dimension—the time—to program locality prediction. Most previous work varies the cache characteristics and measures the average CMR of a given program. Mattson *et al.* [25] measured CMR for a wide range of cache configurations in one pass; Hill [18] extended it to cache with limited associativity. Ding *et al.* [11, 32, 28] enabled the prediction of changes across different inputs, but only for the overall cache miss rate. Our work moves one step further and predicts the temporal changes in program locality.

In the rest of this paper, we first review the locality measurement, i.e. reuse signature, and its prediction in Section 2. Then we describe locality phase analysis in Section 3 and phase sequence prediction in Section 4. Section 5 shows our phase trace prediction system, followed by evaluations in Section 6. Finally, this paper concludes with related work and conclusions.

2 Locality Measurement and Prediction

In 1970, Mattson *et al.* defined the *LRU-stack distance* as the number of distinct data elements accessed between two consecutive references to the same element [25]. They summarized the locality of an execution by the distance histogram, which determines the miss rate of fully-associative LRU cache of all sizes. Building on decades of development by others, Ding and Zhong analyzed large traces by reducing the analysis cost to near linear time. They found that reuse-distance histograms change in predictable patterns in many programs [11]. For brevity we call the LRU stack distance the *reuse distance* as Ding and Zhong did.

¹ The relative error on the miss rate can be large when the miss rate is small.

We measure the program locality by the histogram of the reuse distance. Figure 1 shows an example, where each bin shows the percentage of memory references whose reuse distance falls within a range. Reuse distance histogram is also called the *reuse signature*. It provides the basis for CMR estimation. Given a cache size S , for fully-associative cache, all references with a reuse distance less than S are cache hits and the others are misses. Zhong *et al.* first estimated CMR according to this observation [32].

In [11], Ding and Zhong showed that reuse distance of many programs has a consistent pattern across different inputs. The pattern is a formula with the data size² as the parameter. For a given program input, it predicts the reuse signature of the corresponding execution. The pattern analysis method in [11] has two limitations. First, it uses only two training runs and is likely misled by their noises. Second, the accuracy is limited by the precision of data collection. Accurate prediction requires using large size program inputs and fine-grained reuse distance histograms. The space and time cost of the analysis is consequently high.

In [28], Shen *et al.* presented a set of techniques that overcome these limitations in two ways. First, they use regression to extract signature pattern from more than two training runs. Second, they employ multiple models. They provide a set of prediction methods with higher accuracy and lower overhead. The basic idea is as follows.

To predict the reuse signature of the run on a new input, the key step is to build a model reflecting the reuse pattern with the data size as the parameter. Multiple profiling-runs on inputs of different sizes generate multiple reuse signatures. An arbitrary one is picked as the standard reuse signature and its input as the standard input. It is assumed that the standard reuse signature is composed of multiple models, e.g. constant, linear, sub-linear models. Each model is a function mapping from the data size to the reuse distance. For example, linear model means the reuse distance lengthens linearly with the increase of the program input size. Those models split the standard reuse signature into many chunks. Other training reuse signatures can be formulated as combinations of those chunks. These formulas compose a large group of equations. The least-square linear regression gives the solution, i.e. the size of each chunk in the standard reuse signature. This concludes the building of reuse models. Since any reuse signature can be formulated as a combination of those chunks, the reuse signature of new inputs can be easily computed from the reuse model once the data size is known.

We use the same estimation method as Shen *et al.*, except that our task is to estimate the CMR of each phase instead of the whole execution.

3 Locality Phase Analysis

We use the locality phase analysis to cut a program into phases [29]. It was proposed recently by Shen *et al.*, using program locality for program phase analysis and combines both the data and control information. Figure 2 shows the reuse-distance trace of Tomcatv, a vectorized mesh generation program from SPEC95. X-axis shows the logical time, i.e. the number of memory accesses from the beginning of the execution, and

² The data size is measured by distance-based sampling [11]. For most programs, it is proportional to the number of different data elements or cache blocks accessed during an execution.

Y-axis shows the reuse distance of each data access. A point (x, y) shows that the memory access at the logical time x has the reuse distance y . The disruptive changes in the reuse patterns divide the trace into clearly separated phases. The same phases repeat in a fixed sequence. Shen *et al.* use wavelet filtering and optimal phase partitioning to find these phase-changing points. They then insert static markers into the program binary to enable on-line phase prediction. Their results show high consistency of CMRs for each phase—most with a standard deviation less than 0.00001.

Many phase-analysis methods exist: some start from the measurement of program behavior using a shifting window [3, 4, 9, 10, 12, 30] and some start from program structures, e.g. loops and subroutines [19, 22, 23]. There are three reasons for us to choose the locality phase analysis. First, locality phases have highly consistent behavior. Second, the analysis does not depend on the code structure being the same as the phase structure. Two iterations of a loop may have dramatically different behavior because of different data. Finally, the cache miss rate is closely related with the program locality.

4 Phase Sequence Prediction

The phase sequence of a program may be different for different inputs. We represent phase sequence as regular expressions. Expression 1 shows an example of the regular expression of a phase trace. p_x represents a phase ID. Brackets separate the phases into segments. The exponents show the repetitions of a segment. For many programs, the regular expressions from different runs have the same structure but different exponent values. To predict the phase sequence for these programs, we need to determine the value of the exponents for any given input. We now describe our prediction method.

$$p_1 p_2 \dots p_{15} (p_{16} p_{17} (p_{18} \dots p_{25} (p_{26} p_{27})^2 p_{28} p_{29} p_{30})^5 p_{31} p_{32})^{50} \quad (1)$$

We first manually find the input parameters, i.e. the numbers that change from an input to another input. The parameters can be the number of iterations, the data size, or the data content. We need to identify the parameters that control the phase sequence.

Through several training runs on different inputs, we collect multiple phase sequences and convert them into regular expressions. We used the following steps to find the correlation between the exponent values in the phase sequences and the parameters from the inputs.

- If an exponent value does not change in the training runs, its value is constant and independent of the input.
- If an exponent changes its value, we find the input parameter that has the same value.
- The remaining exponents are more complex functions of the input parameters. They may depend on multiple input parameters (e.g. the number of rows and columns of an input matrix). In the worst case, no automatic method can find the formula, since it is not computable. Still we can often determine or approximate the formula by studying the contribution of each parameter separately. Suppose in training runs we can change the value of just one parameter at a time. If we assume simple polynomials (linear, root, square, or constant patterns), then a fixed number of training

runs are enough to determine the coefficients through the following regression technique: first to determine the coefficient for each polynomial to fit the exponents of training runs; then to find the polynomial that generates the regular expressions closest to the phase sequences.

We studied five benchmarks shown in table 1. They come from different sources. Applu, FFT, Tomcatv, and Swim have parameters in their input files affecting the phase sequence. For Compress, its phase sequence can be changed by changing constants in its source code. They input either the matrix size or iteration numbers. Those benchmarks have simple phase sequence patterns—the exponents take either a constant or the same value as some input parameter. The first two steps in the previous list are enough to obtain perfect phase-sequence prediction.

5 Phase-based CMR Prediction System

In this section, we describe our whole prediction system as shown in Figure 5. The rectangles with thick boundaries show the four main steps of the system. We use gray parallel rectangles for input and output of the system, white parallel rectangles for intermediate results, and squares for operations.

The upper dotted box shows the training process, which constructs two models—the reuse signature model for each phase and the sequence model of all phases. Given a binary program, the first step is the locality phase analysis. It takes a training input to profile and apply wavelet filtering, optimal phase partitioning, and static marker insertions to generate a binary program with static phase markers. The second step is to execute the marked binary on each training input to collect the reuse-distance signature of each phase on each input and also the phase sequence of each execution. Regression analysis on those reuse signatures yields a reuse-signature model for each phase. A phase sequence model is built from the analysis of phase sequences.

After the construction of the two models, program locality-trace prediction is straightforward. The lower dotted box in Figure 5 shows the prediction process. Given a new input, the two models generated in the training process will give the corresponding reuse signatures and phase sequence according to the new data size. The CMR of each phase for any cache size can be estimated from its reuse signature. The CMR temporal trace can be obtained by aligning the CMRs along the new phase sequence.

6 Evaluation

We test five benchmarks, shown in table 1. FFT is an implementation from a textbook, Swim and Tomcatv are from SPEC95 suit. All have four training inputs and one test input. We pick these programs because they are readily available from our previous studies.

We use ATOM to instrument the programs for data collection and phase marker insertion. All programs are compiled by the Alpha compiler using “-O3” flag.

Our phase analysis cuts a program into phases. In table 1, the static phase numbers show the number of distinct phases. Dynamic phase number is the number of occurrences of static phases during an whole execution. At this step, we control the input so

Table 1. The input size and phase numbers of benchmarks with descriptions

	Applu	Compress	FFT	Swim	Tomcatv
Train-1	12^3	10^2	32x4	160^2	100^2
Train-2	20^3	10^3	64x4	200^2	200^2
Train-3	32^3	10^4	128x4	300^2	300^2
Train-4	40^3	10^6	256x4	400^2	400^2
Test	60^3	1.4×10^8	512x4	512^2	513^2
Num. static phases	195	4	7	17	9
Num. dynamic phases	645	52	37	92	15
Source	Spec95	Spec95	Textbook	Spec95	Spec95
Description	solution of five coupled nonlinear PDE's	an in-memory version of the common UNIX compression utility	fast Fourier transformation	finite difference approximations for shallow water equation	vectorized mesh generation

that all runs of a benchmark have the same phase sequence. The reason is to focus on phase-based CMR prediction, instead of being distracted by a changing phase sequence.

6.1 Experiment Results

Table 2 shows the average accuracy of the predicted phase reuse signatures and cache hit rates. The first row shows the accuracy when we measure the reuse of data elements. The second row shows the accuracy when we consider cache block reuse (of 32-byte blocks). While the prediction accuracy for the element pattern is less than 90% for some programs, the accuracy for the cache-block pattern is never below 93%.

Figure 4 shows the real and the predicted reuse signature of one phase of FFT, Swim and Tomcatv. We use black bars for real data, labeled by the benchmark names, and use gray bars for estimated data, labeled by the benchmark names with suffix “-p”. Most data references have large reuse distances for Swim; while as to Tomcatv, about 79% references have reuse distance shorter than 32. But all graphs show accurate prediction.

Table 2. Average accuracy of phase reuse signature prediction and cache-hit-rate prediction

Benchmark	Applu	Compress	FFT	Swim	Tomcatv	Average
Element reuse	0.867	0.841	0.963	0.832	0.921	0.885
Block reuse (32 bytes)	0.942	0.933	0.968	0.938	0.963	0.949
Cache-hit-rate (32 bytes)	0.982	0.980	0.983	0.982	0.979	0.981

Figure 5 shows the real and the predicted CMR-traces for FFT, Swim, and Tomcatv. (The Swim graph is enlarged for readability.) These three benchmarks have very different CMR curves. FFT has only 15 dynamic phases, whose hit rates are all larger than

0.7. The measured curve of *Swim* fluctuates dramatically. Some phases almost have no short-distance data reuse so that the cache hit rates are close to zero. The hit rates of other phases are greater than 0.79. For this dramatically changing curve, our method also yields accurate prediction. Tomcatv is a benchmark with less varying hit rates. It has twice as many phases as FFT has. All predicted curves are very close to the measured ones.

The last row of table 2 shows the average accuracy of the hit-rate prediction on an instance of fully associative cache of 32KB with 32-byte blocks. Our prediction method yields 98.1% average accuracy. The experiment on set-associative caches remains our future work. We expect a similar accuracy based on the previous results [32].

The prediction accuracy is similar to that of whole program hit rate prediction in [32]. The purpose of this work is not to improve the accuracy but to explore the possibility of predicting more—the prediction with time as a new dimension. The results show that the locality-phase based approach can work well for tests in the experiments.

7 Related Work

Cache behavior measurement Much work has been done in trace-driven simulation. Mattson *et al.* measured cache misses for all cache sizes in one simulation using a stack algorithm [25]. Hill extended the algorithm to cache systems with limited associativity [18]. Zhong *et al.* predicted cache misses across different program inputs by utilizing data reuse signature patterns [32]. They predicted the average cache miss rate of the whole program’s execution. Shen *et al.* extended their method using regression techniques and achieved more accurate locality prediction with less overhead [28]. Marin and Mellor-Crummey reported similar reuse-distance prediction used inside a comprehensive performance tool [24]. Fang *et al.* examined the reuse pattern of each program instruction of 11 SPEC2K CFP benchmark programs and predicted the miss rate of 90% of instructions with a 97% accuracy [13]. Our work, for the first time, predicts cache temporal behavior through the execution of a program across different inputs.

Phase analysis In this work, we use locality-based phase analysis [29], which yields program phases with different data access patterns. Those patterns can be used for accurate locality prediction. There are many other studies in program phase analysis. Allen and Cocke pioneered interval analysis to convert a program control flow into a hierarchy of regions [1]. Hsu and Kremer used program regions to control processor voltages to save energy [20]. Balasubramonian *et al.* [3], Huang *et al.* [22], and Magklis *et al.* [23] selected large enough procedures and loops as phases. Balasubramonian *et al.* and later researchers divide an execution into fixed-size windows, classify past intervals using machine or code-based metrics, and predict future intervals using last value, Markov, or table-driven predictors [3, 4, 9, 10, 12, 30]. The advantage of locality-based phase analysis is that each phase has its own data access pattern and the pattern is stable across multiple phase occurrences.

Locality analysis and prediction Dependence analysis analyzes data accesses and is used extensively in program locality measurement and improvement. In [2], Allen and

Kennedy gave a comprehensive description on this subject. Gannon *et al.* estimated the amount of memory needed by a loop [15]. Other researchers used various types of array sections to measure data access in loops and procedures. Such analysis includes linearization for high-dimensional arrays by Burke and Cytron [5], linear inequalities for convex sections by Triolet *et al.* [31], regular sections by Callahan and Kennedy [6], and reference list by Li *et al.* [21]. Havlak and Kennedy studied the effect of array section analysis on a wide range of programs [17]. Cascaval and Padua extended the dependence analysis to estimate the distance of data reuses [7]. One limitation of dependence analysis is that it does not model cache interference caused by the data layout. Ferrente *et al.* gave an efficient approximation of cache interference in a loop nest [14]. Recent studies used more expensive (worst-case super-exponential) tools to find the exact number of cache misses. Ghosh *et al.* modeled cache misses of a perfect loop nest as solutions to a set of linear Diophantine equations [16]. Chatterjee *et al.* studied solutions of general integer equations and used Presburger solvers like Omega [8]. The precise methods are effective for a single loop nest. For full applications, researchers have combined compiler analysis with cache simulation. McKinley and Temam carefully measured various types of reference locality within and between loop nests [26]. Mellor-Crummey *et al.* measured fine-grained reuse and program balance through a tool called HPCView [27].

8 Conclusions

In this paper, we described an approach to predict the temporal CMR-trace for all inputs to a program. Through locality phase analysis, we cut a program into segments with regular locality patterns. By building a regression model, we accurately predict the reuse signature of a program for a new input. CMR estimation yields the new CMR of each phase. We have developed a method for determining the phase sequence from input parameters. The CMR-trace for the new input is then calculated from the phase sequence and the per-phase locality prediction. Our experiment shows over 98% average accuracy of the phase CMR prediction. The CMR-traces predicted by our method coincides with the real CMR-trace very well for even dramatically changing curves. This approach is important to reduce explorations in dynamic systems and improve the accuracy of the prediction. It can also be used in benchmark and compiler design.

Our technique enables a new dimension of program locality prediction—the time. We expect that a similar technique can help to predict the temporal pattern of other performance metrics and yield better understanding of computer programs.

References

1. F. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19:137–147, 1976.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 2001.
3. R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures.

- In *Proceedings of the 33rd International Symposium on Microarchitecture*, Monterey, California, December 2000.
4. R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proceedings of International Symposium on Computer Architecture*, San Diego, CA, June 2003.
 5. M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.
 6. D. Callahan, J. Cocke, and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, October 1988.
 7. C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of International Conference on Supercomputing*, San Francisco, CA, June 2003.
 8. S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, UT, 2001.
 9. A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working-set analysis. In *Proceedings of International Symposium on Computer Architecture*, Anchorage, Alaska, June 2002.
 10. A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of International Symposium on Microarchitecture*, December 2003.
 11. C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
 12. E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.
 13. C. Fang, S. Carr, S. Onder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, Washington DC, June 2004.
 14. J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.
 15. K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.
 16. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4), 1999.
 17. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
 18. M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, University of California, Berkeley, November 1987.
 19. C.-H. Hsu and U. Kermer. The design, implementation and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
 20. C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scaling. In *Workshop on Power-Aware Computer Systems*, Cambridge, MA, November 2000.

21. Z. Li, P. Yew, and C. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, January 1990.
22. M. Huang and J. Renau and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, June 2003.
23. G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, , and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, June 2003.
24. G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*, New York City, NY, June 2004.
25. R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
26. K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, 1999.
27. J. Mellor-Crummey, R. Fowler, and D. B. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, June 2001.
28. X. Shen, Y. Zhong, and C. Ding. Regression-based multi-model prediction of data reuse signature. In *Proceedings of the 4th Annual Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, November 2003.
29. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004. (To appear).
30. T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of International Symposium on Computer Architecture*, San Diego, CA, June 2003.
31. R. Triolet, F. Irigoien, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.
32. Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.

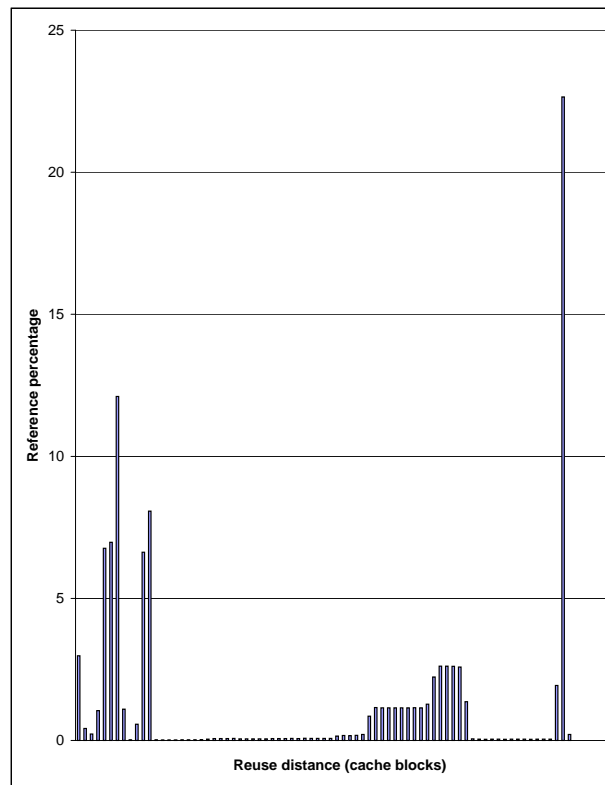


Fig. 1. The reuse distance histogram example

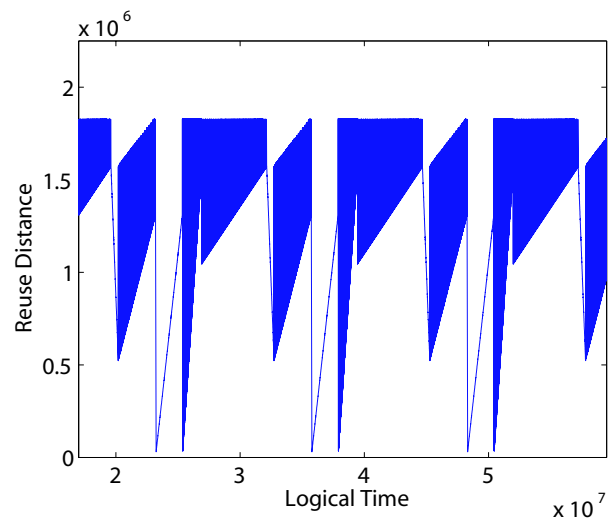
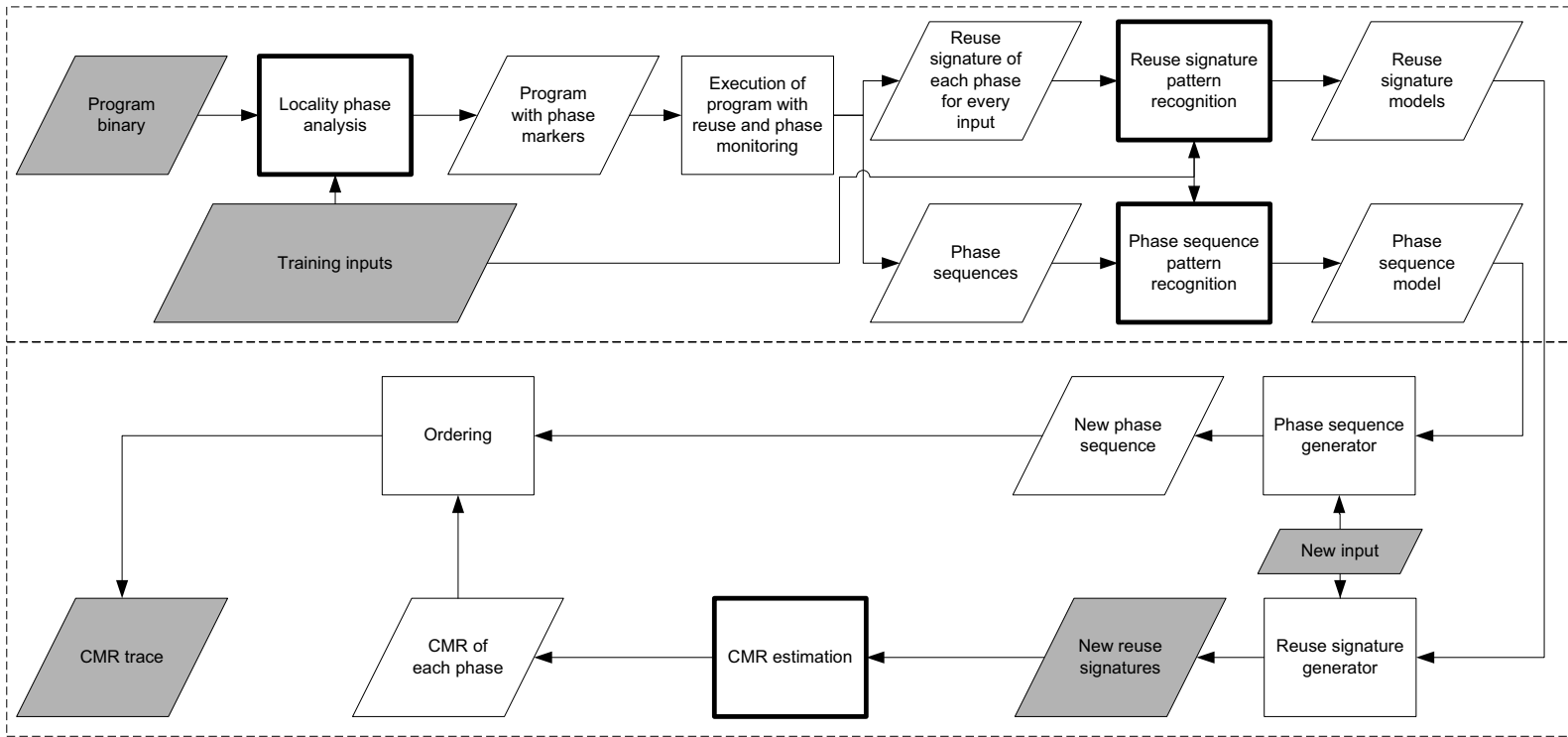
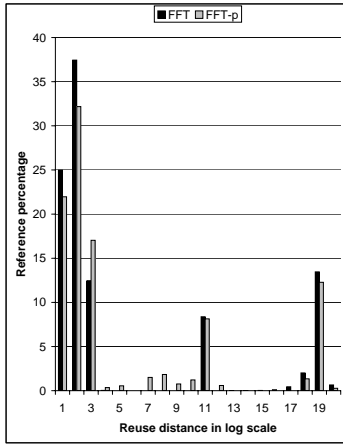


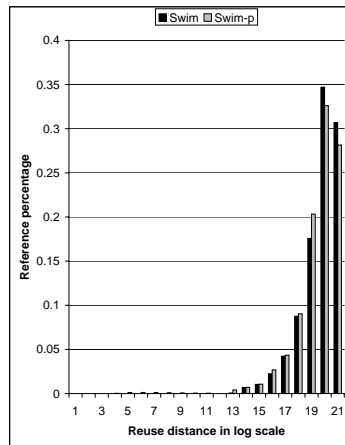
Fig. 2. The reuse-distance trace of Tomcatv

Fig. 3. Locality-trace prediction system

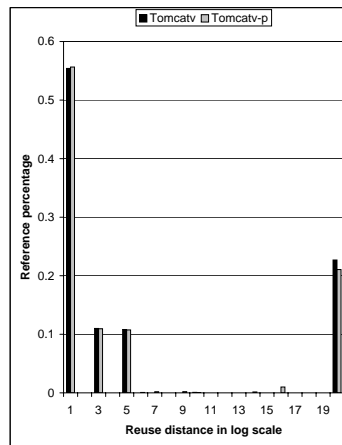




(a) A phase of FFT

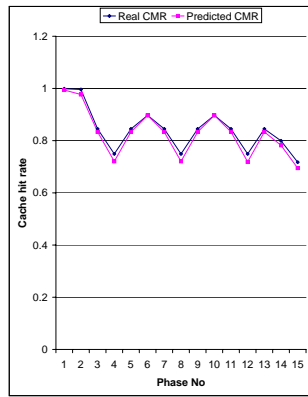


(b) A phase of Swim

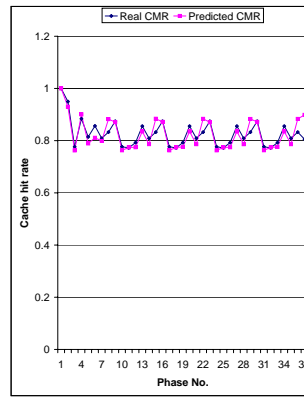


(c) A phase of Tomcatv

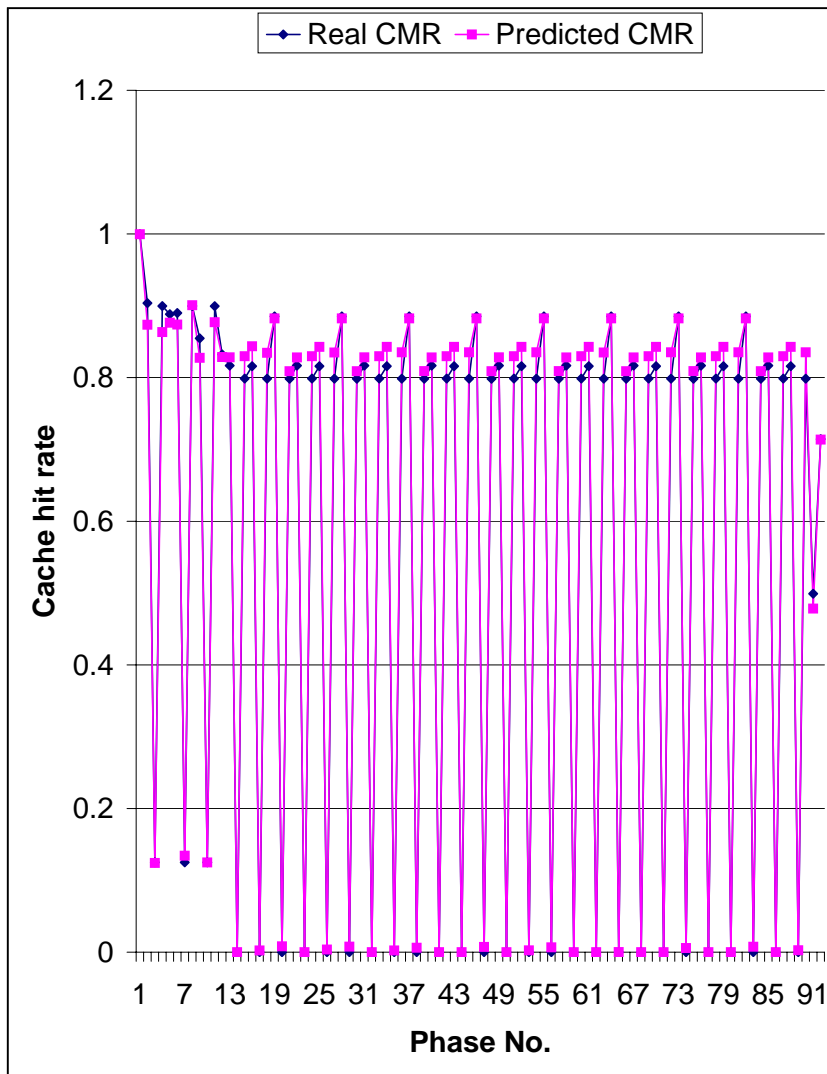
Fig. 4. Real and predicted reuse signatures



(a) FFT



(b) Tomcatv



(c) Swim

Fig. 5. Cache hit rate prediction