Provided for non-commercial research and educational use only.
Not for reproduction or distribution or commercial use.

This article was originally published in a journal published by Elsevier, and the attached copy is provided by Elsevier for the author's benefit and for the benefit of the author's institution, for non-commercial research and educational use including without limitation use in instruction at your institution, sending it to specific colleagues that you know, and providing a copy to your institution's administrator.

All other uses, reproduction and distribution, including without limitation commercial reprints, selling or licensing copies or access, or posting on open internet sites, your personal or institution's website or repository, are prohibited. For exceptions, permission may be sought for such use through Elsevier's permissions site at:

http://www.elsevier.com/locate/permissionusematerial

Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 783–796
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Author's personal copy

Predicting locality phases for dynamic memory optimization[☆]

Xipeng Shen[a], Yutao Zhong[b], Chen Ding[c,*]

[a]Department of Computer Science, The College of William and Mary, Williamsburg, VA, USA
[b]Department of Computer Science, George Mason University, Fairfax, VA, USA
[c]Department of Computer Science, University of Rochester, P.O. Box 270226, Rochester, NY 14627, USA

Received 6 August 2005; received in revised form 12 December 2006; accepted 18 January 2007
Available online 19 March 2007

Abstract

Dynamic data, cache, and memory adaptation can significantly improve program performance when they are applied on long continuous phases of execution that have dynamic but predictable locality. To support phase-based adaptation, this paper defines the concept of locality phases and describes a four-component analysis technique. *Locality-based phase detection* uses locality analysis and signal processing techniques to identify phases from the data access trace of a program; *frequency-based phase marking* inserts code markers that mark phases in all executions of the program; *phase hierarchy construction* identifies the structure of multiple phases; and *phase-sequence prediction* predicts the phase sequence from program input parameters. The paper shows the accuracy and the granularity of phase and phase-sequence prediction as well as its uses in dynamic data packing, memory remapping, and cache resizing.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Program phase prediction; Phase hierarchy; Locality analysis and optimization; Reconfigurable architecture; Dynamic optimization

1. Introduction

Over the past 30 years, memory performance increasingly determines the program performance on high-end machines. Although programs employ a large amount of data, they do not use all data at all times. Run-time adaptation has been extensively studied, which may reorganize the entire program data multiple times during an execution [16,21,34,43,44,47].

The adaptive techniques divide programs into phases that satisfy two requirements. First, the phase behavior must recur, so the run-time analysis can determine the new data layout based on the past behavior. Second, the phase must have a large size, so the benefit of data reorganization outweighs the high cost of large-scale data movement. All previous techniques used manual analysis to find large, recurring phases. However, manual analysis limited the size and the number of programs that could benefit from run-time adaptation. In this paper, we show that the analysis can be completely automated through locality phase prediction.

We define a *program locality phase* by the following three conditions. First, a phase is a unit of predictable behavior. Second, a phase is marked by some statements in the program. We call them *phase markers*. If the control flow reaches a beginning or ending phase marker, the phase must begin or end. We call the period of execution between a consecutive pair of beginning and ending markers *a phase instance*. Finally, instances of different phases are either disjoint or nested, so they cannot partially overlap. The properties of a phase make it possible for run-time data adaptation. Whenever a phase is entered, the program can reorganize the data based on the predicted behavior of the phase. Nested phases give different granularity of prediction and allow for adaptation with a different level of aggressiveness.

Locality phase prediction has four components. The first is *locality-based phase detection*, which analyzes the memory access trace in profiling runs to identify locality phases. By examining the distance of data reuses, it can “zoom in” and “zoom out” over long execution traces. It uses a signal processing technique, wavelet analysis, to identify phase changes. Then the second component, *frequency-based*

[☆] This paper extends the earlier reports published at the 2004 ASPLOS Conference and the 2004 LCPC workshop.
[*] Corresponding author. Fax: +1 585 273 4556.
E-mail addresses: xshen@cs.wm.edu (X. Shen), yzhong@cs.gmu.edu (Y. Zhong), cding@cs.rochester.edu (C. Ding).

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.01.010

phase marking finds the phase boundaries and inserts phase markers into the program code. It operates at the binary level and does not need access to the source code. Since phases often repeat in groups, the third component, *phase hierarchy construction*, captures the composite patterns as a parameterized regular expression. Finally, *phase-sequence prediction* predicts the phase sequence from the input parameters of a program. During an execution, a run-time system monitors the first few instances of a phase and uses the result to predict the behavior of the later instances. Together, the four components combine information from the program code, data, and machine environment.

The new technique targets programs that have recurring phases. A major class is scientific or commercial simulation. Take for example, a program that tests the aging of an airplane model by repeatedly sweeping through the mesh structure of the model in many time steps. The behavior in each time step is similar because most operations are the same despite some local variations in the control flow. Given a different input, for example another airplane model, the behavior of the new simulation may change radically but will remain consistent between the new time steps. Similar phase behavior is common in structural, mechanical, molecular, and other scientific, engineering, and commercial simulations. These programs have great demand for computing resources and would benefit from adaptive data and memory transformations if we can predict their dynamic locality patterns.

Through an empirical evaluation, we show that the new analysis finds recurring phases of widely varying sizes and nearly identical behavior. While the locality and the length of phases change in tune with program inputs, the behavior can be predicted with high accuracy at run time. We show the use of the prediction in dynamic data packing and memory remapping at the program level and cache resizing at the architecture level. We show that for several programs, the new method predicts the complete phase sequence from the program input. Finally, the phase hierarchy enables correlation analysis of phase instances from multiple runs, which we use to predict phase locality for all program inputs.

Phase prediction is not effective on all programs. Some programs may not have predictable behavior, let alone by our method. We limit our analysis to programs that have large predictable phases. As dependence analysis analyzes loops that can be analyzed, we predict patterns that are amenable to our analysis. We now show that the phase detection, marking, and behavior prediction lead to better understanding of important classes of dynamic programs in scientific and commercial computing.

2. Phase detection, marking, and behavior prediction

This section first presents locality-based phase detection, which models a program trace as a signal and the phase analysis as a search for a handful of phase boundaries in a large program trace. The section then describes the other three components: frequency-based phase marking, phase hierarchy construction, and phase-sequence prediction.

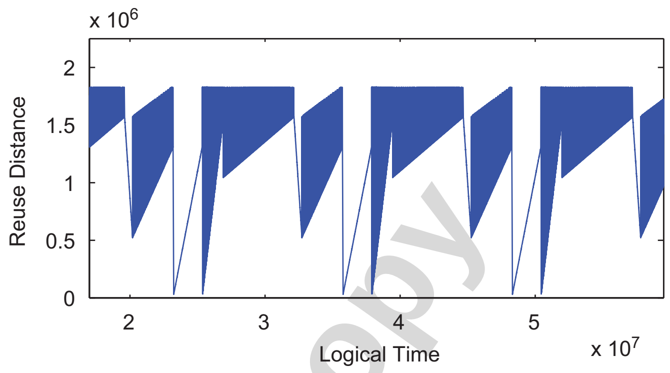


Fig. 1. The reuse distance trace of *Tomcatv* from SPEC95.

2.1. Locality-based phase detection

2.1.1. Reuse distance as a signal

Reuse distance is the same as *LRU-stack distance*, first defined by Mattson et al. in 1970. For each access on a data access trace, the reuse distance is the number of distinct data elements accessed between this and the previous access to the same datum [33]. In a recent study, Ding and Zhong showed that reuse distance can be approximated with any fixed precision in near linear time [17].

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

The reuse distance changes continuously throughout the trace. Most are gradual changes but they are separated by disruptive transitions. We consider the disruptive transitions as points of a phase change, a radical change in the data reuse pattern. In this example, the radical changes divide the trace into clearly separated phase instances. A group of five instance types repeat in sequence. Reading the code documentation, we see indeed that the program has a sequence of time steps, each has five sub-steps—preparing data, computing residual values, solving two tridiagonal 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 code. It should be noted that manual phase marking is often effective especially for programs with clear loop and function structures. However, the difficulties increase for programs with many loops and subroutines and with dynamic bindings which are increasingly common due to object-oriented design. Moreover, at least in principle, a phase change may happen on a branch in a subroutine and elude loop and function-based analysis.

The locality data suggest four properties of phase behavior. First, the behavior may change continuously in an execution;

[1] To reduce the size of the graph, we show the reuse distance trace after variable-distance sampling described in Section 2.1.2.

Author's personal copy

however, major shifts are marked by radical rather than gradual changes. Second, phases have different lengths. The length of one phase has little relation with that of others. Third, the length changes greatly with program inputs. For example, a phase instance of *Tomcatv* contains a few hundred million memory accesses in a training run but over 25 billion memory accesses in a test run. Finally, instances of the same phase have a similar behavior. *A phase is a unit of repeating behavior rather than a unit of uniform behavior.*

Reuse distance represents the memory behavior as a signal, which may capture locality in cases where pure program or hardware measures are not adequate. 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, so a miss rate curve depends on the window size. Single cache hit or miss is a binary event and depends on the hardware setup, while each reuse distance is a precise scale independent of hardware.

2.1.2. Variable-distance sampling

Instead of analyzing all accesses to all memory locations, we sample a small number of representative memory locations. In addition, for each location, we record only long-distance reuses because they reveal long-range patterns. Variable-distance sampling is based on distance-based sampling [17], which monitors the reuse distance of every access. When the reuse distance is above a threshold (the *qualification threshold* h_q), 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 *distance threshold* h_d). To reduce redundancy, it requires that a new data sample be at least a certain spatial distance away (the *spatial threshold* h_s) in memory from existing location samples.

With the qualification and distance thresholds, we effectively analyze only cache misses (on fully associative cache) as one would with a cache simulator. The difference between the two can be viewed that we monitor a data element if it misses in level-two cache but when we monitor, we observe its misses in level-one cache. In addition, it takes a large piece of data to cause long-distance reuses, for example, in the traversal of an array. Not knowing the data structure of a program, we use the spatial threshold to avoid monitoring the entire array.

Fixed thresholds may be too sensitive or too insensitive. If they collect too much data, the later analysis would take too long. If they do not collect enough data, the later analysis may not detect any phase. An inherent limitation of sampling is that it cannot guarantee to work for all programs on all inputs. In this work we use a resource-based strategy we call *variable-distance sampling*.

We first add constraints as $h_d = \frac{h_q}{4}$ and $h_s = \frac{h_q}{10}$ to reduce three thresholds into one h_q. Given a program, let T be the length of execution in the number of total memory references (measured without reuse distance analysis) and S_t be the target number of samples. The sampling method uses an iterative search on the first T' references to determine h_q. It sets $h_q = 20\,000$ at the beginning. In each iteration, we count the number of samples S' in T' and estimate the total number of samples to be $S_r = S' * T/T'$. If the estimate is within the range $[0.2 * S_t, 5 * S_t]$, the search terminates. If the number of samples is too large (because h_q is too small), it sets $Lower\ Bound = h_q$. If the number of samples is too small, it sets $Upper\ Bound = h_q$. The value for h_q of the next iteration is then $\frac{Lower\ Bound + Upper\ Bound}{2}$ if both bounds are defined. Otherwise, h_q is changed by the ratio S_r/S_t. In our implementation, S_t is 20 055 and T' is 80 million. The method finds the appropriate thresholds for all our test programs in less than 20 iterations.

The algorithm assumes that for the same h_q, the sampling rate of the whole execution is not too different from that of the first T' memory accesses, which is the case in our test programs. It takes several hours for the later steps of wavelet filtering and optimal phase partitioning to analyze the tens of thousands of samples, although the long time is acceptable for our offline analysis and can be improved by a more efficient implementation (currently written in Matlab and Java). The samples may include partial results for the executions that have uneven reuse density. However, the target sample size is large, so it is likely to have enough samples from all major phases. If a data sample has too few access samples to be useful, the next step will remove them as noise.

2.1.3. Wavelet filtering

Treating the sampled reuse distance trace as a signal, we use the *Discrete Wavelet Transform* (*DWT*) as a filter to find abrupt changes in the reuse pattern. DWT is a common technique used in signal processing to show the change of frequency over time [14]. DWT applies two functions to data: the scale function and the wavelet function. The first smooths the signal by averaging its values over 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 using the scaled results from the last level instead of 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 some variants of the following formulas:

$$c_j(k) = \langle f(x), 2^{-j}\phi(2^{-j}x - k)\rangle, \tag{1}$$

$$w_j(k) = \langle f(x), 2^{-j}\psi(2^{-j}x - k)\rangle, \tag{2}$$

where $\langle a, b\rangle$ is the scalar product of a and b, $f(x)$ is the input signal, j is the analysis level, and ϕ and ψ are the scaling and the wavelet functions, respectively. Many wavelet families exist in the literature, for example, *Haar, Daubechies*, and *Mexican-hat*. We use *Daubechies-6* in our experiments, but other families we tested produced a similar result. At high-resolution levels, the points with high wavelet coefficient values signal abrupt changes; therefore they are likely phase changing points.

The data access trace is not a single signal but a mix of many signals, each consisting of the access sub-trace on a memory location. Separating different signals is critical because a gradual change in the sub-trace may be seen as an abrupt change in

Author's personal copy

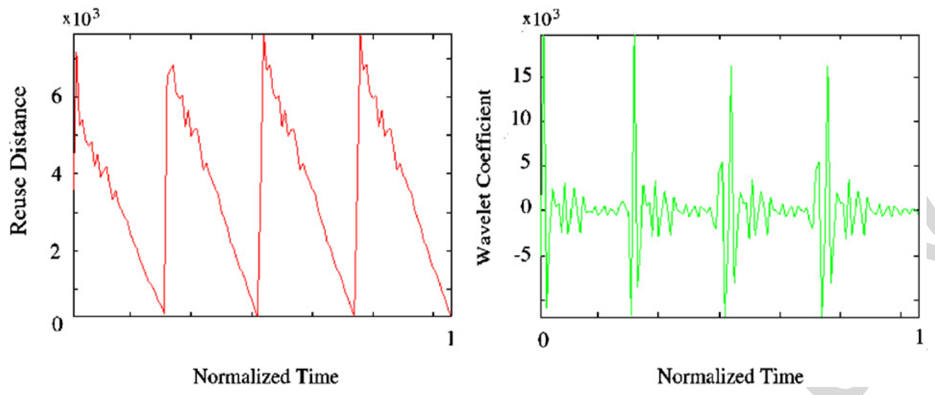
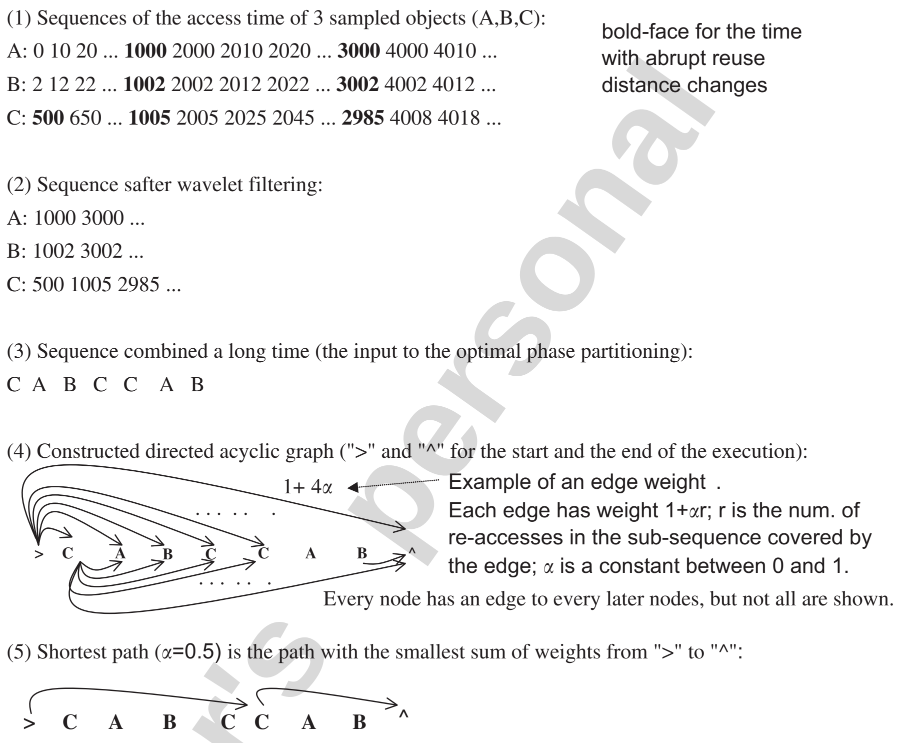


Fig. 2. A wavelet transform example, where gradual changes are filtered out.



(1) Sequences of the access time of 3 sampled objects (A,B,C):
A: 0 10 20 ... **1000** 2000 2010 2020 ... **3000** 4000 4010 ...
B: 2 12 22 ... **1002** 2002 2012 2022 ... **3002** 4002 4012 ...
C: **500** 650 ... **1005** 2005 2025 2045 ... **2985** 4008 4018 ...
bold-face for the time with abrupt reuse distance changes
(2) Sequence safter wavelet filtering:
A: 1000 3000 ...
B: 1002 3002 ...
C: 500 1005 2985 ...
(3) Sequence combined a long time (the input to the optimal phase partitioning):
C A B C C A B
(4) Constructed directed acyclic graph (">" and "^" for the start and the end of the execution):
1+ 4α
Example of an edge weight .
Each edge has weight 1+αr; r is the num. of re-accesses in the sub-sequence covered by the edge; α is a constant between 0 and 1.
> C A B C C A B ^
Every node has an edge to every later nodes, but not all are shown.
(5) Shortest path (α=0.5) is the path with the smallest sum of weights from ">" to "^":
> C A B C C A B ^

Fig. 3. Illustration of phase detection.

Author's personal copy

the overall trace and consequently cause false positives in the wavelet analysis.

The wavelet filtering computes the level-1 coefficient for each access sample and removes from the signals the accesses with a low wavelet coefficient value. An access is kept only if its coefficient ω is greater than $m + 3\delta$, where m and δ are the mean and standard deviation of the wavelet coefficients. The difference between such a coefficient and others is statistically significant. We have experimented with coefficients of the next four levels and found the level-1 coefficient adequate.

Fig. 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 ignores the changes due to the local peaks and preserves the four points of global phase changes. As a result of the filtering, most accesses to the data sample are removed except for those that mark the phase boundary.

The filtering process is illustrated with an example in Fig. 3. Part (1) shows a sequence of the access time of three data samples A, B, and C. Bold-face numbers show the time of abrupt reuse distance changes as identified by the wavelet analysis (the actual reuse distance is not shown). The filtering removes all but these time samples for each data sample, as shown in Part (2). The time samples from the three data samples are recombined into the filtered trace, shown in Part (3). The filtered trace includes only the identity of the data not the access time or the reuse distance.

Wavelets are used by Joseph et al. to analyze the change of processor voltage over time and to make online predictions using an efficient Haar-wavelet implementation [27]. Instead of the whole trace, our method filters access sub-traces. Fourier transform is used by Sherwood et al. to find periodic patterns in an execution trace [40]. It shows the frequencies in the whole signal, while wavelets give the time in addition to the frequency information. After wavelet filtering, the remaining accesses from sub-traces are ordered by their logical time into a new trace we call the *filtered trace*.

2.1.4. Shortest-path-based phase partitioning

At a phase boundary, the access pattern changes for many data. Since the wavelet filtering removes reuses of the same data within a phase, the remaining elements are mainly accesses to different memory locations clustered at phase boundaries. For example in Fig. 3(3), the elements in the two sub-traces "*ABC*" and "*CAB*" signal two-phase changes (not six-phase changes). The first C may due to noise or incomplete variable-length sampling.

Our observations of traces from real programs 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 wavelet filtering: not all reuses represent a phase change.

We convert the filtered trace into a directed acyclic graph (e.g. Fig. 3(4)) where each node is an access in the filtered trace. Each node has a directed edge to all succeeding nodes. Each edge, e.g. from access a to b, has a weight defined as $w = r\alpha + 1$, where $1 \geqslant \alpha \geqslant 0$, and r is the number of node recurrences between a and b. For example, the trace *aeefgefb* has two recurrences of e and one recurrence of f between a and b, so the edge weight between the two nodes is $3\alpha + 1$.

Intuitively, the weight measures the fit of the segment between a and b 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 fewest reuses in each phase. The factor α controls the relative penalty for too large or too small phases. In extreme cases, if $\alpha \geqslant 1$, we prohibit any reuses in a phase. We may have as many phases as the length of the filtered trace. If α is 0, we get one phase. In experiments, we found that the phase partitions were similar when α was between 0.2 and 0.8, suggesting that the noise in the filtered trace was acceptable and the algorithm was not highly sensitive to the value of α. We used 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 other nodes, and a sink node that has directed edges coming from all other 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.

Fig. 3(4) and (5) illustrates the shortest-path-based analysis. First, a start and an end node is added to the filtered trace. A directed edge is added from a node to every later node with the weight $1 + 0.5r$, where r is the number of repeated accesses between the source and the target of the edge. The shortest path from the start to the end is the one that passes the third C in the sequence. The path has the weight 2.5. It partitions the trace into two phases at the last access of C. Other partitions have a worse weight. For example, the weight of treating the whole trace as a phase is 3, the weight of the edge from the start to the end. On the other hand, if we do not allow repeated accesses in a phase and treat each occurrence of C as a phase change, the partition weight is also 3. The min-path analysis steers between under-partitioning and over-partitioning. Note that the shortest path is not unique: the path from the start to the second "C" before reaching the end also has the minimal weight of 2.5. The ambiguity is alleviated by the next step, frequency-based phase marking, which selects phase markers based on the *number* of phases not the time of the phase change.

2.2. Frequency-based phase marking

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. The analysis examines all instruction blocks, which is equivalent to examining all program instructions.

Phase detection cannot locate the precise time of phase transitions. The wavelet filtering loses accurate time information because samples are considered a pair at a time (to measure the difference). The precision is in the order of hundreds of memory accesses while a typical basic block has fewer than ten memory references. Moreover, the phase may change through a transition period instead of a transition point. Since the exact time of a phase change is difficult to attain, we use the frequency of the phase changes.

We define the frequency of a phase by the number of its instances 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 instance. 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. Therefore, 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 mutually 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 time gaps between the remaining blocks. If a time gap is larger than a threshold, it is considered as a phase instance. The threshold is determined by the length distribution of the time gaps, the frequency of the phase changes, and the length of execution. Since we used training runs of at least 3.5 million memory accesses, we simply used 10 000 instructions as the threshold. In other words, a phase execution must consume at least 0.3% of the total execution to be considered significant.

Author's personal copy

We can use a smaller threshold to find sub-phases after we find large phases.

Once the phase instances are identified, the analysis considers the block just before the instance as the beginning marker separating the preceding instance and this one. Two instances belong to the same phase if they are led by the same beginning block. The analysis picks markers that mark most if not all instances of a phase in the training run. Our previous work used the block that comes after a phase instance as the marker [39]. It leads to ambiguity in prediction. When instances of the same phase are followed by instances of different phases, a single marker cannot predict the exact identity of the next phase. A similar problem happens when instances of the same phase trail instances of different phases.

We solve the problem by picking first the beginning marker and then the ending marker, which is the first block after a phase instance. The pairing introduces a potential problem when a beginning marker is followed by different ending markers in the trace. The technique examines different pairings until finding a consistent pair. It further ensures the consistency by using multiple training runs. We have considered several improvements that consider the size of the gap, use multiple markers for instances of the same phase, and correlate marker selection across multiple runs. However, the simple 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.

2.3. Phase hierarchy construction

The instrumented program with phase markers outputs a sequence of phase instances during an execution. We call it the phase sequence in short. We identify its hierarchical structure through grammar compression. The purpose is to identify composite phases, increase the granularity of phase prediction, and enable phase-sequence prediction in the next step. For example, for the *Tomcatv* program shown in Fig. 1, five-phase instances form a time step that repeats as a composite phase.

We use SEQUITUR [35], a linear-time and linear-space method to compress a phase sequence into a context-free grammar. We have developed a novel algorithm that extracts phase repetitions from the grammar and converts them to a regular expression. The algorithm recursively converts non-terminal symbols. Each non-terminal symbol is converted to a regular expression when the right-hand side of its product rule is fully converted. If two adjacent regular expressions on the right-hand side are equivalent (using for example the equivalent test described in [23]), they are recognized as a repetition and merged. The conversion remembers previous results so that the same non-terminal is converted only once. SEQUITUR was used by Larus to find frequent code paths [28] and by Chilimbi to find frequent data access sequences [9]. Our method traverses the non-terminal symbols in the same order, but instead of finding sub-sequences, it produces a regular expression.

2.4. Phase-sequence prediction

The phase sequences of some programs show regular patterns. In this work, we consider programs whose phase sequences from different runs yield regular expressions of the same structure. They differ only by the values of the exponents in the expression. In this case, the phase sequence can be predicted if we can determine the exponent values from the program input.

The following is a regular expression representing the phase sequence of an execution of a molecular dynamics simulation program, *MolDyn*, where p_x represents a phase, and the three exponents represent the number of repetitions of sub-expressions. Checking the code, we found that the two inner exponents represent fixed sub-steps, and the outermost exponent is the number of time steps in the simulation. The former two values are always the same, but the number of time steps changes with the input, as specified by a user

$$p_1 p_2 \cdots p_{15} (p_{16} p_{17} (p_{18} \cdots p_{25} (p_{26} p_{27})^2 p_{28} p_{29} p_{30})^5 p_{31} p_{32})^{50}. \tag{3}$$

In phase-sequence prediction, we assume that we have access to two or more inputs to a program. An input includes everything that may influence the program output, which is the set of unbounded variables in Plotkin's abstract notion (see [11, Section 8.2.2]) and on real systems typically includes user inputs, command line options, data files, command files, and environment variables. We call them the components of an input.

Our analysis uses simple heuristics on the input components that it has access to. In this preliminary study, we manually find the input parameters that change from one input to another. Through several training runs on different inputs, the analysis first collects multiple phase sequences and convert them into regular expressions. Then it compares the parameter and the exponent values for each training run. The following two cases are most frequent.

- Case 1: if an exponent value in the regular expression does not change in the training runs, then we assume that its value is constant and independent of the input.
- Case 2: if an exponent changes its value as one or more input parameters, then we assume that its value is equal to one of these input parameters.

Our current implementation is based on the two simple cases: either the value of an exponent is constant across all inputs, or it equals to some input parameter. The other exponents are labeled as unknown, meaning that the system cannot predict how their value changes in a new execution. Two cases are common. The first is the number of iterations for the convergence in an iterative solver, which is not known without doing the actual computation. The second is the size of multi-dimensional arrays. We do not consider this case since the data size does

Author's personal copy

not affect the phase sequence. If an exponent in the phase sequence depends on multiple parameters, we may analyze the relation by multiple training runs and by (assuming we can) changing the parameters one at a time. In our tests, however, the two simple cases suffice.

To make this method general, we need to solve at least three problems. First, a program may have large input files. However, most of the input can be removed from consideration if they do not contain integer values or if the integer values do not equal to one of the exponents in the phase expression. Second, it may be difficult to identify all components of an input. The analysis can be used on the available components and may succeed if the partial input contains the relevant control parameters. In addition, the analysis can use OS support to collect all the inputs in a program execution. The last and the most difficult problem is to identify the values in multiple inputs that represent the same parameter. It may use the meta information such as the name of flags in a command line and the line number in a file to associate values from different inputs. It may also narrow the analysis down to the numbers that equal to one of the exponents in the phase expression and then run correlation analysis between the numbers from different inputs. It may also ask the user if the number of candidates is not too large.

Phase analysis summarizes the program behavior into a regular expression. Because the high-level phase behavior is much simpler and often well structured, we can find the few input parameters that determine the high-level program behavior.

3. Evaluation

We measure the accuracy of phase and phase-sequence prediction and then show the use of phase prediction in dynamic data packing, phase-based memory remapping, and adaptive cache resizing.

Table 1 shows the test suit. 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. *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 [13]. They are commonly studied in dynamic program optimization [16,20,34,43]. We include two SPEC programs, a compiler and a database, as an example of programs that our technique cannot handle because they do not have identically repeating phase behavior. 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.

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

Table 2
The accuracy and coverage of phase prediction

	Strict accuracy		Relaxed accuracy	
	Accuracy (%)	Coverage (%)	Accuracy (%)	Coverage (%)
FFT	100	96.4	99.7	97.8
Applu	100	98.9	100	99.7
Compress	100	92.4	100	93.3
Tomcatv	100	45.6	99.9	99.8
Swim	100	72.8	90.2	99.8
Mesh	100	93.7	100	99.6
MolDyn	96.5	13.5	13.3	99.5
Average	99.5	73.3	86.1	98.5

We use ATOM to instrument programs to collect the data and instruction trace on a Digital Alpha machine [42]. All programs are compiled by the Alpha compiler using "-O5" flag. After phase analysis, we again use ATOM to insert markers into programs. 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 hundred times longer than those used in phase detection.

3.1. Phase behavior prediction

We build a run-time predictor to predict the phase length (i.e. the number of run-time instructions) and cache miss rate. It contains a monitor that measures the first instance of a phase and uses it to predict later instances.

Table 2 shows two sets of results for phase-length prediction. The left half shows the accuracy and coverage of strict phase prediction, where we require that phase behavior repeats exactly including its length. They are chosen automatically from the profiling runs. 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, the coverage increases to 99% for five

Author's personal copy

Table 3
The number and the size of phase instances in detection and prediction runs

Tests	Number of leaf phases		Execution length (M inst.)		Avg. leaf phase (M inst.)		Avg. largest phase (M inst.)	
	Det.	Pre.	Det.	Pre.	Det.	Pre.	Det.	Pre.
FFT	14	122	23.8	5730.4	2.5	50.0	11.6	232.2
Applu	645	4437	254.3	335 019.8	0.4	75.5	3.29	644.8
Compress	52	52	52.0	62 418.4	0.7	800.2	2.2	2712.0
Tomcatv	35	5250	175.0	24 923.2	4.9	4.7	34.9	33.2
Swim	91	8101	376.7	33 334.9	4.1	4.1	37.6	37.0
Mesh	4691	4691	5151.9	5151.9	1.1	1.1	98.2	98.2
MolDyn	59	569	11.9	50 988.1	0.2	89.5	4.0	1699.6
Average	798	3317	863.66	73 938.1	2.0	146.5	27.4	779.6

programs and 98% and 93% for the other two, as shown in the right half of the table. The accuracy drops to 90% in *Swim* and 13% in *MolDyn*. *MolDyn* has a large number of uneven phases in the computation to find neighbors for each particle. For programs with varying phase behavior, the profiling step can often reveal the inconsistency, as it does for *MolDyn* and *Tomcatv*. We can therefore avoid predicting inconsistent phases.

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 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.

Not all programs can be analyzed by our technique. In *Gcc*, the phase length is determined by the function being compiled. Given an input, the distance-based sampling finds peaks that roughly correspond to the functions in the input file. The size and location of the peaks depend on the input. *Vortex* is an object-oriented database. The test input first constructs a database and then performs a set of queries. However, in other inputs, the construction and queries may come in any order. The exact behavior, like *Gcc*, is input dependent and unpredictable. We have developed an extension, which uses a regular input to induce a regular behavior and makes a program amenable to our analysis. However, the new extension requires programmer support and needs to address a different set of constraints [37]. We do not consider these two programs further in this paper.

We measured the miss rate on a real machine to include effects from the code and the operating system. We found that the other factors cause behavior variation in small-length phases but not in long-range phases [39].

3.2. *Phase-sequence prediction*

The most used measure of locality is the cache miss rate of a program execution. Because it summarizes the behavior of possibly many billions of instructions into a single number, the information loss is substantial. Furthermore, the miss rate often changes when the same program uses another input. Take an analogy of weather prediction. The annual average temperature at Rochester does not tell how much the temperature fluctuates between different seasons and wavers between neighboring cities. Adaptation schemes require predictions that are sensitive to time and inputs. In this section, we use phase-sequence prediction to predict the locality changes across phases and program inputs.

Phase-sequence prediction uses multiple inputs in training runs, as described in Section 2.4. Using phase markers and correlating the behavior of the same phase under different inputs, the analysis builds a model that predicts the reuse signature of each phase. A reuse signature is a histogram of the reuse distance of all memory accesses by a phase instance in a training run. The reuse signature gives the cache miss rate for all sizes of fully associative cache, which is very close to the miss rate of two or more ways set associative cache of the same size [48]. We can then predict how the miss rate changes across phases and program inputs.

We use the same test suit as mentioned earlier except for *Mesh* and *MolDyn*, for which we do not have enough test inputs. Table 4 shows the training and test inputs of the remaining programs. Here different inputs differ only in the size of the input data, and they have the same number of static phases and run-time phase instances, as given in the table.

Table 5 shows the average accuracy of phase-based cache hit-rate prediction. The first row shows the accuracy when we predict the histogram at the granularity of data elements. The second row shows the accuracy when 32-byte cache blocks are

Authors personal copy

Author's personal copy

Table 4
The training and test inputs

Inputs	*Applu*	*Compress*	*FFT*	*Swim*	*Tomcatv*
Train 1	12^3	10^2	32×4	160^2	100^2
Train 2	20^3	10^3	64×4	200^2	200^2
Train 3	32^3	10^4	128×4	300^2	300^2
Train 4	40^3	10^6	256×4	400^2	400^2
Test	60^3	1.4×10^8	512×4	512^2	513^2
Num. of phases	195	4	7	17	9
Num. of instances	645	52	37	92	15

Table 5
Average accuracy of phase reuse signature prediction and cache hit-rate prediction

Benchmark	*Applu*	*Compress*	*FFT*	*Swim*	*Tomcatv*	Avg.
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

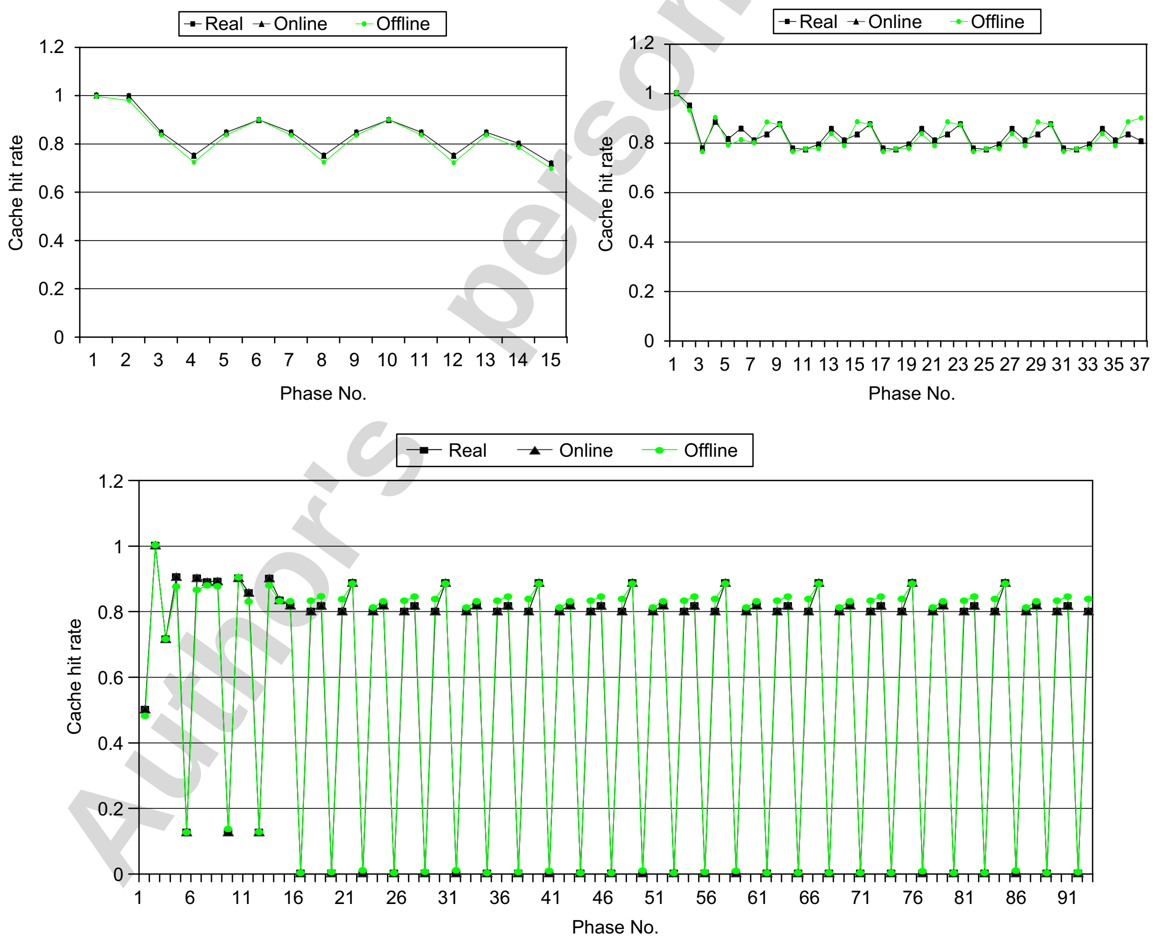


Fig. 4. Cache hit rate online and offline prediction: FFT (top left), TOMCATV (top right), and SWIM (bottom).

used. While the prediction accuracy for the element granularity is less than 90% for some programs, the accuracy for cache blocks is never below 93%.

Fig. 4 shows the temporal curve of the measured and the predicted cache hit rates for *FFT*, *Swim*, and *Tomcatv*. The three programs are representative in their different cache behavior

Author's personal copy

and varying numbers of phases and phase instances. *FFT* has only 15 dynamic phase instances, whose all hit rates are greater than 70%. The curve of *Swim* oscillates dramatically; some phases have hit rates greater than 79%, while some are close to zero. *Tomcatv* has twice as many phases as *FFT*. The "Offline" curves show the prediction using the method described above. The per-phase prediction is accurate in most cases. That method does not require the execution of the program, which suggests the power to predict for any hardware configuration. The "Online" curves show the prediction during the execution of the program. In that case, we use the cache hit rate of the first instance of a phase for the prediction of its later instances. The predicted curves are so close to the real ones that they are not distinguishable from each other.

The locality prediction using reuse distances has been studied extensively after the pioneering work by Mattson et al. [33]. Smith and Hill extended it to cache with limited associativity [22]. Ding et al. enabled the prediction of the overall cache miss rate across program inputs [17,48,38]. This work moves one step further and predicts the temporal changes in program locality. It enables a new dimension of program locality prediction—the time. The results show that the locality-phase-based approach works well for the test programs in our experiments. The temporal change of locality predicted by our method agrees with the measurement closely even for programs with dramatically changing locality. The new method is therefore important to reduce the overhead and errors in software and hardware adaptation. For example, it allows a compiler to trade a slowdown in an infrequent phase for an improvement in a frequently executed phase. It may also improve benchmark design by avoiding repeated instances of the same phase. Next we show three more concrete uses of the phase behavior prediction.

3.3. *Uses of locality phase prediction*

We discuss three uses of the locality phase prediction. Due to lack of space, we give major results without describing all the experimental setup. A detailed description can be found in the conference publication [39].

The first use is in dynamic data packing. Four of the test programs, *Swim*, *Tomcatv*, *Mesh*, and *Moldyn*, are the simulation of grid, mesh, and N-body systems in time steps. These programs benefited from dynamic data packing [16,20,34,43,44]. These techniques are automatic except for the manual marking of phases especially the time step loop.

We compare the manual marking with automatic marking. The outermost phase in the phase hierarchy correctly marks the time step loop in all four programs. Hence, locality phase prediction allows dynamic data packing to be automatically applied for physical, engineering, and biological simulation.

The second use is in phase-based memory remapping. We assume that the machine is equipped with the *Impulse* memory controller, developed by Carter and his colleagues at University of Utah [47,46]. *Impulse* reorganizes data, e.g. creating a column-major version of a row-major array, via remapping instead of actually copying. For programs with regular data access patterns, the Impulse support removes most of the cost of dynamic data remapping.

Table 6
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%)

We consider affinity-based array remapping, where arrays that tend to be accessed concurrently are interleaved by remapping [49]. 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 locations of phase markers. Table 6 shows the execution time in seconds on a 2 GHz 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 [49], as shown in Table 6. 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 [46].

Finally, we use locality phases to drive cache reconfiguration. During an execution, cache resizing reduces the physical cache size without increasing the miss rate [3,25]. 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 mapping to 8-way set associative, so the cache size can change between 32 and 256 KB in 32 KB units. In the adaptation, we need to predict the smallest cache size that yields the same miss rate as the 256 KB 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 10 K 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. In simulation, we use *Cheetah*, a cache simulator that measures the misses of all eight cache sizes at the same time [45].

For comparison we use interval-based methods, which classify past intervals and predict the behavior class of the next interval using methods like last-value or Markov models. We test interval-based prediction using five different interval lengths: 10K, 1M, 10M, 40M, and 100M memory accesses using last-value prediction [3]. In addition, we test a predictor One other scheme uses the instruction footprint of 10M instruction windows [15]. We test a version of this scheme based on a Markov

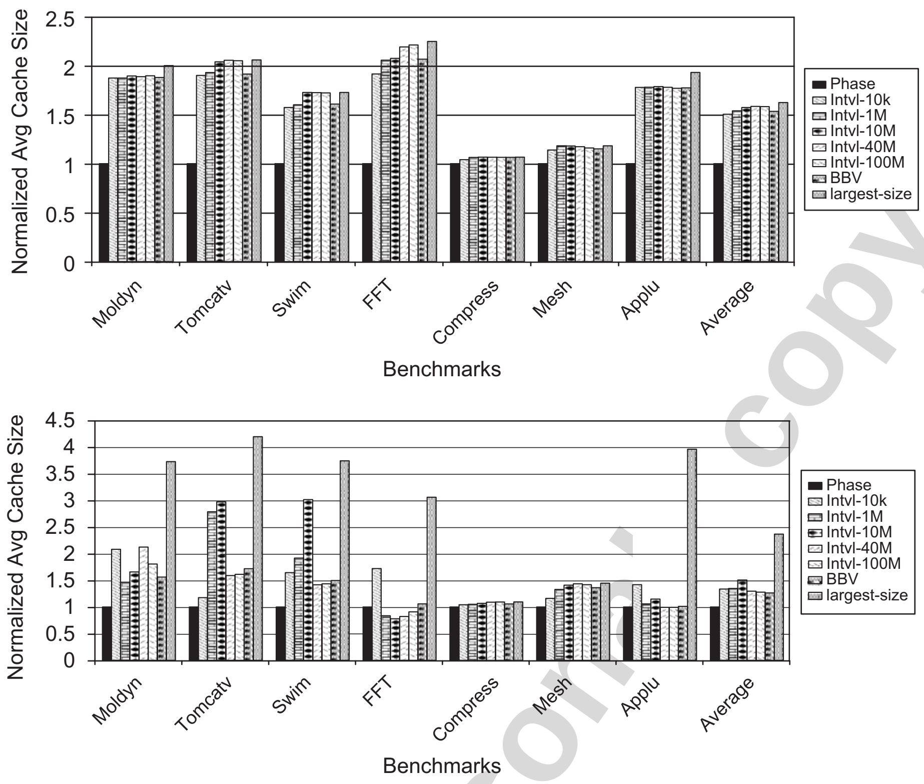


Fig. 5. Average cache-size reduction by phase, interval, and BBV prediction methods, assuming perfect phase-change detection and minimal-exploration cost for interval and BBV methods. Upper graph: no increase in cache misses. Lower graph: at most 5% increase.

Author's personal copy

model and called basic-block vector (BBV) analysis [41]. Interval methods are reactive: they constantly monitor every past interval to decide whether a phase change has occurred. At a phase change, the phase method reuses the best cache size stored for the same phase.

Fig. 5 shows the average cache size from phase, interval, and BBV methods. The top graph shows the results of adaptation with no miss-rate increase. The results are normalized to the phase method. The largest cache size, 256 KB, 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. The lower graph in Fig. 5 shows the results of adaptation with a 5% bound on the miss-rate increase. It shows a similar advantage of locality phases over fixed-length intervals.

The results for interval and BBV methods are idealistic because they use perfect phase-change detection. The result of the phase method is real. 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.

4. Related work

Profiling analysis has been used to identify phases based on loops, functions, and their call sites [3,19,25,29–31]. Huang et al. selected code units that account for at least a given amount of run time (5%) [29]. Georges et al. used an adaptive threshold for different Java programs [19]. Hsu and Kremer used program regions, each of which has a single entry and a single exit and may span a sequence of statements or loops [24]. It is guaranteed to be an atomic unit of execution under all program inputs [1]. Lau et al. considered loops and call sites that have a large size and small behavior variation, and showed that their analysis was faster (because it is code-based) and equally effective for cache resizing as the locality phase prediction [29]. Locality phase prediction does not rely on the static program structure and may identify phase changes even if they do not join the boundary of a region, loop, or procedure. It can identify larger code units as phases, for example, the time step in a scientific simulation program, which may span thousands of lines of code with repeated function calls and indirect data access. It is needed for techniques such as software-based dynamic data remapping [16]. The technique can be directly used on highly optimized binary code where the loop and procedure structures may be obfuscated by compiler transformations.

A number of groups have studied interval phases, which identifies patterns in general executions but often require hardware support and do not associate phases with program code. Joseph et al. first used the 1D Harr wavelet to predict dynamic voltage changes [27]. Recently, Huffmire and Sherwood analyzed the data access trace using the 2D Harr wavelet and found that the new metric predicted the memory behavior better than code-based metrics [26], and Cho and Li used 1D Harr wavelet to derive a complexity metric that can identify different dynamics in execution intervals that have the same average behavior [10]. Since these studies considered only bounded size windows, they used the wavelet transform at all levels.

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 and observed that phases have steady or slow changing locality inside but disruptive transitions in between [4]. They showed experimentally that a set of Algol-60 programs spent 90% time in major phases. Bunt et al. measured the change of locality as a function of page sizes (called the locality curve) in hand-marked hierarchical phases [6]. Using the PSIMUL tool at IBM, Darema et al. showed recurring memory-access patterns in parallel scientific programs [12]. These studies did not predict locality phases. A number of other studies used time or reuse distance as well as predictors such as Markov models to improve virtual memory management. Recently, several studies found predictable patterns in the overall locality but did not consider the phase behavior [17,18,32].

One of the most effective techniques for locality analysis is dependence theory and loop-nest optimization (see [2] for a recent textbook). However, basic dependence analysis is not as effective for programs with input-dependent control flow and data indirection. Various extensions of dependence analysis have been developed [36,43,50]. The framework that combines compile- and run-time checks for data parallelization is known as inspector–executor [13]. The correctness of dynamic data reorganization can be ensured by the support of run-time data remapping [16]. For regular programs, the reuse distance can be directly measured by a compiler. Cascaval and Pudua extended dependence analysis to measure reuse distance without profiling or sampling [8]. Beyls and D'Hollander formulated reuse distance equations and compared it with profiling-based analysis (when used for cache-hint insertion) [5]. Cascaval et al. used continuous monitoring of page reuse distances to manage data on machines that supported large, multi-size pages [7].

5. Conclusions

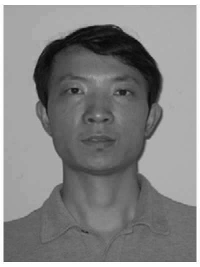
The paper presents a method for predicting hierarchical locality phases in programs with input-dependent but consistent phase behavior. Based on profiling runs, it predicts program executions hundreds of times larger and predicts the length, the locality, and the sequence of locality phases. It has solved an open problem in dynamic data adaption, naming identifying the outermost phase in memory-intensive applications such as time steps in scientific simulation. 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%. In addition, the phase structure allows the analysis to find input parameters that affect the high-level program behavior. It predicts the temporal change of locality across program inputs with 98% accuracy. It also recognizes programs with inconsistent phase behavior and avoids false predictions. It is more effective at identifying long, recurring phases than previous methods based on program code, execution intervals, and manual analysis, because the new technique not only combines analyses of the program code and data access but also integrates offline training and online prediction. The accurate phase-behavior prediction should help modern adaptation techniques for increasing performance, reducing energy, and other improvements to the computer system design.

References

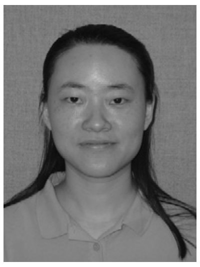
[1] F. Allen, J. Cocke, A program data flow analysis procedure, Commun. ACM 19 (1976) 137–147.
[2] R. Allen, K. Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-Based Approach, Morgan Kaufmann, Los Altos, CA, 2001.
[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, S. Dwarkadas, 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] A.P. Batson, 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] K. Beyls, E.H. D'Hollander, Generating cache hints for improved program efficiency, Journal of Systems Architecture 51 (4) (2005) 223 –250.
[6] R.B. Bunt, J.M. Murphy, S. Majumdar, A measure of program locality and its application, in: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, 1984.
[7] C. Cascaval, E. Duesterwald, P.F. Sweeney, R.W. Wisniewski, Multiple page size modeling and optimization, in: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, St. Louis, MO, 2005.
[8] C. Cascaval, D.A. Padua, Estimating cache misses and locality using stack distances, in: Proceedings of International Conference on Supercomputing, San Francisco, CA, June 2003.
[9] 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.
[10] C. Cho, T. Li, Complexity-based program phase analysis and classification, in: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, 2006.
[11] K. Cooper, L. Torczon, Engineering a Compiler, Morgan Kaufmann, Los Altos, CA, 2003.
[12] F. Darema, G.F. Pfister, K. So, Memory access patterns of parallel scientific programs, in: Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1987.
[13] R. Das, M. Uysal, J. Saltz, Y.-S. Hwang, Communication optimizations for irregular scientific computations on distributed memory architectures, J. Parallel Distrib. Comput. 22 (3) (1994) 462–479.
[14] I. Daubechies, Ten Lectures on Wavelets, Capital City Press, Montpelier, Vermont, 1992.
[15] A.S. Dhodapkar, J.E. Smith, Managing multi-configuration hardware via dynamic working-set analysis, in: Proceedings of International Symposium on Computer Architecture, Anchorage, Alaska, June 2002.
[16] C. Ding, 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.

Author's personal copy

[17] C. Ding, 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.
[18] C. Fang, S. Carr, S. Onder, Z. Wang, Instruction based memory distance analysis and its application to optimization, in: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, St. Louis, MO, 2005.
[19] A. Georges, D. Buytaert, L. Eeckhout, K. De Bosschere, Method-level phase behavior in Java workloads, in: Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, October 2004.
[20] H. Han, 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.
[21] H. Han, C.W. Tseng, Locality optimizations for adaptive irregular scientific codes, Technical Report, Department of Computer Science, University of Maryland, College Park, 2000.
[22] M.D. Hill, Aspects of cache memory and instruction buffer performance, Ph.D. thesis, University of California, Berkeley, November 1987.
[23] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.
[24] C.-H. Hsu, U. Kremer, 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.
[25] M. Huang, J. Renau, J. Torrellas, Positional adaptation of processors: application to energy reduction, in: Proceedings of the International Symposium on Computer Architecture, San Diego, CA, June 2003.
[26] T. Huffmire, T. Sherwood, Wavelet-based phase classification, in: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, 2006.
[27] R. Joseph, Z. Hu, 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.
[28] J.R. Larus, Whole program paths, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia, May 1999.
[29] J. Lau, E. Perelman, B. Calder, Selecting software phase markers with code structure analysis, in: Proceedings of International Symposium on Code Generation and Optimization, March 2006.
[30] W. Liu, M. Huang, Expert: expedited simulation exploiting program behavior repetition, in: Proceedings of International Conference on Supercomputing, June 2004.
[31] G. Magklis, M.L. Scott, G. Semeraro, D.H. Albonesi, 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.
[32] G. Marin, 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.
[33] R.L. Mattson, J. Gecsei, D. Slutz, I.L. Traiger, Evaluation techniques for storage hierarchies, IBM System J. 9 (2) (1970) 78–117.
[34] J. Mellor-Crummey, D. Whalley, K. Kennedy, Improving memory hierarchy performance for irregular applications, Internat. J. Parallel Programming 29 (3) (2001).
[35] C.G. Nevill-Manning, I.H. Witten, Identifying hierarchical structure in sequences: a linear-time algorithm, Journal of Artificial Intelligence Research 7 (1997) 67–82.
[36] L. Rauchwerger, D. Padua, The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, La Jolla, CA, June 1995.
[37] X. Shen, C. Ding, S. Dwarkadas, M.L. Scott, Characterizing phases in service-oriented applications, Technical Report TR 848, Department of Computer Science, University of Rochester, November 2004.
[38] X. Shen, Y. Zhong, 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.
[39] X. Shen, Y. Zhong, 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.
[40] T. Sherwood, E. Perelman, 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.
[41] T. Sherwood, S. Sair, B. Calder, Phase tracking and prediction, in: Proceedings of International Symposium on Computer Architecture, San Diego, CA, June 2003.
[42] A. Srivastava, 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.
[43] M.M. Strout, L. Carter, 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.
[44] M.M. Strout, P. Hovland, Metrics and models for reordering transformations, in: Proceedings of the Second ACM SIGPLAN Workshop on Memory System Performance, Washington, DC, June 2004.
[45] R.A. Sugumar, 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.
[46] L. Zhang, Efficient remapping mechanism for an adaptive memory system, Ph.D. Thesis, Department of Computer Science, University of Utah, August 2000.
[47] L. Zhang, Z. Fang, M. Parker, B.K. Mathew, L. Schaelicke, J.B. Carter, W.C. Hsieh, S.A. McKee, The impulse memory controller, IEEE Transactions on Comput. 50 (11) (2001).
[48] Y. Zhong, S.G. Dropsho, C. Ding, Miss rate prediction across all program inputs, in: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, New Orleans, LA, September 2003.
[49] Y. Zhong, M. Orlovich, X. Shen, 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.
[50] C.Q. Zhu, P.C. Yew, A scheme to enforce data dependence on large multiprocessor systems, IEEE Trans. Software Eng. 13 (6) (1987).



Xipeng Shen is an assistant professor of computer science at the College of William and Mary. He received his Ph.D. in computer science from the University of Rochester. His research focuses on large-scale program behavior analysis, in particular, using statistic techniques to detect and predict program behavior patterns and apply them to program optimizations. His past work covers locality phase analysis, efficient data locality measurement and prediction, and parallel sorting.



Yutao Zhong is an assistant professor of computer science at the George Mason University. She received her Ph.D. in computer science from the University of Rochester. Her research focuses on compiler-assisted program locality analysis, including efficient locality pattern modelling and prediction, memory performance estimation, and program data management based on reference affinity.

Author's personal copy

Chen Ding is an associate professor in the Department of Computer Science, University of Rochester. His research focuses on the analysis and adaptation of large-scale program behavior. He is the recipient of the Early Career Principal Investigator award from the Office of Science of the US Department of Energy, the CAREER award from US National Science Foundation, the Center for Advanced Studies Faculty Fellowship from IBM, and a best-paper award from the IEEE International Parallel and Distributed Processing Symposium. He was the co-organizer of the first and the General Chair of the second ACM SIGPLAN Workshop on Memory System Performance (MSP) in 2002 and 2004.

Author's personal copy