Predicting Hierarchical Phases in Program Data Behavior

Xipeng Shen Yutao Zhong Chen Ding

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

ABSTRACT

Computer memory hierarchy becomes increasingly powerful but also more complex to optimize. Run-time adaptation emerges as a promising strategy. For software, it means adjusting data behavior at different phases of an execution. For hardware, it means reconfiguring the memory system at different times.

This paper presents a method that predicts the memory phases of a program when it runs. The analysis first detects memory phases in a profiling run using variable-distance sampling, wavelet filtering, and optimal phase partition. It then identifies the phase hierarchy through grammar compression. Finally, it inserts phase markers into a program through binary rewriting. The technique is a unique combination of locality profiling and phase prediction.

The new method is tested on a wide range of programs against programmer manual analysis and pure hardware monitoring. It predicts program executions that are thousands of times longer than profiling runs. The average length of the predicted phases is over 700 million instructions, and the *length* is predicted with 99.5% accuracy. When tested for cache adaptation, it reduces the cache size by 40% without increasing the number of cache misses. These results suggest that phase prediction can significantly improve the many adaptation techniques now used for increasing performance, reducing energy, and other improvements to the computer system design.

1. INTRODUCTION

It has long been observed that programs have marked phased behavior. As early as 1976, Batson and Madison defined a phase as a period of execution that accesses a sub-set of program data. From running real Algol-60 programs, they observed that 90% of execution time was covered by major phases, and that phases have a hierarchical structure where smaller phases accessed locality subsets and were embedded in larger ones [3]. The phase locality has many later uses, among recent ones are memory management, trace compression [21] and the file caching for PCs, servers and databases [14, 25].

With the widening gap between processor speed and memory speed, cache is receiving increasing attention. As the hardware becomes increasingly powerful but also more complex to optimize. Run-time adaptation is rapidly emerging as a promising strategy, demonstrated extensively through dynamic data reorganization in a program [8, 11, 19, 23] and through cache or processor reconfiguration on a machine [1, 2, 7, 13, 16, 17, 22]—six of them appeared in one year (2003). As discussed in more detail in Section 4, these new techniques are based on either program code, run-time machine usage, or their combinations. None has considered program data locality as a factor in program phase prediction.

In this paper, we return to the phase definition of Batson and Madison and study locality phases and its uses in data and cache adaptation. We present a method that predicts the memory phases of a program when it runs. The technique operates in four steps. The first analyzes the data locality in a profiling run. By examining distances of data reuses in decreasing length, the analysis can ”zoom out” and ”zoom in” over very long traces and detects major phases using variable-distance sampling, wavelet filtering, and optimal phase partition. The second step then analyzes the instruction trace and identifies the phase boundaries in the program code. It considers all program points as possible phase changing points and favors those points that identify unique program phases. The third step recognizes the hierarchical phase structure using grammar compression. It converts a compressed grammar to a phase hierarchy. Finally, the last step uses binary rewriting to instrument the program. During execution, the instrumented program monitors the behavior of the first occurrence of each phase and uses it as the prediction for later occurrences.

We motivate the use of this analysis through the example of dynamic data reorganization. Recent studies have shown that runtime data reorganization substantially improves the cache and consequently program performance of dynamic programs in diverse areas including scientific simulation, weather prediction, drug manufacture, and automobile design [8, 11, 19, 23]. A common class is N-body simulation, which computes the movement of objects and their changing interactions in many time steps. The computation is dynamic because the distribution of the objects remains unknown until run time and may change during execution. An adaptive program would monitor its data access pattern and re-adjust the data layout periodically. The transformation needs to know the top-level program phase, for example, a time step in the N-body simulation. A smaller phase does not give the whole pattern of data access. A larger phase cannot either unless it includes an integer number of time steps. For complex programs, previous work used programmer input to identify the top level phases, which may include thousands of lines in program code and span seconds or minutes in time and consequently billions or trillions of instructions in program execution [8]. If the phase prediction can identify all program phases, it will allow the dynamic data optimization to be fully automated.

In addition to program adaptation, phases have many uses in machine adaptation. In a phase where a program stalls frequently for memory, the processor can switch to a slower speed using dynamic voltage scaling, a technique already implemented in processors from AMD, Intel and Transmeta. For future processors, researchers have also devised schemes where cache size and associativity can be dynamically adjusted for different parts of an execution [1]. Machine adaptation involves significant overhead including the cost of changing hardware configuration, the time to find the right configuration, and the performance loss due to a bad

decision or the exploration overhead. To be harmless, a machine should have a hard bound on the performance loss due to the adaptation and let a user to set the bound as appropriate. Phase prediction can significantly improve machine adaptation by choosing the right configuration at the right time.

In the evaluation section, we show that the automatically inserted phase markers coincide with the markers inserted by a programmer. When tested for cache adaptation, phase prediction consistently outperforms pure hardware methods. Compared to other existing analysis, the new method can find repeating phases of a large size that is up to 3 billion instructions.

Our analysis may not be effective on all programs. Some programs may not have predictable phases. Some phases may not be predictable by looking at their data locality. In addition, we limited the 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 locality analysis can still identify phases even though it cannot predict the exact length.

2. HIERARCHICAL PHASE ANALYSIS

A phase can be very small, e.g. one instruction; it can also be very large, e.g. a whole program. Different uses may need different sizes. Our system detects phases and builds a phase hierarchy. The whole program is the root. Phases are nodes in the hierarchy. The phases on the highest level (except the root) are the largest phases. At leaves are the smallest phases—basic blocks. We could extend the hierarchy to individual instructions if needed in practice. The phase hierarchy represents all predictable data behavior phases. This section describes the four steps of phase analysis.

2.1 Phase detection

Given the execution trace of a program on a training run, we need to detect the major phases. Since the trace may contain hundreds of billions of instructions and memory accesses, we must first look for coarse-grain patterns, just as one should not use microscopes to take a picture of a bird. Previous phase analysis either used fixed thresholds [16, 17] or fixed length intervals [1, 10, 22], which are not flexible. Programs are different. Even for the same program, they can be too small for the execution from one input but too large for another. Furthermore, different parts of an execution may need a different treatment.

To avoid using any pre-determined thresholds, we use locality analysis, which automatically selects the "natural" thresholds for each execution. We first illustrate the use of locality analysis through an example. Then we describe the three steps of phase detection. The first step, variable-distance sampling, samples program data and collects their long-distance reuses. Long distances are caused by large phases; therefore, distances reflect the actual size of phases. The second step, wavelet filtering, uses the powerful signal-processing technique to reduce noises in the samples. The last step, optimal phase partition, finds phases as basic units of repetition.

2.1.1 Reuse distance

Mattson defined the LRU-stack distance as the number of distinct data elements accessed between two consecutive references to the same element [18]. For brevity we call it *R*euse distance as Ding and Zhong did [9]. We convert a data-access trace into a reuse-distance trace by replacing each access by its reuse distance. The first use of an element has zero distance. The reuse distance trace reveals the data reuse pattern in an execution.

We use *Tomcatv*, a mesh generation program from SPEC95, as an example. The program is regular enough that compiler loop analysis can recognize the major phases. However, to handle all programs, our tool needs to find the same phases without knowing loop or other program information. It must detect phases in large execution traces. *Tomcatv* has hundred millions of memory accesses in a small input and over 25 billion memory accesses in a realistic input. Figure 1(a) shows the sampled reuse distance trace of a small input [1]. The trace has 29 different intervals, each is a dark block or a line. The change is smooth within each interval but abrupt between intervals. It is important to note that the global patterns are seen without using any fixed-size windows. The other important observation is that the trace reveals high-level repetitions. Intervals 6 to 11 form a period that repeats four times, counting intervals 1 to 5 as an incomplete one. The incompleteness is because some data don't have reuses until the later iterations.

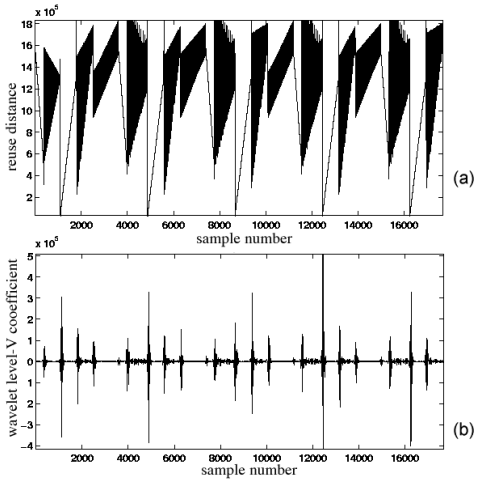


Figure 1: (a) Reuse distance trace of TOMCATV. (b) Level-V wavelet coefficients of the reuse distance trace of TOMCATV.

```
Initialization;
for i := 0 to 4 do
    Calculate array values for this iteration;
    Determine Maximum values of residuals;
    Solve tridiagonal systems–Step I;
    Solve tridiagonal systems–Step II;
    Add corrections;
end for
Output results;
```

Figure 2: The program structure of *Tomcatv*, vectorized mesh generation program from SPEC95fp benchmark suite

Figure 2 shows the program structure of *Tomcatv*. Indeed it has five iterations, and each iteration has five major steps. By instrumenting the source code, we obtained the execution time of each iteration and each step and found that they aligned with the intervals, demonstrating the relation between reuse distance and the program phase structure.

2.1.2 Variable-distance sampling

Instead of analyzing all accesses to all data, variable-distance sampling finds a representative sub-set of data and their accesses.

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

Major memory phases are formed by the traversal of large data structures, so they must contain accesses with long reuse distances. We can ignore short-distance reuses of data. In addition, we need not monitor every element in large structures. It is sufficient to sample a small number of them. For example, if a program repeatedly traverses a large array, sampling a single element is often enough to get the basic reuse pattern for all elements.

Variable-distance sampling is based on the distance-based sampling described by Ding and Zhong [9]. Their sampler monitors the reuse distance of each access. When the reuse distance is above a 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 another threshold. To avoid picking too many data samples, it requires that a new data sample to be at least a certain space distance away in memory from existing data samples.

The three thresholds in Ding and Zhong's method are difficult to control. Variable-distance sampling uses feedbacks to find the suitable thresholds. It takes as the input the desirable size of the trace after sampling, that is, the number of access samples N. Given an arbitrary execution trace P, it divides P into intervals, each with 10 million accesses. Then the sampler starts with an initial set of thresholds. After each interval, the variable-distance sampler checks whether the number of samples is too many or too few considering the trace length and the targeted sample trace length. It changes the thresholds accordingly to decrease or increase the number of samples.

The variable-distance sampling collects samples at an even rate. It may include partial results for some data samples when the execution has an uneven reuse density. However, the target size of the sampled trace is large. The redundancy ensures that these samples together form a complete picture of program data access. If a data sample has too few access samples to be useful, the later analysis will remove its effect as noises. The variable-distance sampling cannot always meet the exact target size of sampled trace, but it does not need to. The later analysis only wants the sampled trace to have a reasonable size. Sampling is part of the off-line analysis. It can spend more time to find appropriate thresholds. In practice, variable-distance sampling arrives at steady thresholds in less than 20 iterations.

2.1.3 Wavelet filtering

Viewing the reuse-distance trace as a signal, we use wavelet as a filter to expose abrupt changes in reuse pattern. Wavelet is a common technique in signal processing, image processing and other areas [6]. Like Fourier Transform, wavelet gives the frequency information for a temporal signal. One important difference, however, is that wavelet gives the *time-frequency* representation of the signal, instead of only the frequency representation from Fourier Transform. Wavelet tells what frequencies appear during an interval. Fourier transform only shows what frequencies appear during the whole signal lifetime. The formula of wavelet transform is as follows:

$$W_f(u,s) = \langle f(t), \psi_{u,s} \rangle = \int_{-\infty}^{+\infty} \frac{1}{\sqrt{s}} \psi^*(\frac{t-u}{s}) dt$$

where, $f(t)$ is the input signal, $\psi_{u,s}$ is a function family generated by translations and dilations of a window (the reason for the name wavelet). Many different wavelet families exist in literature, such as *Haar, Daubechies, Biorthogonal*. We use *Daubechies-6*, a commonly used wavelet, in our experiments. Other families produce a similar result.

Usually, wavelet applies two functions to data: scale function and 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 length. After finishing the calculations on the whole signal, it starts the next level process, which does the same thing, on the scaled results from the last level instead of on the original signal. This process may continue for many times, which makes wavelet a multi-granularity process. A sequence of wavelet parameters are output on each level. They correspond to the reuse-distance trace. The phase changing points usually correspond to the high parameter values of the first 5 levels. Figure 1(b) shows the wavelet level-V parameters, which our analysis uses, for the reuse distance trace of *Tomcatv*, shown in Figure 1(a). The wavelet filtering step extracts the points with high parameter values as the possible phase changing points.

We use wavelet to filter the reuse-distance trace of each data sample separately and then recompile the remaining points from all data samples in time order. The new trace is called *trace-w*. Intuitively, this process removes repeated accesses to each sampled memory location in the same phase. But accesses to different locations in *trace-w* may correspond to the same phase boundaries. We use optimal phase partition to remove those redundant accesses and determine the exact phase boundaries.

2.1.4 Optimal phase partition

In the sampled trace *trace-w*, wavelet analysis remains a unique data access of data samples in each phase. On the other hand, it may have many data samples accessed in a phase. The reason is that at a phase changing point, many data (e.g. array elements) change their access patterns. These two properties of *trace-w* suggest two conditions for a phase.

First, a phase should include the 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 to the same data sample. Data reuses in *trace-w* indicate different phases. The complication, however, comes from the imperfect filtering by wavelet, so not all reuses represent a phase change.

We convert *trace-w* into a directed acyclic graph where each node is an access in the trace. Each node has a directed edge to the node of all later accesses. 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 reuses between a and b. For example, in trace $aceefgefbd$, the weight on edge from c to b is $3\alpha+1$. The results are the same as $alpha = 1$ when α is greater than 1. The smaller α is, more generous we are to repetitions of accesses, which is for the case when many reuses exist in the same phase even after wavelet filtering.

Intuitively, the weight is the measure of how fit the interval 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 no reuses in any of them. Since the trace is not perfect, the weight and the factor α control the relative penalty for too many in-phase reuses 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 *trace-w*. If α is 0, we want the fewest number of phases, which is one. Our experiment uses $\alpha = 0.4$.

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 flowing 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 path has the minimal penalty is the shortest path in this graph.

2.1.5 Observations

Reuse distance provides the memory access pattern of programs. A change from one access pattern to another signals a phase change. Wavelet filters remove the temporal redundancy—repeated access that are not on the boundary of a phase. Shortest-path analysis removes the spatial redundancy—data that are accessed in the same phase. We note the remaining difficulties for this scheme. A pattern change may happen through a transition period instead of a transition point. The wavelet does not give the accurate temporal information because of the averaging effect of the scaling operation. In addition, noises exist in program behavior. As a result, the exact time of a phase change is very difficult to attain. The lack of the precise time becomes a problem when we try to mark the phase boundaries in the program code, a step we describe in the next section.

2.2 Phase marker selection

The instruction trace of an execution is recorded at the granularity of basic blocks [2]. The result is a block trace, where each element is the label of a basic block. By analyzing the block trace, this step finds the basic blocks in the code that uniquely mark detected phases. Past program analysis considered only a subset of code locations as possible phase-changing points, for example only function entries, loop headers, or call sites [13, 16, 17]. Our analysis examines all instruction blocks as marker candidates, which is equivalent to examining all program instructions as candidates. This is especially important at the binary level, after a program is aggressive optimized by compiler transformations such as procedure in-lining, code splitting, common-code extraction, and loop fusion.

As explained at the end of Section 2.1, phase detection finds the number of phases but cannot locate the precise time of the phase transition—the precision is in the unit of hundreds while the length of one basic-block execution is often less than 10 (memory accesses). Furthermore, it is impossible to find a single point if the transition is gradual. We solve this problem by using the number of detected phases instead of the time of the detected transition.

Let the number of detected phases be M. Any phase boundary block cannot appear more than M times in the block trace. Therefore, the first step of the marker selection filters the block trace and keeps only blocks whose frequency is no more than M. If a loop is a phase, the filtering will remove the occurrences of the loop body block and keep only the header and the exist block. If a set of mutual recursive functions forms a phase, the filtering will remove the body blocks from these functions and keep the blocks before and after the root invocation. After filtering, the remaining blocks are possible places for phase markers. They provide a coarser view of the program execution, where the phase boundaries are easier to find than they were in the original trace that had a lot of noise.

After filtering, long temporal intervals or blank regions appear between remaining blocks. These are the holes left by the removed blocks. The second step identifies large blank regions as phases. The size of a blank region should be based on the length distribution of the blank regions, the expected number of phases M, and the total execution length. But we found that a single threshold—10000 in logical time—was sufficient because our training runs had at least 3.5M memory accesses, and the block trace is sufficiently sparse after filtering.

Once the phases are identified, the analysis considers the blocks that are immediately after a phase as markers for the end of this phase and the beginning of the next phase. The analysis then checks for uniqueness. It labels blank regions with their length and checks whether a block always leads regions of the same length. These blocks are unique markers. The analysis picks one unique marker for each phase boundary—the one that marks the most if not all instances of the phase in the training run.

The phase frequency gives a necessary but not sufficient condition. It is possible that a phase is fragmented by infrequently executed code blocks in the middle of the phase. However, the break-down does not happen more often than M times. In addition, these partial phases will be regrouped in the next step, phase hierarchy construction.

[2] A basic block in a program binary is a sequence of contiguous instructions that has only one entry at the beginning and one exit at the end.

```
Input string:
a b a b a b a b a b a b a b c c c c e f c c
c c e f

Output grammar:
0 -> 1 1 1 2 3 3
1 -> 2 2
2 -> [a] [b]
3 -> 4 4 [e] [f]
4 -> [c] [c]
```

Figure 3: An example of SEQUITUR grammar compression. The output grammar stores the repeating phases in a hierarchy rather than a list.

2.3 Phase hierarchy construction

After the previous two steps to identify the phases, the construction step organizes them in a hierarchy, where each node is either a basic phase, a sequence of sub-phases, or a repetition of a phase. For example for the *Tomcatv* program we showed earlier in Figure 2, the first two steps find all the basic phases. This step needs to recognize that every six phases form an iteration that repeats five times. By constructing the phase hierarchy, this step exposes phases at all granularity.

We use a grammar compression algorithm, SEQUITUR, to build the phase hierarchy from a phase sequence. SEQUITUR, developed by Nevill-Manning and Witten, is a linear-time and linear-space compression method [20]. Given a sequence of symbols, it constructs a hierarchy and represents it by a context free grammar(CFG). The output of a compressed grammar satisfies two basic constraints:

(1) No consecutive sequence appears twice.
(2) Each rule must be used at least twice.

The first rule is important for reducing the total size of the grammar because it ensures that no more compression opportunity exists. However, for a phase trace, it will hide the repetition. Figure 3 shows an example phase sequence and its compressed grammar from SEQUITUR. A phase hierarchy should recognize that “ab” repeats 7 times, “ccccef” twice, and “c” 4 times inside “ccccef”. The grammar from SEQUITUR has five non-terminals and shows little hint of the repetitions because the repeating symbols are represented by a grammar hierarchy rather than a list. The first rule is essential, so we cannot change inside the SEQUITUR algorithm.

We now present an algorithm that extracts phase repetitions from a compressed grammar and represents them explicitly in a regular expression. The algorithm, shown in Figure 3, recursively converts non-terminal symbols into regular expressions. It remembers the previous result so that a non-terminal is converted only once. A merge step happens for each non-terminal when its right-hand side is all converted. Two adjacent regular expression can be merged if

their internal structure is the same. For the example in Figure 3, the algorithm recognizes that the rule 4 is $(c)^2$, the rule 3 is $((c)^4ef)$, and so on. In the end, it converts the rule 0 to $(ab)^7((c)^4(ef))^2$.

So far we have built a hierarchy above the detected phases. The same method can also construct a hierarchy within each phase. Taking the basic-block trace of each phase, we use grammar compression first and then convert the result to a regular expression or a hierarchy. The leaf of the hierarchy is one basic block. The code-block hierarchy does not lend itself for run-time prediction (using the method we will describe in the next section). The reason is cost—it is too expensive to monitor at the basic block level. However, the hierarchy can be used for trace analysis, for example, reducing the time of simulating the phase.

SEQUITUR was used by Larus to find frequent code paths [15] and by Chilimbi to find frequent access sequences [4]. Their methods models the grammar as a DAG and finds frequent sub-sequences of a given length. Our method traverses the non-terminal symbol in the same order as theirs, but instead of finding sub-sequences, we produce a regular expression.

2.4 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. Other phases in the hierarchy do not have a unique marker. For them we insert a predictor into the program. The predictor keeps the phase hierarchy, monitors the program execution, and makes prediction based on an on-line phase history. Since the hierarchy is a regular expression, the predictor needs a finite automaton to recognize the current phase in the phase hierarchy. In the programs we tested so far, this simple method suffices.

During execution, when a phase is first executed, the program records its behavior—for example the number of misses—using either hardware counters or an instrumented version of the program. The results are stored in the marker and in a centralized table maintained by the predictor. When the phase is executed again, the past result will give the prediction at the beginning of the phase execution. The run-time analysis allows accurate prediction for input-dependent phase behavior. The predictor gives large granularity for composite phases. The cost of the markers and the predictor is negligible because they are invoked once per phase execution, which consists of at least millions of instructions as shown in the evaluation.

3. EVALUATION

We evaluate our technique in three experiments. The first measures the granularity and accuracy of the phase prediction in machine-independent metrics. The second compares the automatically inserted markers with programmer inserted markers. The match is important in dynamic data reorganization by a program. Finally, we use phase prediction on a simulated machine that can dynamically change its cache size. We compare phase-based adaptation with hardware interval-based adaptation.

Our test suite is given in Table 1. We pick programs from different sets of commonly used benchmarks to get an interesting mix, representing common computation tasks in signal processing, combinatorial optimization, structural and unstructured N-body simulations, compiler, and database. *FFT* is a basic implementation from any standard 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 behavior depends on program inputs and changes during execution [5]. They are commonly used in studying dynamic program optimization [8, 11, 19, 23]. 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. 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

For programs from SPEC, we use the *test* or *train* input for phase detection and *ref* input for phase prediction. We pick different inputs in other programs. For the prediction of *Mesh*, we used the same mesh as training input but with the sorted edges. For all other programs, the prediction is tested on executions hundreds times longer than those used in phase detection. The size difference will be shown in detail later.

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

3.1 Phase prediction

We present an evaluation of phase prediction that is not tied to any specific use. Like Batson and Madison [3], we believe that phase behavior is essentially a program property. Therefore, we use machine independent measures, in particular, the number of instructions and the instruction mix.

The quality of phase prediction depends on three measures: accuracy, coverage, and granularity. We compare accuracy by the predicted phase length or instruction mix with the measured one. Since the two metrics have similar accuracy, we include only results on the number of instructions. The total length of the executed phases gives the coverage. The average length of the executed phases gives the granularity. In general, the higher the three measures are, the better the adaptation is because it supports more aggressive reorganization and reconfiguration in software and hardware. The range of granularity is also important when different adaptation demands different size phases. Phase hierarchy represents all sizes of phases. Due to the space limitation, we show the two extremes of the granularity—the average size of leaf (smallest) phases and the outermost (largest) phases in the hierarchy.

We present results for all programs except for *Gcc* and *Vortex*, which will be discussed at the end of this section. Table 2 shows two sets of results. The upper half shows the accuracy and coverage for strict phase prediction, where we require phase behavior repeats exactly including its length. Except for *MolDyn*, the accuracy is perfect in all programs, that is, the number of instructions in a phase is predicted exactly at the beginning of the phase! The strict

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, because particles have a different number of neighbors. In all programs, the phase analysis can achieve perfect accuracy or full coverage, demonstrating its flexibility in detecting phases of different behavior consistency.

The granularity of the phase hierarchy is shown in Table 3. 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 in all programs is much larger than the detection run, with on average 100 times longer executions in all programs and 400 times more phases. The average size of the leaf phases ranges from 200 thousand to five million instructions in the detection run and from 1 million to 800 million in the prediction run. The largest phase is 13 times the size of the leaf phase in the detection run and 50 times in the prediction run on average.

The results show that the test programs change behavior with the program input in different ways. By training on small data inputs, the analysis predicts for executions that are over 1000 times longer in *Applu* and *Compress* and nearly 5000 times longer in *MolDyn*. The longer executions may have about 100 times more phases (*Tomcatv*, *Swim*, and *Applu*) and over 1000 times larger average phase size (in *Compress*). The different phase sizes indicate that unlikely a fixed interval or threshold would work well for all programs on all inputs.

The programs *Gcc* and *Vortex* are unique because their phase length are not predictable. In *Gcc*, the phase length is determined by the function being compiled. Predicting the next phase is impossible because it means predicting the next function in the input program, which may have any number of functions of any size. The upper figure of Figure 4 shows the reuse-distance sampling result. The peaks roughly correspond to the functions in the input file. *Vortex* is an object-oriented database and has the same property. It processes a sequence of transactions. The number and the size of transactions cannot be predicted. The lower figure of Figure 4 shows the reuse-distance sampling result of the test run. According to documentation, the test run first initialize the database and then perform a sequence of queries. The change of access pattern in the middle of the sampling graph matches the change of behavior in this input. However, such changes are not predictable in general. Ding and Zhong showed that the cumulative reuse signature of these two programs are stable across different inputs [9]. It suggests a prediction strategy based on some statistical average. We do not consider this extension in this paper and will not discuss these two programs further.

3.2 Comparison with manual analysis

We hand-analyzed the major structure of each program and inserted phase markers (*manual markers*) based on our reading of the code and its documentation. Every phase represents complete logical (sub)-steps in the program function. We also used the help of profiling tool *gprof* to find important functions. The process is time consuming. In this section we evaluate whether the phase analysis can do what we did manually. We compare the overlap of automatically and manually inserted markers, the phase structure from the two analysis, and the accuracy of analysis for the largest program phases.

We compare the manual marking with the automatic marking as

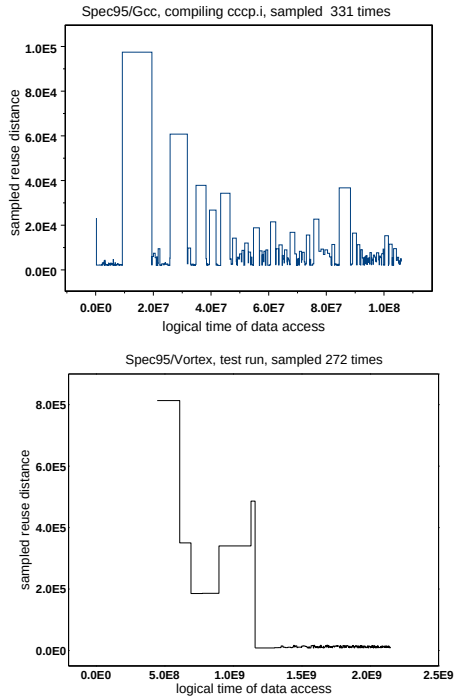


Figure 4: Sampled reuse distance trace of *Gcc* and *Vortex* from SPEC95Int. The phase behavior is not predictable even at run time.

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 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.

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

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

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

Table 4 shows comparison with manually inserted markers for detection and prediction runs. The columns for each run give the recall and 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 better recall occur more often in longer runs. Therefore, 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

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	78	52.0	0.667	2.2	78	62418.4	800.2	2712.0
Tomcatv	36	175.0	4.9	34.9	5251	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	802	863.66	1.98	27.39	3321.3	73938.1	146.5	779.58

Table 4: 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

Table 5: The number of largest phases

Benchmarks	Manual analysis	Phase analysis
FFT	30	30
Applu	350	350
Compress	25	25
Tomcatv	750	750
Swim	900	900
Mesh	50	50
MolDyn	30	30

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.

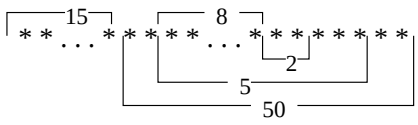
We manually constructed the main structures of the seven benchmarks. We now compare with the hierarchy structure of manually marked phases. Figure 5(a) shows the program structure of *Mesh*, which contains three nested loops. We use stars ($*$) for a phase marked manually. The number of the stars is given above the star sequence. The number of loop iterations is below the sequence. Figure 5(b) shows the predicted phase hierarchy. Each phase is labeled by the basic block number. The detected structure is the same as the program structure except the automatic analysis recognizes more basic phases than programmers. For all other phases, including the 50 time steps of the mesh simulation, the prediction captures the exact program behavior.

The largest phases at the top of the phase hierarchy is important in dynamic data reorganization because it gives the overall access pattern at run time. Table 5 compares the prediction accuracy for the largest phases. The prediction finds exactly the same number of largest phases (executed in the prediction run) as the manual analysis finds. Therefore, the analysis captures the top-level program phases and consequently enables fully automated data adaptation. Previous studies showed that such data adaptation can improve program performance by integer factors for a wide range of dynamic applications including scientific simulation, weather prediction, drug manufacture, and automobile design [8, 11, 19, 23].

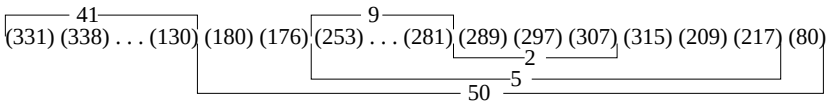
3.3 Application in cache adaptation

Phase prediction can also improve cache adaptation. Dynamically reducing the cache size has important benefits, including improving cache access time and reducing power density and energy consumption [1, 16]. We use a simplified model where cache has the same number of sets, 512, but can change from direct mapped to 8-way set associative. One cache block is 32 bytes. Hence, the cache can change size from 16KB to 128KB in 16KB intervals. The goal of this cache adaptation is to minimize the average cache size while maintaining program performance. For each part of an execution, it needs to find the smallest cache size such that the increase in the number of misses is no more than k% of the number of misses using the full cache size. The value of k gives the bound of performance loss, which we set to be 0% and 5% in our experiment.

A phase is a unit of predictable behavior and not necessarily a unit of uniform behavior. The best cache size for different parts may be different but the best size for the same part of the phase is

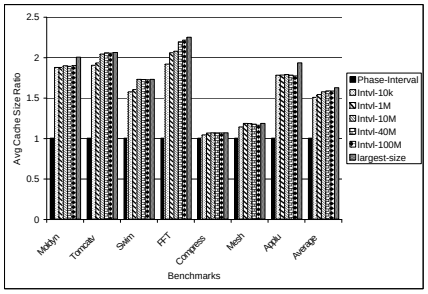


(a)

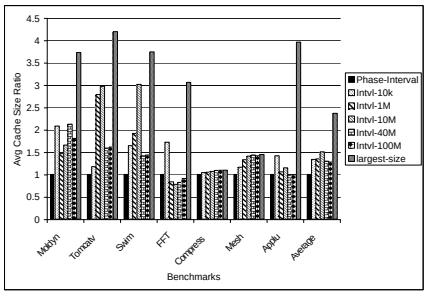


(b)

Figure 5: Comparison between the phase hierarchy and the program structure of *Mesh*. (a) The program structure. A star is a manually marked phase. The number of the stars is above the star sequence. The number of loop iterations is below the sequence. (b) The phase hierarchy. A phase is labeled by the number of the basic block number at its boundary.



(a) The average cache size found by different adaptation methods when the amount of cache misses does not increase.



(b) The average cache size found by different adaptation methods when the amount of miss increase is no more than 5%.

Figure 6: The effect of different adaptation methods on cache resizing

always the same across multiple occurrences. To capture the behavior change inside a phase, we divide it into fixed-length intervals of 10 thousand memory accesses (called phase intervals). The adaptation finds the best cache size during the first few phase executions and reuses it for later runs.

We compare phase-based adaptation with hardware interval-based methods. The latter divides an execution into fixed-length intervals using the number of cycles, instructions or other run-time measures. Based on the history, interval prediction guesses the behavior of the next interval. For cache resizing, Balasubramonian et al. predicted constant behavior between successive intervals [1]. We use the same prediction and measure with five different interval lengths: 10k, 1M, 10M, 40M, and 100M memory accesses. As discussed before, finding the right interval length is tricky because the best length may depend on programs, program inputs, and program phases. In addition, interval-based method needs to constantly monitor for the change in interval behavior. When a change is suspected, it needs to find the best cache size by exploration. The constant monitoring and the uncertain exploration may lead to sub-optimal results and exceed a given bound of the performance loss. In contrast, phase adaptation only explores during the first few occurrences of a phase and then reuse the optimal size wherever the phase is executed. Since it measures the performance with full cache size as part of the exploration, it can gauge the exact loss due to adaptation and guarantee any given bound on the performance loss.

The adaption finds the best cache size through run-time exploration. Since different exploration schemes have different costs, 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. In the third occurrence, we use the best cache size found through the simulation. In the experiment, we know the best cache size of each phase and interval by running it through *Cheetah*, a cache simulator that measures the miss rate of all eight cache sizes at the same time [24]. Phase adaption explores each phase only once. It always finds the best cache size because the behavior of phase repeats exactly. The exploration is negligible because of the many repetitions of the phases. The interval-based method needs to explore whenever a sufficient change is detected. It may not find the best cache size if the intervals used in exploration differ from the succeeding intervals. In fact, the interval-based method cannot easily guarantee a given bound on performance loss if the exploration happens

on every behavior change. We measure the best possible result of interval adaptation—we assume minimal cost in its exploration and perfect cache size in its adaptation. We compare the realistic result of phase adaption with the ideal result of interval adaptation.

Figure 6 shows the average cache size from phase adaptation and interval adaptation using five fixed interval lengths. Figure 6(a) shows the results of adaptation for no performance loss. The results are normalized to those of phase adaptation. The largest cache size, 128KB, is shown as the last bar in each group. For the same performance bound, different intervals find different cache sizes, but all intervals require a cache size much larger than phase adaptation does, at least 50% larger in five programs and on average 50% across all seven programs. In fact, most interval methods gain little improvement for the 0% bound case. The average cache size is at least 90% of the full cache size. The phase adaptation, however, reduces the cache size by half for most programs.

Figure 6(b) shows the results of adaptation for 5% performance loss in the same format as Figure 6(a). The effect of interval methods varies greatly. The 10M interval was 20% better than phase adaptation for *FFT* but 50% a factor of 2 or 3 worse for two other programs. The 100M interval is the best on average among all intervals. Still, it is about 50% worse for four programs and on average 30% worse than phase adaptation. Interval methods on average reduce cache size by 50%, while phase adaptation reduces it by another 20%.

The results show interesting variations across different programs. *Compress* has stable behavior so all interval adaptation performs well, and the smaller interval the better the result. Still, none outperforms phase adaptation. *FFT* has varied behavior, which causes the low coverage in no performance loss case. But its variation magnitude is so small that 5% performance loss can tolerate most of the variations, which explains the good performance of interval methods for *FFT* in Figure 6(b). As shown in section 3.1, *MolDyn* has uneven behavior. However, we can still reduce its cache size with cache miss rate increasing 0.003-0.006.

No single interval gives consistent improvement across all programs. In contrast, phase adaptation consistently perform better than all fixed intervals by at least 25% on average. In fact, it never loses except for *FFT* at the 5% bound. It shows convincingly that data locality characterizes program phase behavior better for memory adaptation than fixed length intervals.

Several studies used variable size intervals [2]. The range of possible interval lengths is $O(N)$ where N is the execution length. A run-time search cannot afford to examine many choices. One scheme is to double the interval length until the behavior is stable, which limits the search space to $O(logN)$ [2]. In comparison, our method considers all possible phase lengths and determines the phase length through reuse pattern analysis. At run time, it determine the change in phase length due to different program inputs. The flexibility and accuracy are important as shown by our results in Section 3.1 because the phase size differs greatly in different programs, the same program but different phases, or the same phase using different program inputs. The results in this section show the direct benefit in adaptation for a particular cache model.

Earlier studies used more complex models of cache and measured the effect on cycle time and energy consumption through cycle-accurate simulation. Since simulating the full execution would take too long, they either used 100 million instruction windows or reduced the program input size [1, 16]. Not needing a CPU simulator, we can run all programs to completion. In addition, the results are easier to be reproduced and compared with by other people.

4. RELATED WORK

This work is a unique combination of data analysis and phase prediction. It builds on the past work on locality analysis and the recent work in phase prediction. In this section we review related studies. Due to the limited space, we discuss only the empirical work based on real program traces rather than analytical models.

Locality and phase analysis Batson and Madison defined *activity set* as the set of data segments that are reused after the set is formed at the top of the LRU stack. Consequently, phases are "bounded locality intervals", which form a hierarchy where subphases access activity sub-sets [3]. Mattson et al. introduced stack distances for efficient locality measurement [18]. Sugumar and Abraham used stack distances in cache-miss classification [24]. These studies measure rather than predict program behavior. Ding and Zhong gave an approximation algorithm that measured LRU stack distance in effectively linear time [9]. They found predictable patterns in the overall locality but did not consider phase behavior.

Phalke and Gopinath predicted the time distance pattern (called inter-reference gap) using k order Markov chains and showed many uses of the phase information including better page replacement and trace compression [21]. Zhou et al. [25] and Jiang and Zhang [14] used the LRU stack distance for file caching and showed that the distance models adapted well to the access pattern in server and database traces and significantly outperformed LRU or frequency-based schemes. While on-line locality analysis works for virtual memory and file systems, it is too expensive for use in cache. A modern microprocessor makes possibly one billion cache accesses each second and cannot afford to analyze each memory access for locality. This work performs locality analysis through training runs. During execution, a program signals at the beginning and ending points of its phases, allowing cache to specialize for each phase using simple hardware or software controls. The novel techniques developed for this work—data sampling, hierarchy construction, and marker insertion—should also improve the stability, accuracy, and granularity of phase prediction in virtual memory and file systems.

Program region analysis Program interval analysis models a program as a hierarchy of intervals. For scientific programs, most computation and data access are in loop nests. Havlak and Kennedy showed that the inter-procedural array-section analysis accurately summarizes the program data behavior for dependence testing [12]. Recent work used program regions to control processor voltages to save energy. Hsu and Kremer defined a region as a sequence of straight-line code, which may span loops and functions [13]. They considered all possible program paths to ensure that a region is an atomic unit of execution under all program inputs. For general purpose programs, Huang et al. [16] and Magklis et al. [17] selected as phases functions whose number of instructions exceeds a threshold in a profiling run. These studies found the best voltage for program regions on a training input and then tested the program on a different input. They observed that different inputs did not affect the voltage setting. Huang et al. measured the effect of phase-based cache resizing in terms of the relative energy saving but did not report the average cache size reduction [16].

In contrast to other program analysis, our work finds phases by examining program data locality so that the analysis is largely independent of programming styles and program structures. For example, it can identify a time step in N-body simulation from many program regions or functions. It can mark a phase when the boundary points are not logically linked in a program. It can find large phases when the code is not straight-line. The extensive code reuse in modern programs makes straight-line code scarce, especially in object-oriented programs or space-optimized binaries. Our analysis, however, can discover repeated data traversals regardless whether it is coded in loops, recursive function calls, iterators, or templates. Fur-

thermore, our analysis accurately predict changes in locality due to different program inputs. For memory adaptation, program inputs are important because they change the data footprint of program regions or functions.

Array section analysis relies on compile-time knowledge of program data access and cannot analyze dynamic programs that use complex control flow and data indirection. Our analysis does not rely on compiler knowledge and handles sequential programs of arbitrary complexity. Its locality model has found patterns not only in general program traces [3, 9] but also in database and file access traces [14, 25].

Execution interval analysis The execution interval analysis divides an execution trace into fixed-length intervals using the number of cycles, instructions or other run-time measures. Based on the history, interval prediction guesses the behavior of the next interval. For cache resizing, Balasubramonian et al. predicted constant behavior between successive intervals [1]. A later extension dynamically doubled the interval size until reaching a steady state [2]. Three studies predicted changes in interval behavior using last value, Markov, or table-driven predictors [7, 10, 22]. Interval prediction works well if the interval length does not matter, for example, if an execution consists of long steady phases. Otherwise it is difficult to find the best interval length for a particular program on a particular input. If the interval is too small, the program behavior may vary too frequently. If the interval is too large, the analysis may miss phase changes inside intervals. The best interval length may depend on programs, program regions, and program inputs. Existing work used intervals of length from 100 thousand [1] to 10 million instructions [22] and executions from 10ms to 10 seconds [10]. For programs that are not amenable to program analysis, the interval-based method is the last resort because it is implemented completely in hardware and does not need any software support. However, for many important programs, our phase analysis can identify repeating phases of all sizes using program and locality information. The high accuracy and large granularity improve the effect of adaptation and protect it from causing unbounded slowdowns.

5. CONCLUSIONS

The paper presents a method for predicting hierarchical phases in both program instruction and data locality, an important novel aspect. This method predicts program executions thousands of times longer than profiling runs with 99.5% accuracy. It can recognize a phase with billions or trillions of instructions, allowing dynamic data optimization to be fully automated. When tested for cache adaptation, it reduces the cache size by 40% over hardware monitoring adaptation without increasing the number of cache misses. Its hierarchical structure spans a large range of phases, allowing different granularity for different uses. These results suggest that phase prediction can significantly improve the many adaptation techniques now used for increasing performance, reducing energy, and other improvements to the computer system design.

6. ACKNOWLEDGMENT

This work is supported by the National Science Foundation (Contract No. CCR-0238176, CCR-0219848, and EIA-0080124) and the Department of Energy (Contract No. DE-FG02-02ER25525).

7. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] H. Han and C. W. Tseng. Locality optimizations for adaptive irregular scientific codes. Technical report, Department of Computer Science, University of Maryland, College Park, 2000.

[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[13] 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.

[14] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Marina Del Rey, California, June 2002.

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

[16] 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.

[17] 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.

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

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

[20] 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.

[21] V. Phalke and B. Gopinath. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Ottawa, Ontario, Canada, 1995.

[22] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[23] 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.

[24] 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.

[25] Y. Zhou, P. M. Chen, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX Technical Conference*, June 2001.