Locality Phase Prediction

Xipeng Shen Yutao Zhong Chen Ding

Computer Science Department, University of Rochester
{xshen,ytzhong,cding}@cs.rochester.edu

ABSTRACT

As computer memory hierarchy becomes adaptive, its performance increasingly depends on forecasting the dynamic program locality. This paper presents a method that predicts the locality phases of a program by a combination of locality profiling and run-time prediction. By profiling a training input, it identifies locality phases by sifting through all accesses to all data elements using variable-distance sampling, wavelet filtering, and optimal phase partitioning. It then constructs a phase hierarchy through grammar compression. Finally, it inserts phase markers into the program using binary rewriting. When the instrumented program runs, it uses the first few executions of a phase to predict all its later executions.

Compared with existing methods based on program code and execution intervals, locality phase prediction is unique because it uses locality profiles, and it marks phase boundaries in program code. The second half of the paper presents a comprehensive evaluation. It measures the accuracy and the coverage of the new technique and compares it with best known run-time methods. It measures its benefit in adaptive cache resizing and memory remapping. Finally, it compares the automatic analysis with manual phase marking. The results show that locality phase prediction is well suited for identifying large, recurring phases in complex programs.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization, compilers*

General Terms

Measurement, Performance, Algorithms

Keywords

program phase analysis and prediction, phase hierarchy, locality analysis and optimization, reconfigurable architecture, dynamic optimization

ASPLOS'04, October 7–13, 2004, Boston, Massachusetts, USA.

1. INTRODUCTION

Memory adaptation is increasingly important as the memory hierarchy becomes deeper and more adaptive, and programs exhibit dynamic locality. To adapt, a program may reorganize its data layout multiple times during an execution. Several studies have examined dynamic data reorganization at the program level [11, 15, 25, 27, 32] and at the hardware level [20, 35]. They showed impressive improvements in cache locality and prefetching efficiency. Unfortunately, these techniques are not yet widely used partly because they need manual analysis to find program phases that benefit from memory adaptation. In this paper, we show that this problem can be addressed by locality-based phase prediction.

Following early studies in virtual memory management by Batson and Madison [4] and by Denning [8], we define a locality phase as a period of a program execution that has stable or slow changing data locality inside the phase but disruptive transition periods between phases. For optimization purpose, we are interested in phases that are repeatedly executed with similar locality. While data locality is not easy to define, we use a precise measure in this paper. For an execution of a phase, we measure the locality by its miss rate across all cache sizes and its number of dynamic instructions. At run time, phase prediction means knowing a phase and its locality whenever the execution enters the phase. Accurate prediction is necessary to enable large-scale memory changes while avoiding any adverse effects.

Many programs have recurring locality phases. For example, a simulation program may test the aging of an airplane model. The computation sweeps through the mesh structure of the airplane repeatedly in many time steps. The cache behavior of each time step should be similar because the majority of the data access is the same despite local variations in control flow. Given a different input, for example another airplane model or some subparts, the locality of the new simulation may change radically but it will be consistent within the same execution. Similar phase behavior are common in structural, mechanical, molecular, and other scientific and commercial simulations. These programs have great demand for computing resources. Because of their dynamic but stable phases, they are good candidates for adaptation, if we can predict locality phases.

We describe a new prediction method that operates in three steps. The first analyzes the data locality in profiling runs. By examining the distance of data reuses in varying lengths, the analysis can "zoom in" and "zoom out" over long execution traces and detects locality phases using *variable-distance sampling*, *wavelet filtering*, and *optimal phase partitioning*. The second step then analyzes the instruction trace and identifies the phase boundaries in the code. The third step uses grammar compression to identify phase hierarchies and then inserts program markers through binary rewriting. During execution, the program uses the first few instances of a

phase to predict all its later executions. The new analysis considers both program code and data access. It inserts static markers into the program binary without accessing the source code.

Phase prediction has become a focus of much recent research. Most techniques can be divided into two categories. The first is interval based. It divides a program execution into fixed-length intervals and predicts the behavior of future intervals from past observations. Interval-based prediction can be implemented entirely and efficiently at run time [2, 3, 9, 10, 13, 30]. It handles arbitrarily complex programs and detects dynamically changing patterns. However, run-time systems cannot afford detailed data analysis much beyond counting the cache misses. In addition, it is unclear how to pick the interval length for different programs and for different inputs of the same program. The second category is code based. It marks a subset of loops and functions as phases and estimates their behavior through profiling [17, 21, 22]. Pro-active rather than reactive, it uses phase markers to control the hardware and reduce the need for run-time monitoring. However, the program structure may not reveal its locality pattern. A phase may have many procedures and loops. The same procedure or loop may belong to different locality phases when accessing different data at different invocations. For example, a simulation step in a program may span thousands of lines of code with intertwined function calls and indirect data access.

In comparison, the new technique combines locality analysis and phase marking. The former avoids the use of fixed-size windows in analysis or prediction. The latter enables pro-active phase adaptation. In addition, the phase marking considers all instructions in the program binary in case the loop and procedure structures are obfuscated by an optimizing compiler.

In evaluation, we show that the new analysis finds recurring phases of widely varying sizes and nearly identical locality. The phase length changes in tune with program inputs and ranges from two hundred thousand to three billion instructions—this *length* is predicted with 99.5% accuracy. We compare it with other phase prediction methods, and we show its use in adaptive cache resizing and phase-based memory remapping.

Locality phase prediction is not effective on all programs. Some programs may not have predictable phases. Some phases may not be predictable from its data locality. We limit our analysis to programs that have large predictable phases, which nevertheless include important classes of dynamic programs. For some programs such as a compiler or a database, the analysis can still identify phases but cannot predict the exact locality.

2. HIERARCHICAL PHASE ANALYSIS

This section first motivates the use of locality analysis and then describes the steps of locality-based phase prediction.

2.1 Locality Analysis Using Reuse Distance

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 [24]. 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, our earlier work reduced the analysis cost to near linear time. A number of recent studies found that reuse-distance histograms change in predictable patterns in many programs [12, 14, 23, 28]. In this work we go one step further to see whether predictable patterns exist for subparts of a program. For brevity we call the LRU stack distance between two accesses of the same data the *reuse distance* of the second access.

Reuse distance reveals patterns in program locality. We use the example of *Tomcatv*, a vectorized mesh generation program from SPEC95 known for its highly memory-sensitive performance. Figure 1 shows the reuse distance trace. Each data access is a point in the graph—the x-axis gives the logical time (i.e. the number of data accesses), and the y-axis gives the reuse distance [1]. The points are so numerous that they emerge as solid blocks and lines.

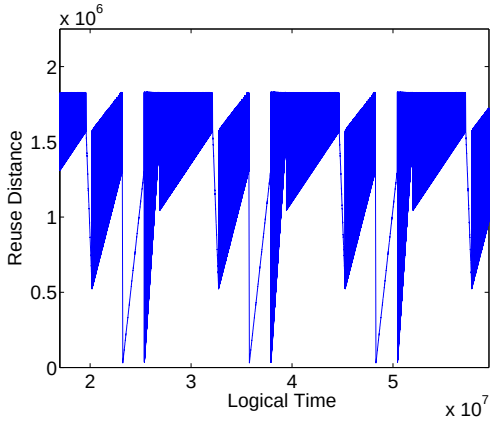

Figure 1: The reuse-distance trace of Tomcatv

The reuse distance of data access changes continuously throughout the trace. We define a phase change as an abrupt change in data reuse pattern. In this example, the abrupt changes divide the trace into clearly separated phases. The same phases repeat in a fixed sequence. Reading the code documentation, we see indeed that the program has a sequence of time steps, each has five substeps—preparation of data, residual values, solving two tridiagnoal systems, and adding corrections. What is remarkable is that we could see the same pattern from the reuse distance trace without looking at the program.

The example confirms four commonly held assumptions about program locality. First, the data locality may change constantly in an execution; however, major shifts in program locality are marked by radical rather than gradual changes. Second, locality phases have different lengths. The size of one phase has little relation with the size of others. Third, the size changes greatly with program inputs. For example, the phases of *Tomcatv* contain a few hundred million memory accesses in a training run but over twenty-five billion memory accesses in a test run. Finally, a phase often recurs with similar locality. *A phase is a unit of repeating behavior rather than a unit of uniform behavior.* To exploit these properties, locality phase prediction uses reuse distance to track fine-grain changes and find precise phase boundaries. It uses small training runs to predict larger executions.

Reuse distance measures locality better than pure program or hardware measures. Compiler analysis cannot fully analyze locality in programs that have dynamic data structures and indirect data access. The common hardware measure, the miss rate, is defined over a window. Even regular programs may have irregular cache miss rate distributions when we cut them into windows, as shown later in Figure 3. It is difficult to find a fixed window size that matches the phases of unequal lengths. We may use the miss trace, but a cache miss is a binary event—hit or miss for a given cache configuration. In comparison, reuse distance is a precise scale. It is purely a program property, independent of hardware configurations.

[1]To reduce the size of the graph, we show the reuse distance trace after variable-distance sampling described in Section 2.2.1.

To speculate is to see. Reuse distance shows an interesting picture of program locality. Next we present a system that automatically uncovers the hierarchy of locality phases from this picture.

2.2 Off-line Phase Detection

Given the execution trace of training runs, phase detection operates in three steps: variable-distance sampling collects the reuse distance trace, wavelet filtering finds abrupt changes, and finally, optimal phase partitioning locates the phase boundary.

2.2.1 Variable-Distance Sampling

Instead of analyzing all accesses to all data, we sample a small number of representative data. In addition, for each data, we record only long-distance reuses because they reveal global patterns. Variable-distance sampling is based on the distance-based sampling described by Ding and Zhong [12]. Their sampler uses ATOM to generate the data access trace and monitors the reuse distance of every access. When the reuse distance is above a threshold (the *qualification threshold*), the accessed memory location is taken as a data sample. A later access to a data sample is recorded as an access sample if the reuse distance is over a second threshold (the *temporal threshold*). To avoid picking too many data samples, it requires that a new data sample to be at least a certain space distance away (the *spatial threshold*) in memory from existing data samples.

The three thresholds in Ding and Zhong's method are difficult to control. Variable-distance sampling solves this problem by using dynamic feedback to find suitable thresholds. Given an arbitrary execution trace, its length, and the target number of samples, it starts with an initial set of thresholds. It periodically checks whether the rate of sample collection is too high or too low considering the target sample size. It changes the thresholds accordingly to ensure that the actual sample size is not far greater than the target. Since sampling happens off-line, it can use more time to find appropriate thresholds. In practice, variable-distance sampling finds 15 thousand to 30 thousand samples in less than 20 adjustments of thresholds. It takes several hours for the later steps of wavelet filtering and optimal phase partitioning to analyze these samples, although the long time is acceptable for our off-line analysis and can be improved by a more efficient implementation (currently using Matlab and Java).

The variable-distance sampling may collect samples at an uneven rate. Even at a steady rate, it may include partial results for executions that have uneven reuse density. However, the target sample size is large. The redundancy ensures that these samples together contain elements in all phase executions. If a data sample has too few access samples to be useful, the next analysis step will remove them as noise.

2.2.2 Wavelet Filtering

Viewing the sample trace as a signal, we use the *Discrete Wavelet Transform (DWT)* as a filter to expose abrupt changes in the reuse pattern. The DWT is a common technique in signal and image processing [7]. It shows the change of frequency over time. As a mutli-resolution analysis, the DWT applies two functions to data: the scale function and the wavelet function. The first smooths the signal by averaging its values in a window. The second calculates the magnitude of a range of frequencies in the window. The window then shifts through the whole signal. After finishing the calculations on the whole signal, it repeats the same process at the next level on the scaled results from the last level instead of on the original signal. This process may continue for many levels as a multi-resolution process. For each point on each level, a scaling and a wavelet coefficient are calculated using the variations of the following basic formulas:

$$c_j(k) = < f(x),\ 2^{-j}\phi(2^{-j}x - k) >$$
$$w_j(k) = < f(x),\ 2^{-j}\psi(2^{-j}x - k) >$$

where, $< a, b >$ is the scaler product of a and b, $f(x)$ is the input signal, j is the analysis level, ϕ and ψ are the scaling and wavelet function respectively. Many different wavelet families exist in the literature, such as *Haar, Daubechies*, and *Mexican-hat*. We use *Daubechies-6* in our experiments. Other families we have tested produce a similar result. On high-resolution levels, the points with high wavelet coefficient values signal abrupt changes; therefore they are likely phase changing points.

The wavelet filtering takes the reuse-distance trace of each data sample as a signal, then computes the level-1 coefficient for each access and removes from the trace the accesses with a low wavelet coefficient value. An access is kept only if its coefficient $\omega > m + 3\delta$, where m is the mean and δ is the standard deviation. The difference between this coefficient and others is statistically significant. We have experimented with coefficients of the next four levels and found the level-1 coefficient adequate.

Figure 2 shows the wavelet filtering for the access trace of a data sample in *MolDyn*, a molecular dynamics simulation program. The filtering removes accesses during the gradual changes because they have low coefficients. Note that it correctly removes accesses that correspond to local peaks. The remaining four accesses indicate global phase changes.

Sherwood et al. used the Fourier transform to find periodic patterns in execution trace [29]. The Fourier transform shows the frequencies appeared during the whole signal. In comparison, wavelets gives the *time-frequency* or the frequencies appeared over time. Joseph et al. used wavelets to analyze the change of processor voltage over time and to make on-line predictions using an efficient Haar-wavelet implementation [18]. We use wavelets similar to their off-line analysis but at much finer granularity (because of the nature of our problem). Instead of filtering the access trace of all data, we analyze the sub-trace for each data element. This is critical because a gradual change in the subtrace may be seen as an abrupt change in the whole trace and cause false positives in the wavelet analysis. We will show an example later in Figure 3 (b), where most abrupt changes seen from the whole trace are not phase changes.

After it filters the sub-trace of each data sample, the filtering step recompiles the remaining accesses of all data samples in the order of logical time. The new trace is called a *filtered trace*. Since the remaining accesses of different data elements may signal the same phase boundary, we use optimal phase partitioning to further remove these redundant indicators.

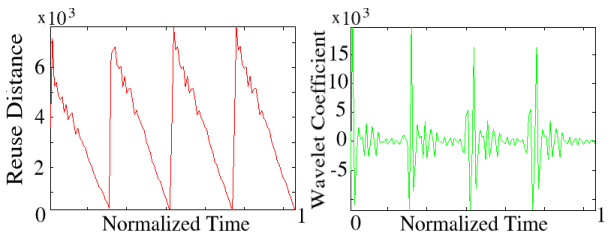

Figure 2: A wavelet transform example, where gradual changes are filtered out

2.2.3 Optimal Phase Partitioning

At a phase boundary, many data change their access patterns. Since the wavelet filtering removes reuses of the same data within a phase, the remaining is mainly accesses to different data samples clustered at phase boundaries. These two properties suggest two conditions for a good phase partition. First, a phase should include accesses to as many data samples as possible. This ensures that we do not artificially cut a phase into smaller pieces. Second, a phase should not include multiple accesses of the same data sample, since data reuses indicate phase changes in the filtered trace. The complication, however, comes from the imperfect filtering by the wavelet transform. Not all reuses represent a phase change.

We convert the filtered trace into a directed acyclic graph where each node is an access in the trace. Each node has a directed edge to all succeeding nodes. Each edge (from access a to b) has a weight defined as $w = \alpha r + 1$, where $1 \geq \alpha \geq 0$, and r is the number of node recurrences between a and b. For example, the trace $aceefgefbd$ has two recurrences of e and one recurrence of f between c and b, so the edge weight between the two nodes is $3\alpha + 1$.

Intuitively, the weight measures how fit the segment from a to b is as a phase. The two factors in the weight penalize two tendencies. The first is the inclusion of reuses, and the second is the creation of new phases. The optimal case is a minimal number of phases with least reuses in each phase. Since the trace is not perfect, the weight and the factor α control the relative penalty for too large or too small phases. If α is 1, we prohibit any reuses in a phase. We may have as many phases as the length of the filtered trace. The result when $\alpha \geq 1$ is the same as $\alpha = 1$. If α is 0, we get one phase. In experiments, we found that the phase partitions were similar when α is between 0.2 and 0.8, suggesting that the noise in the filtered trace was acceptable. We used $\alpha = 0.5$ in the evaluation.

Once α is determined, shortest-path analysis finds a phase partition that minimizes the total penalty. It adds two nodes: a source node that has directed edges flowing to all original nodes, and a sink node that has directed edges coming from all original nodes. Any directed path from the source to the sink gives a phase partition. The weight of the path is its penalty. Therefore, the best phase partition gives the least penalty, and it is given by the shortest path between the source and the sink.

Summary of off-line phase detection The program locality is a product of all accesses to all program data. The phase detection first picks enough samples in time and space to capture the high-level pattern. Then it uses wavelets to remove the temporal redundancy and phase partitioning to remove the spatial redundancy. The next challenge is marking the phases in program code. The wavelet filtering loses accurate time information because samples are considered a pair at a time (to measure the difference). In addition, the locality may change through a transition period instead of a transition point. Hence the exact time of a phase change is difficult to attain. We address this problem in the next step.

2.3 Phase Marker Selection

The instruction trace of an execution is recorded at the granularity of basic blocks. The result is a block trace, where each element is the label of a basic block. This step finds the basic blocks in the code that uniquely mark detected phases. Previous program analysis considered only a subset of code locations, for example function and loop boundaries [17, 21, 22]. Our analysis examines all instruction blocks, which is equivalent to examining all program instructions. This is especially important at the binary level, where the high level program structure may be lost due to aggressive compiler transformations such as procedure in-lining, software pipelining, loop fusion, and code compression.

As explained earlier, phase detection finds the number of phases but cannot locate the precise time of phase transitions. The precision is in the order of hundreds of memory accesses while a typical basic block has fewer than ten memory references. Moreover, the transition may be gradual, and it is impossible to locate a single point. We solve this problem by using the frequency of the phases instead of the time of their transition.

We define the frequency of a phase by the number of its executions in the training run. Given the frequency found by the last step, we want to identify a basic block that is always executed at the beginning of a phase. We call it the *marker block* for this phase. If the frequency of a phase is f, the marker block should appear no more than f times in the block trace. The first step of the marker selection filters the block trace and keeps only blocks whose frequency is no more than f. If a loop is a phase, the filtering will remove the occurrences of the loop body block and keep only the header and the exit blocks. If a set of mutual recursive functions forms a phase, the filtering will remove the code of the functions and keep only the ones before and after the root invocation. After filtering, the remaining blocks are candidate markers.

After frequency-based filtering, the removed blocks leave large blank regions between the remaining blocks. If a blank region is larger than a threshold, it is considered as a phase execution. The threshold is determined by the length distribution of the blank regions, the frequency of phases, and the execution length. Since the training runs had at least 3.5 million memory accesses, we simply used 10 thousand instructions as the threshold. In other words, a phase execution must consume at least 0.3% of the total execution to be considered significant. We can use a smaller threshold to find sub-phases after we find large phases.

Once the phase executions are identified, the analysis considers the block that comes after a region as markers marking the boundary between the two phases. Two regions are executions of the same phase if they follow the same code block. The analysis picks markers that mark most if not all executions of the phases in the training run. We have considered several improvements that consider the length of the region, use multiple markers for the same phase, and correlate marker selection across multiple runs. However, this basic scheme suffices for programs we tested.

Requiring the marker frequency to be no more than the phase frequency is necessary but not sufficient for phase marking. A phase may be fragmented by infrequently executed code blocks. However, a false marker cannot divide a phase more than f times. In addition, the partial phases will be regrouped in the next step, phase-hierarchy construction.

2.4 Marking The Phase Hierarchy

Hierarchical construction Given the detected phases, we construct a phase hierarchy using grammar compression. The purpose is to identify composite phases and increase the granularity of phase prediction. For example, for the *Tomcatv* program showed in Figure 1, every five phase executions form a time step that repeats as a composite phase. By constructing the phase hierarchy, we find phases of the largest granularity.

We use SEQUITUR, a linear-time and linear-space compression method developed by Nevill-Manning and Witten [26]. It compresses a string of symbols into a Context Free Grammar. To build the phase hierarchy, we have developed a novel algorithm that extracts phase repetitions from a compressed grammar and represents them explicitly as a regular expression. The algorithm recursively converts non-terminal symbols into regular expressions. It remembers previous results so that it converts the same non-terminal symbol only once. A merge step occurs for a non-terminal once its

right-hand side is fully converted. Two adjacent regular expressions are merged if they are equivalent (using for example the equivalent test described by Hopcroft and Ullman [16]).

SEQUITUR was used by Larus to find frequent code paths [19] and by Chilimbi to find frequent data-access sequences [5]. Their methods model the grammar as a DAG and finds frequent subsequences of a given length. Our method traverses the non-terminal symbols in the same order, but instead of finding sub-sequences, it produces a regular expression.

Phase marker insertion The last step uses binary rewriting to insert markers into a program. The basic phases (the leaves of the phase hierarchy) have unique markers in the program, so their prediction is trivial. To predict the composite phases, we insert a predictor into the program. Based on the phase hierarchy, the predictor monitors the program execution and makes predictions based on the on-line phase history. Since the hierarchy is a regular expression, the predictor uses a finite automaton to recognize the current phase in the phase hierarchy. In the programs we tested so far, this simple method suffices. The cost of the markers and the predictor is negligible because they are invoked once per phase execution, which consists of on average millions of instructions as shown in the evaluation.

3. EVALUATION

We conduct four experiments. We first measure the granularity and accuracy of phase prediction. We then use it in cache resizing and memory remapping. Finally, we test it against manual phase marking. We compare with other prediction techniques in the first two experiments.

Our test suite is given in Table 1. We pick programs from different sets of commonly used benchmarks to get an interesting mix. They represent common computation tasks in signal processing, combinatorial optimization, structured and unstructured mesh and N-body simulations, a compiler, and a database. *FFT* is a basic implementation from a textbook. The next six programs are from SPEC: three are floating-point and three are integer programs. Three are from SPEC95 suite, one from SPEC2K, and two (with small variation) are from both. Originally from the CHAOS group at University of Maryland, *MolDyn* and *Mesh* are two dynamic programs whose data access pattern depends on program inputs and changes during execution [6]. They are commonly studied in dynamic program optimization [11, 15, 25, 32]. The floating-point programs from SPEC are written in Fortran, and the integer programs are in C. Of the two dynamic programs, *MolDyn* is in Fortran, and *Mesh* is in C. We note that the choice of source-level languages does not matter because we analyze and transform programs at the binary level.

For programs from SPEC, we use the *test* or the *train* input for phase detection and the *ref* input for phase prediction. For the prediction of *Mesh*, we used the same mesh as that in the training run but with sorted edges. For all other programs, the prediction is tested on executions hundreds times longer than those used in phase detection.

We use ATOM to instrument programs to collect the data and instruction trace on a Digital Alpha machine [31]. All programs are compiled by the Alpha compiler using "-O5" flag. After phase analysis, we again use ATOM to insert markers into programs.

3.1 Phase Prediction

We present results for all programs except for *Gcc* and *Vortex*, which we discuss at the end of this section. We first measure the phase length and then look at the phase locality in detail.

Table 2 shows two sets of results. The upper half shows the accuracy and coverage of strict phase prediction, where we require that phase behavior repeats exactly including its length. Except for *MolDyn*, the accuracy is perfect in all programs, that is, *the number of the executed instructions is predicted exactly at the beginning of a phase execution.* We measure the coverage by the fraction of the execution time spent in the predicted phases. The high accuracy requirement hurts coverage, which is over 90% for four programs but only 46% for *Tomcatv* and 13% for *MolDyn*. If we relax the accuracy requirement, then the coverage increases to 99% for five programs and 98% and 93% for the other two, as shown in the lower half of the table. The accuracy drops to 90% in *Swim* and 13% in *MolDyn*. *MolDyn* has a large number of uneven phases when it finds neighbors for each particle. In all programs, the phase prediction can attain either perfect accuracy, full coverage, or both.

Table 1: Benchmarks

Benchmark	Description	Source
FFT	fast Fourier transformation	textbook
Applu	solving five coupled nonlinear PDE's	Spec2KFp
Compress	common UNIX compression utility	Spec95Int
Gcc	GNU C compiler 2.5.3	Spec95Int
Tomcatv	vectorized mesh generation	Spec95Fp
Swim	finite difference approximations for shallow water equation	Spec95Fp
Vortex	an object-oriented database	Spec95Int
Mesh	dynamic mesh structure simulation	CHAOS
MolDyn	molecular dynamics simulation	CHAOS

The granularity of the phase hierarchy is shown in Table 3 by the average size of the smallest (leaf) phases and the largest composite phases. The left half shows the result of the detection run, and the right half shows the prediction run. The last row shows the average across all programs. With the exception of *Mesh*, which has two same-length inputs, the prediction run is larger than the detection run by, on average, 100 times in execution length and 400 times in the phase frequency. The average size of the leaf phase ranges from two hundred thousand to five million instructions in the detection run and from one million to eight hundred million in the prediction run. The largest phase is, on average, 13 times the size of the leaf phase in the detection run and 50 times in the prediction run.

The results show that the phase length is anything but uniform. The prediction run is over 1000 times longer than the detection run for *Applu* and *Compress* and nearly 5000 times longer for *MolDyn*. The longer executions may have about 100 times more phase executions (*Tomcatv, Swim*, and *Applu*) and over 1000 times larger phase size (in *Compress*). The phase size differs from phase to phase, program to program, and input to input, suggesting that a single interval or threshold would not work well for this set of programs.

3.1.1 Comparison of Prediction Accuracy

Figure 3 shows the locality of two representative programs—*Tomcatv* and *Compress*—in two columns of three graphs each. The upper graphs show the phase detection in training runs. The other graphs show phase prediction in reference runs. The upper graphs show a fraction of the sampled trace with vertical lines marking the phase boundaries found by variable-distance sampling, wavelet filtering, and optimal phase partitioning. The lines fall exactly at the points where abrupt changes of reuse behavior happen, showing the effect of these techniques. The phases have different lengths. Some are too short in relative length and the two boundaries be-

Table 2: The accuracy and coverage of phase prediction

Benchmarks		FFT	Applu	Compress	Tomcatv	Swim	Mesh	MolDyn	**Average**
Strict accuracy	Accuracy(%)	100	100	100	100	100	100	96.47	99.50
	Coverage(%)	96.41	98.89	92.39	45.63	72.75	93.68	13.49	73.32
Relaxed accuracy	Accuracy(%)	99.72	99.96	100	99.9	90.16	100	13.27	86.14
	Coverage(%)	97.76	99.70	93.28	99.76	99.78	99.58	99.49	98.48

Table 3: The number and the size of phases in detection and prediction runs

Tests	Detection				Prediction			
	leaf phases	exe. len. (M inst.)	avg. leaf size (M inst.)	avg. largest phase size (M inst.)	leaf phases	exe. len. (M inst.)	avg. leaf size (M inst.)	avg. largest phase size (M inst.)
FFT	14	23.8	2.5	11.6	122	5730.4	50.0	232.2
Applu	645	254.3	0.394	3.29	4437	335019.8	75.5	644.8
Compress	52	52.0	0.667	2.2	52	62418.4	800.2	2712.0
Tomcatv	35	175.0	4.9	34.9	5250	24923.2	4.7	33.23
Swim	91	376.7	4.1	37.6	8101	33334.9	4.1	37.03
Mesh	4691	5151.9	1.1	98.2	4691	5151.9	1.1	98.2
MolDyn	59	11.9	0.202	3.97	569	50988.1	89.6	1699.6
Average	798	863.66	1.98	27.39	3317	73938.1	146.5	779.58

come a single line in the graph. The numbers next to the lines are the basic block IDs where markers are inserted. The same code block precedes and only precedes the same locality phase, showing the effect of marker selection.

The middle two graphs show the locality of predicted phases. To visualize the locality, we arbitrarily pick two different cache sizes—32KB and 256KB cache—and use the two miss rates as coordinates. Each execution of a phase is a cross (X) on the graph. *Tomcatv* has 5251 executions of 7 locality phases: all five thousand crosses are mapped to seven in the graph. Most crosses overlap perfectly. The phase prediction is correct in all cases because the executions of the same phase maps to a single cross except for a small difference in the second and third phase, where the first couple of executions have slightly different locality. We label each phase by the phase ID, the relative frequency, and the range of phase length. The relative frequency is the number of the executions of a phase divided by the total number of phase executions (5251 for *Tomcatv*). The last two numbers give the number of instructions in the shortest and the longest execution of the phase, in the unit of millions of instructions. *Compress* is shown by the same format. It has 52 executions of 4 locality phases: all 52 crosses map to four, showing perfect prediction accuracy. The phase length ranges from 2.9 thousand to 1.9 million instructions in two programs. For each phase, the length prediction is accurate to at least three significant digits.

The power of phase prediction is remarkable. For example, in *Compress*, when the first marker is executed for the second time, the program knows that it will execute 1.410 million instructions before reaching the next marker, and that the locality is the same for every execution. This accuracy confirms our assumption that locality phases are marked by abrupt changes in data reuse.

Phase vs. interval An interval method divides the execution into fixed-size intervals. The dots in the bottom graphs of Figure 3 show the locality of ten million instruction intervals. The 2493 dots in *Tomcatv* and 6242 dots in *Compress* do not suggest a regular pattern.

Both the phases and intervals are partitions of the same execution sequence—the 25 billion instructions in *Tomcatv* and 62 billion in *Compress*. Yet the graphs are a striking contrast between the sharp focus of phase crosses and the irregular spread of interval dots—it indeed matters where and how to partition an execution into phases. Locality phases are selected at the right place with the right length, while intervals are a uniform cut. Compared to the phases, the intervals are too large to capture the two to four million-instruction phases in *Tomcatv* and too small to find the over one billion-instruction phases in *Compress*. While the program behavior is highly regular and fully predictable for phases, it becomes mysteriously irregular once the execution is cut into intervals.

Phase vs. BBV A recent paper [10] examined three phase analysis techniques—procedure-based [21, 22], code working set [9], and basic-block vector (BBV) [30]. By testing the variation in IPC (instruction per cycle), it concluded that BBV is the most accurate. We implemented BBV prediction according to the algorithm of Sherwood et al [30]. Our implementation uses the same ten million-instruction windows and the same threshold for clustering. We implemented their Markov predictor but in this section we use only the clustering (perfect prediction). It randomly projected the frequency of all code blocks into a 32-element vector before clustering. Instead of using IPC, we use locality as the metric for evaluation.

BBV clusters the intervals based on their code signature and execution frequency. We show each BBV cluster by a bounding box labeled with the relative frequency. BBV analysis produces more clusters than those shown. We do not show boxes for clusters whose frequency is less than 2.1%, partly to make the graph readable. We note that the aggregated size of the small clusters is quite large (51%) for *Tomcatv*. In addition, we exclude the outliers, which are points that are farthest from the cluster center (3δ, statistically speaking); otherwise the bounding boxes are larger.

As shown by previous studies [10, 30], BBV groups intervals that have similar behavior. In *Tomcatv*, the largest cluster accounts for 26% of the execution. The miss rate varies by less than 0.3% for the 256KB cache and 0.5% for the 32KB cache. However, the similarity is not guaranteed. In the worst case in *Compress*, a cluster of over 23% execution has a miss rate ranging from 2.5% to 5.5% for the 256KB cache and from 7% to 11% for the 32KB cache. In addition, different BBV clusters may partially intersect. Note that with fine-tuned parameters we will see smaller clusters with lower

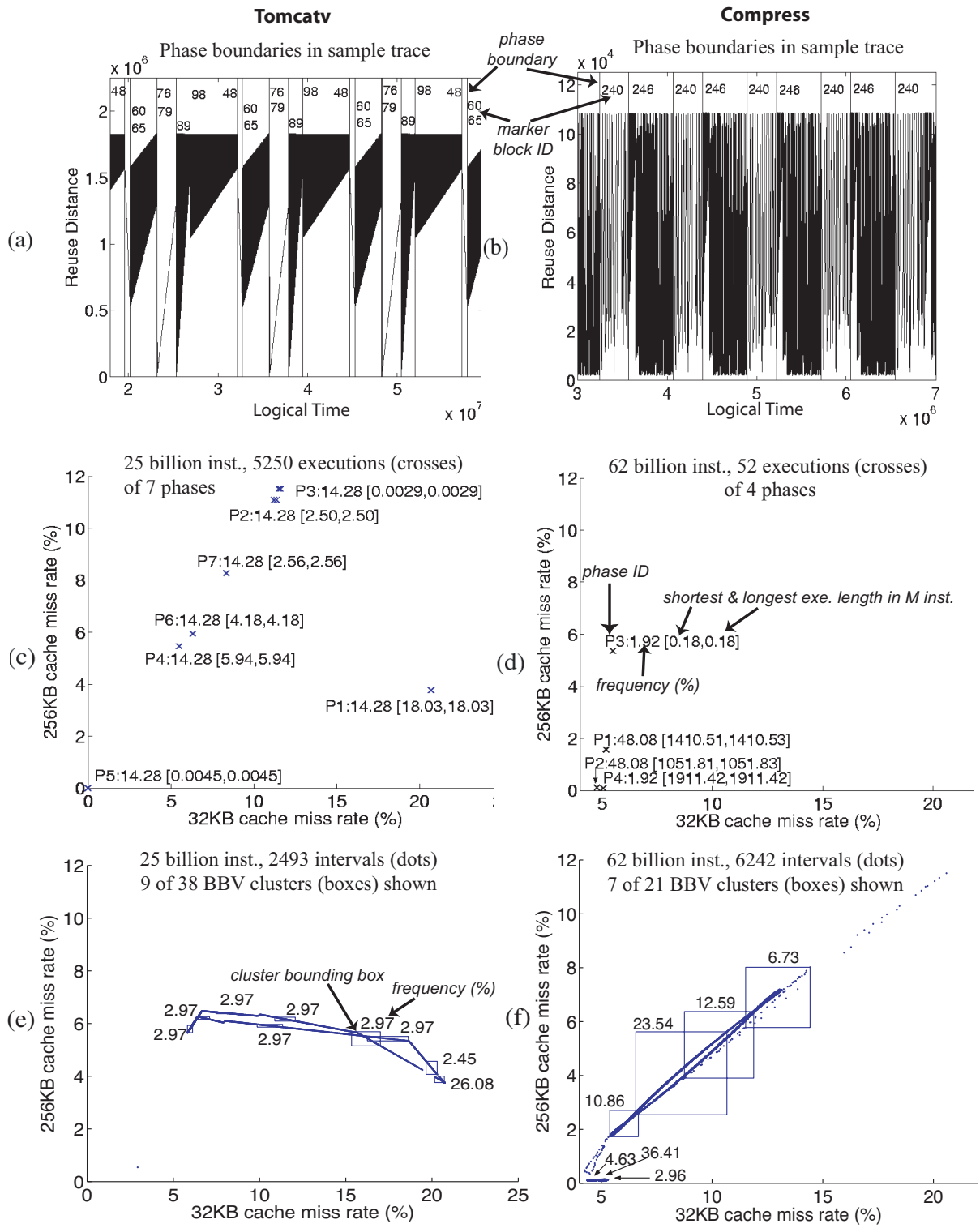

Figure 3: Prediction Accuracy for *Tomcatv* and *Compress*. Part (a) and (b) show the phase boundaries found by off-line phase detection. Part (c) and (d) show the locality of the phases found by run-time prediction. As a comparison, Part (e) and (f) show the locality of ten million-instruction intervals and BBV (basic-block vector) clusters.

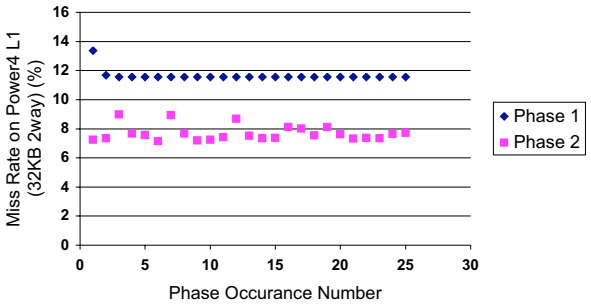

Figure 4: The miss rates of *Compress* phases on IBM Power 4

variation. In fact, in the majority of cases in these programs, BBV produces tight clusters. However, even in best cases, BBV clusters do no have perfectly stacked points as locality phases do.

Table 4 shows the initial and normalized standard deviation. The locality is an 8-element vector that contains the miss rate for cache sizes from 32KB to 256KB in 32KB increments. The standard deviation is calculated for all executions of the same phase and the intervals of each BBV cluster. Then the standard deviation of all phases or clusters are averaged (weighted by the phase or cluster size) to produce the number for the program. The numbers of BBV clustering and prediction, shown by the last two columns, are similarly small as reported by Sherwood et al. for IPC [30]. Still, the numbers for locality phases are much smaller—one to five orders of magnitude smaller than that of BBV-based prediction.

Table 4: Standard deviation of locality phases and BBV phases

	standard deviations		
	locality phase prediction	BBV clustering	BBV RLE Markov prediction
FFT	6.87E-8	0.00040	0.0061
Applu	5.06E-7	2.30E-5	0.00013
Compress	3.14E-6	0.00021	0.00061
Tomcatv	4.53E-7	0.00028	0.0016
Swim	2.66E-8	5.59E-5	0.00018
Mesh	6.00E-6	0.00012	0.00063
MolDyn	7.60E-5	0.00040	0.00067

So far we measure the cache miss rate through simulation, which does not include all factors on real machines such as that of the operating system. We now examine the L1 miss rate on an IBM Power 4 processor for the first two phases of *Compress* (the other two phases are too infrequent to be interesting). Figure 4 shows the measured miss rate for each execution of the two phases. All but the first execution of Phase 1 have nearly identical miss rates on the 32KB 2-way data cache. The executions of Phase 2 show more variation. The effect from the environment is more visible in Phase 2 likely because its executions are shorter and the miss rate lower than those of the first phase.

The comparison with interval-based methods is partial because we use only programs that are amenable to locality-phase prediction. Many dynamic programs do not have consistent locality. For them interval-based methods can still exploit run-time patterns, while our current phase prediction scheme would not work because it assumes that each phase, once in execution, maintains identical locality. Next are two such examples.

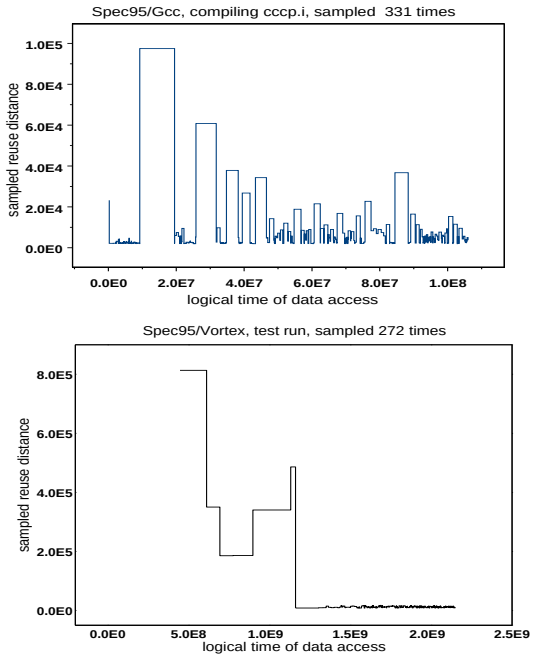

Figure 5: Sampled reuse distance trace of *Gcc* and *Vortex*. The exact phase length is unpredictable in general.

3.1.2 *Gcc* and *Vortex*

The programs *Gcc* and *Vortex* are different because their phase length is not consistent even in the same execution. In *Gcc*, the phase length is determined by the function being compiled. Figure 5 shows the distance-based sample trace. Unlike previous trace graphs, it uses horizontal steps to link sample points. The peaks in the upper graph roughly correspond to the 100 functions in the 6383-line input file. The size and location of the peaks are determined by the input and are not constant.

Vortex is an object-oriented database. The test run first constructs a database and then performs a set of queries. The lower figure of Figure 5 shows the sample trace. It shows the transition from data insertion to query processing. However, in other inputs, the construction and queries may come in any order. The exact behavior, like *Gcc*, is input dependent and not constant.

Our recent results show that an extension of the phase analysis can mark the phases in *Gcc*, which are the compilation of input functions. Still, the prediction results are input dependent. Ding and Zhong showed that the overall reuse pattern of these two programs are stable across many inputs [12]. It suggests a prediction strategy based on statistics. We do not consider this extension in this paper and will not discuss these two programs further.

3.2 Adaptive Cache Resizing

During an execution, cache resizing reduces the physical cache size without increasing the miss rate [2, 21]. Therefore, it can reduce the access time and energy consumption of the cache without losing performance. We use a simplified model where the cache consists of 64-byte blocks and 512 sets. It can change from direct mapped to 8-way set associative, so the cache size can change be-

tween 32KB and 256KB in 32KB units. In the adaptation, we need to predict the smallest cache size that yields the same miss rate as the 256KB cache.

As seen in the example of *Tomcatv*, program data behavior changes constantly. A locality phase is a unit of repeating behavior rather than a unit of uniform behavior. To capture the changing behavior inside a large phase, we divide it into 10K intervals (called phase intervals). The adaptation finds the best cache size for each interval during the first few executions and reuses them for later runs. The scheme needs hardware support but needs no more than that of interval-based cache resizing.

Interval-based cache resizing divides an execution into fixed-length windows. Based on the history, interval prediction classifies past intervals into classes and predicts the behavior class of the next interval using methods such as last-value and Markov models. For cache resizing, Balasubramonian et al. used the cache miss rate and branch prediction rate to classify past intervals [2]. Recent studies considered code information such as code working set [9] and basic-block vector (BBV) [30]. We test interval-based prediction using five different interval lengths: 10K, 1M, 10M, 40M, and 100M memory accesses. In addition, we test a BBV predictor using 10M instruction windows, following the implementation of Sherwood et al [30].

The interval methods do not have phase markers, so they constantly monitor every past interval to decide whether a phase change has occurred. In the experiment, we assume perfect detection: there is a phase change if the best cache size of the next interval differs from the current one. BBV method uses a run-length encoding Markov predictor to give the BBV cluster of the next interval (the best predictor reported in [30]). However, as the last section shows, the intervals of a BBV cluster do not always have identical locality. We use perfect detection for BBV as we do for other interval methods.

At a phase change, the phase method reuses the best cache size stored for the same phase. The BBV method reuses the current best cache size for each BBV cluster. For run-time exploration, we count the minimal cost—each exploration takes exactly two trial runs, one at the full cache size and one at the half cache size. Then we use the best cache size from the third interval. In the experiment, we know the best cache size of each phase or interval by running it through *Cheetah*, a cache simulator that measures the miss rate of all eight cache sizes at the same time [33]. The results for interval and BBV methods are idealistic because they use perfect phase-change detection. The result of the phase-interval method is real. Because it knows the exact behavior repetition, the phase-interval method can amortize the exploration cost over many executions. With the right hardware support, it can gauge the exact loss compared to the full size cache and guarantee a bound on the absolute performance loss.

Figure 6 shows the average cache size from phase, interval, and BBV methods. The first graph shows the results of adaptation with no miss-rate increase. The results are normalized to the phase method. The largest cache size, 256KB, is shown as the last bar in each group. Different intervals find different cache sizes, but all reductions are less than 10%. The average is 6%. BBV gives consistently good reduction with a single interval size. The improvement is at most 15% and on average 10%. In contrast, the phase adaptation reduces the cache size by 50% for most programs and over 35% on average.

Figure 6(b) shows the results of adaptation with a 5% bound on the miss-rate increase. The effect of interval methods varies greatly. The 10M interval was 20% better than the locality phase for *FFT* but a factor of three worse for *Tomcatv* and *Swim*. The 100M interval has the best average reduction of nearly 50%. BBV again shows consistently good reduction with a single interval size. On average it is slightly better than the best interval method. The phase method reduces the cache size more than other methods do for all programs except for *FFT*. *FFT* has varied behavior, which causes the low coverage and consequently not as large cache-size reduction by locality phase prediction. *MolDyn* does not have identical locality, so phase-based resizing causes a 0.6% increase in the number of cache misses. Across all programs, the average reduction using locality phases is over 60%.

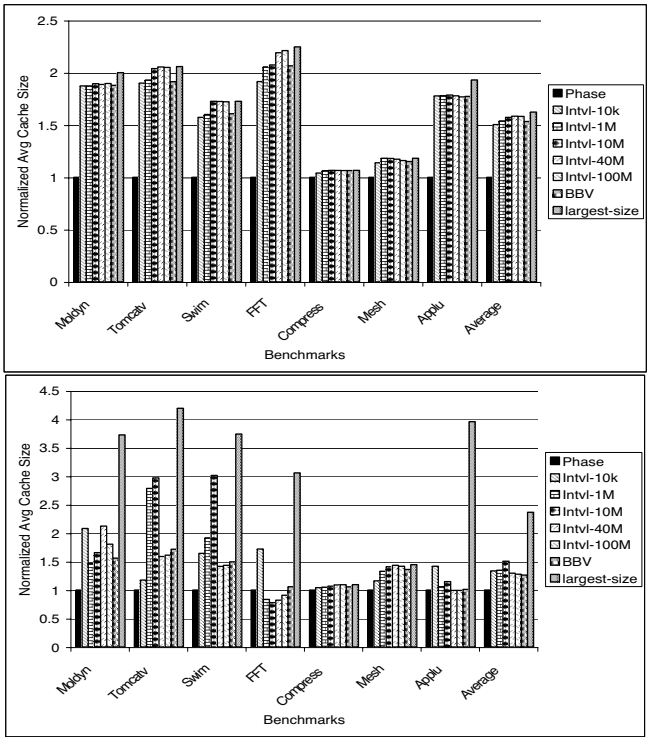

Figure 6: Average cache-size reduction by locality phase, interval, and BBV prediction methods, assuming perfect phase-change detection and minimal-exploration cost for interval and BBV methods. Upper graph: no increase in cache misses. Lower graph: at most 5% increase.

Earlier studies used more accurate models of cache and measured the effect on time and energy through cycle-accurate simulation. Since simulating the full execution takes a long time, past studies either used a partial trace or reduced the program input size [2, 21]. We choose to measure the miss rate of full executions. While it does not give the time or energy, the miss rate is accurate and reproducible by others without significant efforts in calibration of simulation parameters.

3.3 Phase-Based Memory Remapping

We use locality phases in run-time memory remapping. To support data remapping at the phase boundary, we assume the support of the *Impulse* memory controller, developed by Carter and his colleagues at University of Utah [34, 35]. *Impulse* reorganizes data without actually copying them to CPU or in memory. For example, it may create a column-major version of a row-major array via remapping. A key requirement for exploiting *Impulse* is to identify the time when remapping is profitable.

We consider affinity-based array remapping, where arrays that tend to be accessed concurrently are interleaved by remapping [36]. To demonstrate the value of locality phase prediction, we evaluate the performance benefits of redoing the remapping for each phase rather than once for the whole program during compilation. We apply affinity analysis for each phase and insert remapping code at the location of the phase marker. The following table shows the execution time in seconds on 2GHz Intel Pentium IV machine with the *gcc* compiler using *-O3*.

For the two programs, we obtain speedups of 35.5% and 2.8% compared to the original program and 13% and 2.5% compared to the best static data layout [36], as shown in Table 5. In the absence of an *Impulse* implementation, we program the remapping and calculate the running time excluding the remapping cost. Table 7.3 of Zhang's dissertation shows the overhead of setting up remappings for a wide range of programs. The overhead includes setting up shadow region, creating memory controller page table, data flushing, and possible data movement. The largest overhead shown is 1.6% of execution time for static index vector remapping [34].

For example for the 14 major arrays in *Swim*, whole-program analysis shows close affinity between array *u* and *v*, *uold* and *pold*, and *unew* and *pnew*. Phase-based analysis shows affinity group {*u,v,p*} for the first phase, {*u,v,p,unew,vnew,pnew*} for the second phase, and three other groups, {*u,uold,unew*}, {*v,vold,vnew*}, and {*p,pold,pnew*}, for the third phase. Compared to whole-program reorganization, the phase-based optimization reduces cache misses by one third (due to array *p*) for the first phase, by two thirds for the second phase, and by half for the third phase.

Using the two example programs, we have shown that phase prediction finds opportunities of dynamic data remapping. The additional issues of affinity analysis and code transformation are discussed by Zhong et al [36]. The exact interaction with *Impulse* like tools is a subject of future study.

3.4 Comparison with Manual Phase Marking

We hand-analyzed each program and inserted phase markers (*manual markers*) based on our reading of the code and its documentation as well as results from *gprof* (to find important functions). We compare manual marking with automatic marking as follows. As a program runs, all markers output the logical time (the number of memory accesses from the beginning). Given the set of logical times from manual markers and the set from auto-markers, we measure the overlap between the two sets. Two logical times are considered the same if they differ by no more than 400, which is 0.02% of the average phase length. We use the recall and precision to measure their closeness. They are defined by the formulas below. The recall shows the percentage of the manually marked times that are marked by auto-markers. The precision shows the percentage of the automatically marked times that are marked manually.

$$Recall = \frac{|\ M \cap A\ |}{|\ M\ |} \quad (1)$$

$$Precision = \frac{|\ M \cap A\ |}{|\ A\ |} \quad (2)$$

Table 5: The effect of phase-based array regrouping, excluding the cost of run-time data reorganization

Benchmark	Original	Phase (speedup)	Global (speedup)
Mesh	4.29	4.17 (2.8%)	4.27 (0.4%)
Swim	52.84	34.08 (35.5%)	38.52(27.1%)

Table 6: The overlap with manual phase markers

Benchmark	Detection		Prediction	
	Recall	Prec.	Recall	Prec.
FFT	1	1	1	1
Applu	0.993	0.941	0.999	0.948
Compress	0.987	0.962	0.987	0.962
Tomcatv	0.952	0.556	1	0.571
Swim	1	0.341	1	0.333
Mesh	1	0.834	1	0.834
MolDyn	0.889	0.271	0.987	0.267
Average	0.964	0.690	0.986	0.692

where M is the set of times from the manual markers, and A is the set of times from auto-markers.

Table 6 shows a comparison with manually inserted markers for detection and prediction runs. The columns for each run give the recall and the precision. The recall is over 95% in all cases except for *MolDyn* in the detection run. The average recall increases from 96% in the detection run to 99% in the prediction run because the phases with a better recall occur more often in longer runs. Hence, the auto-markers capture the programmer's understanding of the program because they catch nearly all manually marked phase changing points.

The precision is over 95% for *Applu* and *Compress*, showing that automatic markers are effectively the same as the manual markers. *MolDyn* has the lowest recall of 27%. We checked the code and found the difference. When the program is constructing the neighbor list, the analysis marks the neighbor search for each particle as a phase while the programmer marks the searches for all particles as a phase. In this case, the analysis is correct. The neighbor search repeats for each particle. This also explains why *Moldyn* cannot be predicted with both high accuracy and high coverage—the neighbor search has varying behavior since a particle may have a different number of neighbors. The low recall in other programs has the same reason: the automatic analysis is more thorough than the manual analysis.

Four of the test programs are the simulation of grid, mesh and N-body systems in time steps. Ding and Kennedy showed that they benefited from dynamic data packing, which monitored the runtime access pattern and reorganized the data layout multiple times during an execution [11]. Their technique was automatic except for a programmer-inserted directive, which must be executed once in each time step. This work was started in part to automatically insert the directive. It has achieved this goal: the largest composite phase in these four programs is the time step loop. Therefore, the phase prediction should help to fully automate dynamic data packing, which is shown by several recent studies to improve performance by integer factors for physical, engineering, and biological simulation and sparse matrix solvers [11, 15, 25, 32].

Summary For programs with consistent phase behavior, the new method gives accurate locality prediction and consequently yields significant benefits for cache resizing and memory remapping. It is more effective at finding long, recurring phases than previous methods based on program code, execution intervals, their combination, and even manual analysis. For programs with varying phase behavior, the profiling step can often reveal the inconsistency. Then the method avoids behavior prediction of inconsistent phases through a flag (as shown by the experiments reported in Table 2). Using a small input in a profiling run is enough for locality phase prediction. Therefore, the technique can handle large programs and long executions. For programs such as *GCC* and *Vortex*, where little con-

sistency exists during the same execution, the locality analysis can still recognize phase boundaries but cannot yet make predictions. Predictions based on statistics may be helpful for these programs, which remains to be our future work. In addition, the current analysis considers only temporal locality. The future work will consider spatial locality in conjunction with temporal locality.

4. RELATED WORK

This work is a unique combination of program code and data analysis. It builds on past work in these two areas and complements interval-based methods.

Locality phases Early phase analysis, owing to its root in virtual-memory management, was intertwined with locality analysis. In 1976, Batson and Madison defined a phase as a period of execution accessing a subset of program data [4]. They showed experimentally that a set of Algol-60 programs spent 90% time in major phases. However, they did not predict locality phases. Later studies used time or reuse distance as well as predictors such as Markov models to improve virtual memory management. Recently, Ding and Zhong found predictable patterns in the overall locality but did not consider the phase behavior [12]. We are not aware of any trace-based technique that identifies static phases using locality analysis.

Program phases Allen and Cocke pioneered interval analysis to convert program control flow into a hierarchy of regions [1]. For scientific programs, most computation and data access are in loop nests. A number of studies showed that the inter-procedural array-section analysis accurately summarizes the program data behavior. Recent work by Hsu and Kremer used program regions to control processor voltages to save energy. Their region may span loops and functions and is guaranteed to be an atomic unit of execution under all program inputs [17]. For general purpose programs, Balasubramonian et al. [2], Huang et al. [21], and Magklis et al. [22] selected as phases procedures whose number of instructions exceeds a threshold in a profiling run. The three studies found the best voltage for program regions on a training input and then tested the program on another input. They observed that different inputs did not affect the voltage setting. The first two studies also measured the energy saving of phase-based cache resizing [2, 21]. In comparison, the new technique does not rely on static program structure. It uses trace-based locality analysis to find the phase boundaries, which may occur anywhere and not just at region, loop or procedure boundaries.

Interval phases Interval methods 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 [9, 10, 13, 30]. The past work used intervals of length from 100 thousand [2] to 10 million instructions [30] and executions from 10 milliseconds to 10 seconds [13]. Interval prediction works well if the interval length does not matter, for example, when an execution consists of long steady phases. Otherwise it is difficult to find the best interval length for a given program on a given input. The experimental data in this paper show the inherent limitation of intervals for programs with constantly changing data behavior. Balasubramonian et al. searches for the best interval size at run time [3]. Their method doubles the interval length until the behavior is stable. Let N be the execution length, this new scheme searches $O(logN)$ choices in the space of N candidates. In this work, we locate phases and determine their exact lengths through off-line locality analysis. We show that important classes of programs have consistent phase behavior and the high accuracy and large granularity of phase prediction allow adaptation with a tight worst-performance guarantee. However, not all programs are amenable to the off-line analysis. Interval-based methods do not have this limitation and can exploit the general class of run-time patterns.

5. CONCLUSIONS

The paper presents a general method for predicting hierarchical memory phases in programs with input-dependent but consistent phase behavior. Based on profiling runs, it predicts program executions hundreds of times larger and predicts the length and locality with near perfect accuracy. When used for cache adaptation, it reduces the cache size by 40% without increasing the number of cache misses. When used for memory remapping, it improves program performance by up to 35%. It is more effective at identifying long, recurring phases than previous methods based on program code, execution intervals, and manual analysis. It recognizes programs with inconsistent phase behavior and avoids false predictions. These results suggest that locality phase prediction should benefit modern adaptation techniques for increasing performance, reducing energy, and other improvements to the computer system design.

Scientifically speaking, this work is another attempt to understand the dichotomy between program code and data access and to bridge the division between off-line analysis and on-line prediction. The result embodies and extends the decades-old idea that locality could be part of the missing link.

Acknowledgement

This work was motivated in part by the earlier work on phase analysis by our colleagues at Rochester. The presentation was improved by discussions with Sandhya Dwarkadas, Michael Huang, Michael Scott, Dave Albonesi, and Kai Shen at Rochester, John Carter at Utah, Uli Kremer at Rutgers, Brad Calder at UCSD, and the anonymous reviewers from this and another conference committee. Lixin Zhang at IBM and Tim Sherwood at UCSB answered our questions about *Impulse* and *BBV*. The financial support comes in part from the Department of Energy (Contract No. DE-FG02-02ER25525) and the National Science Foundation (Contract No. CCR-0238176, CCR-0219848, and EIA-0080124).

6. REFERENCES

[1] F. Allen and J. Cocke. A proram data flow analysis procedure. *Communications of the ACM*, 19:137–147, 1976.

[2] 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.

[3] 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.

[4] A. P. Batson and A. W. Madison. Measurements of major locality phases in symbolic reference strings. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, Cambridge, MA, March 1976.

[5] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[6] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on

distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[7] I. Daubechies. *Ten Lectures on Wavelets*. Capital City Press, Montpelier,Vermont, 1992.

[8] P.J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), January 1980.

[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 K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[12] 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.

[13] 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.

[14] 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.

[15] H. Han and C. W. Tseng. Improving locality for adaptive irregular scientific codes. In *Proceedings of Workshop on Languages and Compilers for High-Performance Computing (LCPC'00)*, White Plains, NY, August 2000.

[16] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[17] 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.

[18] R. Joseph, Z. Hu, and M. Martonosi. Wavelet analysis for microprocessor design: Experiences with wavelet-based di/dt characterization. In *Proceedings of International Symposium on High Performance Computer Architecture*, February 2004.

[19] J. R. Larus. Whole program paths. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[20] C. Luk and T. C. Mowry. Memory forwarding: enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *Proceedings of International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[21] 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.

[22] 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.

[23] 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.

[24] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[25] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. *International Journal of Parallel Programming*, 29(3), June 2001.

[26] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: a linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[27] V. S. Pingali, S. A. McKee, W. C. Hsieh, and J. B. Carter. Restructuring computations for temporal data cache locality. *International Journal of Parallel Programming*, 31(4), August 2003.

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

[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] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.

[32] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[33] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, Santa Clara, CA, May 1993.

[34] L. Zhang. *Efficient Remapping Mechanism for an Adaptive Memory System*. PhD thesis, Department of Computer Science, University of Utah, August 2000.

[35] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The Impulse memory controller. *IEEE Transactions on Computers*, 50(11), November 2001.

[36] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.