Analysis of Input-Dependent Program Behavior Using Active Profiling

Xipeng Shen
College of William and Mary
Williamsburg, VA, USA
xshen@cs.wm.edu

Chengliang Zhang
University of Rochester
Rochester, NY, USA
zhangchl@cs.rochester.edu

Chen Ding
University of Rochester
Rochester, NY, USA
cding@cs.rochester.edu

Michael L. Scott
University of Rochester
Rochester, NY, USA
scott@cs.rochester.edu

Sandhya Dwarkadas
University of Rochester
Rochester, NY, USA
sandhya@cs.rochester.edu

Mitsunori Ogihara
University of Rochester
Rochester, NY, USA
ogihara@cs.rochester.edu

ABSTRACT

Utility programs, which perform similar and largely independent operations on a sequence of inputs, include such common applications as compilers, interpreters, and document parsers; databases; and compression and encoding tools. The repetitive behavior of these programs, while often clear to users, has been difficult to capture automatically. We present an active profiling technique in which controlled inputs to utility programs are used to expose execution phases, which are then marked, automatically, through binary instrumentation, enabling us to exploit phase transitions in production runs with arbitrary inputs. We demonstrate the effectiveness and programmability of active profiling via experiments with six utility programs from the SPEC benchmark suite; compare to code and interval phases; and describe applications of active profiling to memory management and memory leak detection.

Categories and Subject Descriptors

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

General Terms

Measurement, Performance, Algorithms

Keywords

program phase analysis and prediction, active profiling, memory management, dynamic optimization

1. INTRODUCTION

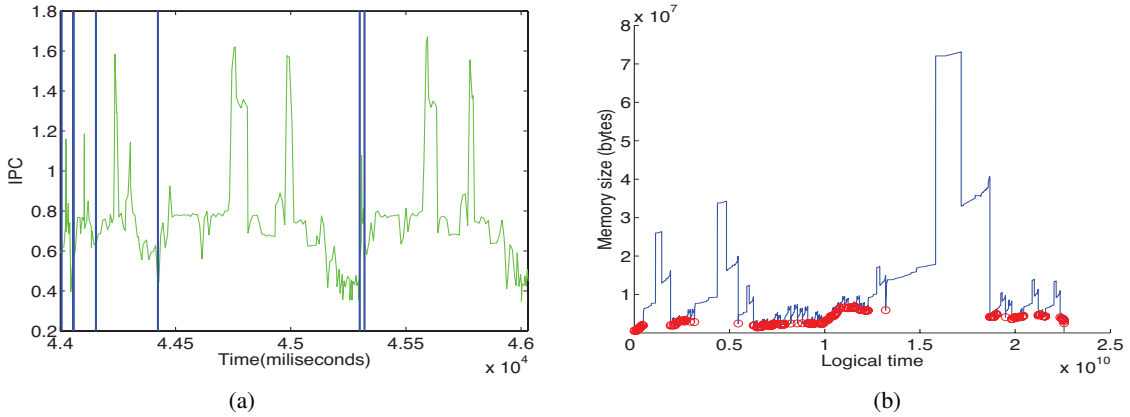
Complex program analysis has evolved from the static analysis of program invariants to include the modeling of behavior variation (of the same code) in different portions of the same run or across different runs. A principal problem for behavior analysis is dependence on program input. Changes in behavior induced by different inputs can easily hide those aspects of behavior that are uniform across inputs, and might profitably be exploited.

Programming environment tools, server applications, user interfaces, databases, and interpreters, for example, use dynamic data and control structures that make it difficult for current static analysis to predict run-time behavior, or for profile-based analysis to predict behavior on inputs that differ from those used in training runs. Yet at the same time, many of these programs have repetitive phases that users understand well at an abstract, intuitive level, even if they have never seen the source code. Many have the common feature that they accept, or can be configured to accept, a sequence of requests, each of which is processed more-or-less independently of the others. We refer to such programs as *utilities*. Program behavior differs not only across different inputs but also across different parts of the same input, making it difficult for traditional analysis techniques to find the phase structure embodied in the code. In many cases, a phase may span many functions and loops, and different phases may share the same code.

An example that we will refer to repeatedly is the GNU Gcc compiler. We use the version included in the SPEC CPU 2000 benchmark set [18]. The source program has 120 files and 222182 lines of C code. Figure 1(a) shows the program speed, measured in IPC (Instructions Per Cycle) by hardware counters on an IBM POWER4 machine, for part of the execution of *GCC* compiling the input *scilab*. Part (b) shows the memory usage, measured by the size of live data, of the same (full) execution. Visual inspection of both figures suggests that something predictable is going on: the IPC curve has multiple instances with two high peaks in the middle and a declining tail, and the memory usage curve is a series of mini-segments with an identical shape. However, the width and height of these features differs so much that an automatic technique may not reliably identify the pattern. The goal of our new analysis is to identify these repetitions as instances of the same phase, as marked for the *GCC* execution in the figure. The analysis should identify such phases in all executions of a program.

We introduce *active profiling*, which addresses the phase detection problem by exploiting the following observation: if we control the input to a utility program, we can often induce it to display an artificially regular pattern of behavior that is amenable to automatic analysis. Based on the regular pattern, the analysis identifies a set of candidate phase markers and then uses real inputs to filter out false positives and identify phase behavior representative of normal usage. Finally, the analysis is repeated to identify inner phases inside each outer phase. Active profiling combines input-level user control and code- and execution-level monitoring and pattern anal-

ExpCS 13–14 June 2007, San Diego, CA



(a) (b)

Figure 1: (a) IPC curve for part of the execution of *GCC* compiling the file *scilab*. Phase boundaries are shown as solid vertical lines. (b) The size of live data in the execution of *GCC* compiling *scilab*. Phase boundaries are marked with circles.

ysis. It is programmable—a user can design inputs to target specific aspects of program behavior. We will demonstrate this through an example in Section 4.3.

Many automatic techniques have been developed for phase analysis, as we will review in Section 6. Highly input-dependent programs challenge some of the basic assumptions in automatic techniques. For example, most profiling methods use a cut-off threshold to remove from consideration loops and procedures that contain too few instructions. If the input size may differ by many orders of magnitude, the threshold may easily be too high or too low for a particular run. In addition, previous techniques focus on CPU-centric metrics and look for recurring patterns. It is not always clear how to include higher-level phenomena like memory allocation and memory leaks in the analysis. In comparison, active profiling allows a user to target the analysis for specific behavior, it considers all program instructions as possible phase boundaries, and it uses multiple inputs to improve the results of the analysis.

Utility programs are the ideal target for this study because they are widely used and commercially important, and because users naturally understand the relationship between inputs and top-level phases. Our technique, which is fully automated, works on programs in binary form. No knowledge of loop or function structure is required, so a user can apply it to legacy code. Because users control the selection of regular inputs, active profiling can also be used to build specialized versions of utility programs for different purposes, breaking away from the traditional "one-binary-fits-all" program model.

We evaluate our techniques on five utility programs from the SPEC benchmark suites and show behavior variation within an execution and across different runs. We demonstrate the programmability of active profiling. We also compare behavior phases found through active profiling with phases based on static program structure (functions and loop nests) and on run-time execution intervals. Finally, we describe our earlier results on the use of behavior phases in improving the performance of garbage collection and detecting memory leaks [15].

2. TERMINOLOGY

We define a *program behavior phase* as follows. First, a phase is a unit of predictable behavior in that its instances, each of which is a continuous segment of program execution, are similar in some important respect. Note that different authors define "phase" in different ways. Our definition includes phases whose behavior can potentially be very nonuniform and whose length can vary by any degree. Some authors, particularly those interested in fine-grain architectural adaptation, define a phase to be an interval, often of bounded length, whose behavior is *uniform* in some important respect (e.g., instruction mix or cache miss rate).

A *phase marker* is a basic block in program code such that if it executes, an instance of the phase it marks must follow. Instances of different behavior phases do not overlap in time. They can nest, in which case we call the phase of the enclosing instance the *outer phase* and the phase of an enclosed instance an *inner phase*. Compilation, for example, is an outer phase that contains inner phases for parsing and semantic analysis, data flow analysis, register allocation, and instruction scheduling.

The goal of *phase detection* is to identify the phase markers in a program. The objective of *phase prediction* is to predict the behavior of a coming phase instance. A typical method is to use past instances of a phase to predict its future instances, hence the importance of similarity among the instances of a phase.

3. ACTIVE PROFILING

There are two main steps in active profiling: constructing regular inputs and selecting phase markers.

3.1 Constructing regular inputs

In utility programs, phases have variable length and behavior as shown for *GCC* in Figure 1. We can induce regularity, however, by issuing a sequence of identical (or nearly identical) requests—in *GCC*, by compiling a sequence of almost-identical functions. The resulting IPC curve is shown in Figure 2. Solid and broken vertical lines indicate outermost and inner phase boundaries that will be identified by our analysis. The fact that behavior repeats a predetermined number of times (the number of input requests) is critical to the analysis.

A utility program provides an interface through which to make requests and supply data. The interface can be viewed as a mini-language. It can be as simple as a command as in a shell program or a stream of bytes as in a data compression program. It can also be as complicated as a full-fledged programming language, as for example, in a Java interpreter or a processor simulator.

To produce a sequence of identical requests, we can often just repeat a request if the service is stateless such as file compression. Care must be taken, however, when the service stores information about requests. A compiler generally requires that all input functions in a file have unique names, so we replicate the same function but give each a different name. A database changes state as a result of insertions and deletions, so we balance insertions and deletions or use inputs containing only lookups.

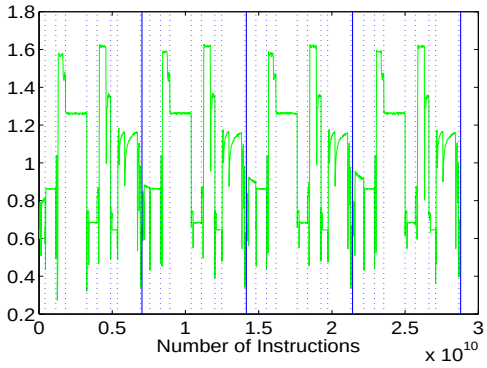


Figure 2: IPC curve of *GCC* on an artificial regular input, with instances of the outer phase (solid vertical lines) and instances of inner phases (broken vertical lines).

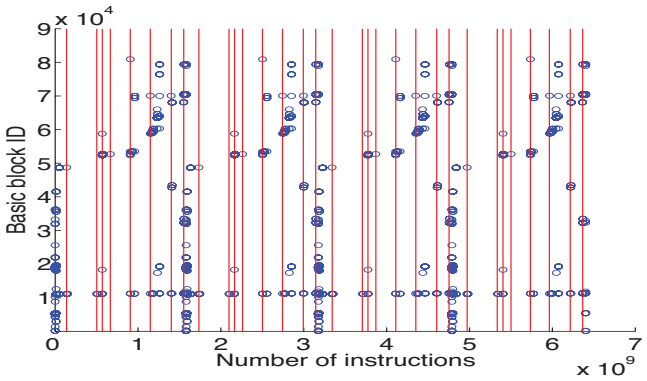


Figure 3: *GCC* inner-phase candidates with inner-phase boundaries.

The appropriate selection of regular inputs is important not only to capture typical program behavior, but also to target analysis at subcomponents of a program. For example, in *GCC*, if we are especially interested in the compilation of loops, we can construct a regular input with a single function that has nothing but a sequence of identical loops. Phase detection can then identify the inner phases devoted to loop compilation. By constructing special inputs, not only do we isolate the behavior of a sub-component of a service, we can also link the behavior to the content of a request. We will discuss the use of targeted analysis for a *Perl* interpreter in Section 4.3.

3.2 Selecting phase markers

The second phase of active profiling is fully automatic. It finds phase markers in three steps. The first step searches for candidate markers in the trace induced by a regular input. The second uses real inputs to remove false positives. Finally, the third step repeats the analysis to find inner phases.

Given a trace of basic blocks executed when running on a regular input of f identical requests, the first step selects only basic blocks that are executed exactly f times. Not all such blocks represent actual phase boundaries. A block may happen to be executed f times during initialization, finalization, memory allocation, or garbage collection. Next, the analysis checks whether the f instances of a basic block have an even distance between them. It measures the $f-1$ inter-occurrence distances for each block, calculates the mean and standard deviation of all distances, and discards blocks whose values are outliers (see the algorithm in Figure 4).

The remaining code blocks all have f evenly spaced occurrences, but still some may not be phase markers. In *GCC*, for example, the identical functions in the input may each contain a branch statement. Code to parse a branch may thus occur once per request with this input, but not with other inputs. In Step 2, the analysis checks whether a block occurs consistently in different inputs. It uses a real input containing g (non-identical) requests. It measures the execution frequency of the candidate blocks and keeps only those that are executed g times. Usually one real input is enough to remove all false positives, but this step can be repeated an arbitrary number of times to make use of all available inputs. At the end of Step 2, the analysis picks the first block in the remaining block sequence as the marker of the outermost phase.

The other blocks may mark interesting points *within* the outermost phase. In Step 3, the analysis selects inner phases of a non-trivial length and picks a marker for each inner phase. To illustrate we take the following trace as an example:

basic block trace:	...	8	10	...	9	100	...
logical time:	...	10	110	...	200	210	...
gap size:		100	...		10		

In the basic block trace, each number represents a basic block. The logical time is measured by the number of instructions executed. The size of a gap is the difference between the logical times of two adjacent basic blocks in the trace. Assuming 20 and 10 are the mean and standard deviation of the size of all the gaps in the trace, the gap between block 8 and block 10 is more than 3 standard deviations larger than the mean, so we select it as an inner phase instance and take block 8 as the marker for that inner phase. In comparison, the gap between block 9 and block 100 is too small to be considered an inner phase. Figure 3 shows a trace of *GCC* on regular input. Each circle on the graph represents an instance of a candidate inner phase marker. The vertical lines separate the inner phase instances.

4. EVALUATION

This section reports active profiling results on a set of five utility programs. It shows behavior variation both within a single run and across different runs. It gives an example of programmable phase analysis. It compares behavior phases with procedure and interval phases. Finally, it demonstrates two uses of behavior phases.

Table 1: Benchmarks.

Benchmark	Description	Source
Compress	UNIX compression utility	SPEC95Int
GCC	GNU C compiler 2.5.3	SPEC2KInt
LI	XLisp interpreter	SPEC95Int
Parser	natural language parser	SPEC2KInt
Vortex	object oriented database	SPEC2KInt
Perl	Perl interpreter	SPEC2KInt

4.1 Experimental settings

We test six utility programs (Table 4) from the SPEC 95 and 2K benchmark suites: a file compression utility, a compiler, two interpreters, a natural language parser, and an object-oriented database. Three other utility programs—two more compression utilities—exist in these suites. We have not yet experimented with them because they do not contribute a new application type. All test programs are written in C. Phase analysis is applied to the binary code.

Data Structure

$innerMarkers$	: the set of inner phase markers
$outerMarker$	: the outermost phase marker
$traceR$	: the basic block trace recorded in the regular training run
$traceI$	: the basic block trace recorded in the normal (irregular) training run
$RQSTR$	: the number of requests in the regular input
$RQSTI$	: the number of requests in the normal input
$setB$	: the set of all basic blocks in the program
$setB1, setB2, setB3$	: three initialy empty sets
b_i	: a basic block in $setB$
$timeR(b_i, j)$	: the instructions executed so far when b_i is accessed for the jth time
$V_i =< {V_i}^1, {V_i}^2, \ldots, {V_i}^k >$	: the recurring distance vector of basic block b_i in $traceR$, where ${V_i}^j = timeR(b_i, j+1) - timeR(b_i, j)$

Algorithm

step 1) Select basic blocks that appear $RQSTR$ times in $traceR$ and put them into $setB1$.
step 2a) From $setB1$, select basic blocks whose recurring distance pattern is similar to the majority and put them into $setB2$.
step 2b) From $setB2$, select basic blocks that appear $RQSTI$ times in $traceI$ and put them into $setB3$.
step 3) From $setB3$, select basic blocks that are followed by a long computation in $traceR$ before reaching any block in $setB3$ and put those blocks into $innerMarkers$; $outerMarker$ is the block in $innerMarkers$ that first appears in $traceR$.

Procedure Step2a()

// M and D are two initially empty arrays
for every b_i in $setB1$ {
V_i = GetRecurringDistances(b_i, $traceR$);
m_i = GetMean(V_i);
d_i = GetStandardDeviation(V_i);
M.Insert(m_i);
D.Insert(d_i); }
if (!IsOutlier(m_i,M) && !IsOutlier(d_i,D)) {
$setB2$.AddMember(b_i); }
End

Procedure IsOutlier(x, S)

// S is a container of values
m = GetMean(S);
d = GetStandardDeviation(S);
if ($|x - m| > 3 * d$) return true;
return false;
End

Figure 4: Algorithm for phase marker selection and procedures for recurring-distance filtering.

We construct regular inputs as follows. For *GCC* we use a file containing 4 identical functions, each with the same long sequence of loops. For *Compress*, which is written to compress and decompress the same input 25 times, we provide a file that is 1% of the size of the reference input in the benchmark suite. For *LI*, we provide 6 identical expressions, each containing a large number (34945) of identical sub-expressions. For *Parser*, we provide 6 copies of the sentence "John is more likely that Joe died than it is that Fred died." (That admittedly nonsensical sentence is drawn from the reference input and, not surprisingly, takes an unusually long time to parse.) The regular input for *Vortex* is a database and 3 iterations of lookups. Since the input is part of the program, we modify the code so that it performs only lookups but neither insertions nor deletions in each iteration.

We use ATOM [30] to instrument programs for the phase analysis on a decade-old Digital Alpha machine, but measure program behavior on a modern IBM POWER4 using its hardware performance counters, which are automatically read every 10ms. Not all hardware events can be measured simultaneously. We collect cache miss rates and IPC (in a single run) at the boundaries of program phases and, within phases, at 10ms intervals.

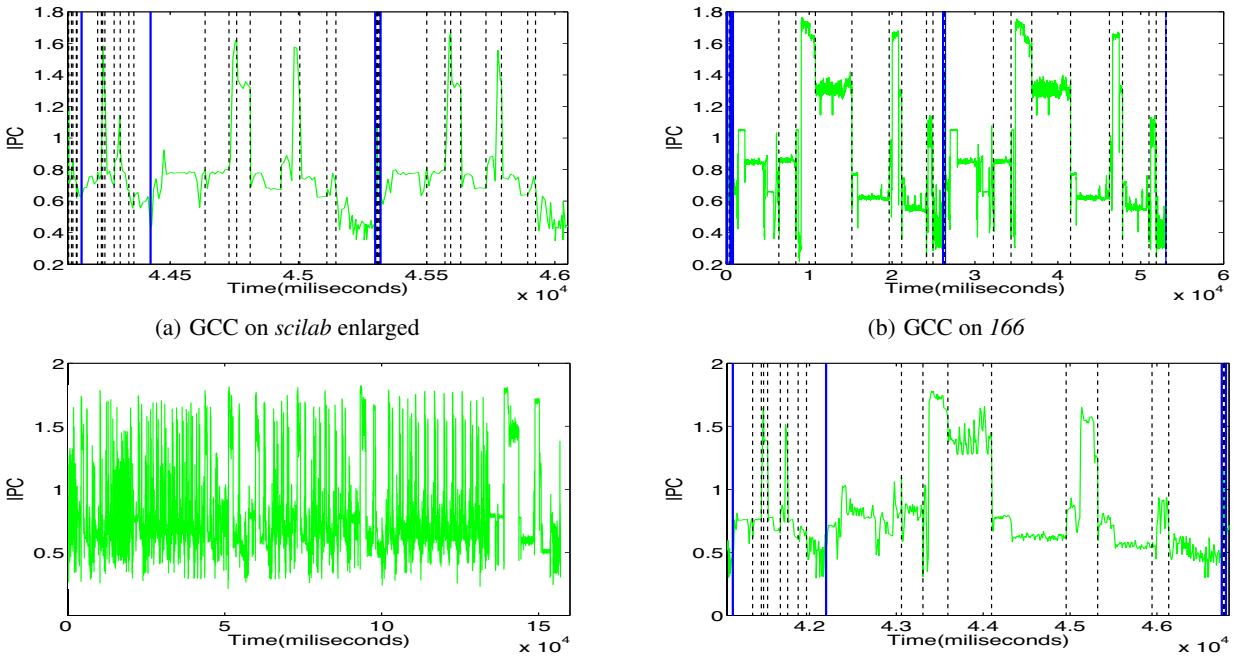
4.2 Behavior variation in utility programs

We first report *GCC*'s phases in detail as it is the most complex example. Then we briefly discuss the results of other programs.

GCC comprises 120 files and 222182 lines of C code. The phase detection technique successfully finds the outermost phase, which begins the compilation of an input function. We also find 9 inner phases. Though the analysis tool never considers source code, for the sake of curiosity we mapped the automatically inserted markers back into the source, where we confirmed that the markers separate standard compilation stages.

The first phase ends at the end of function "loop_optimize", which performs loop optimization on the current function. The second phase ends in the middle of function "rest_of_compilation", where the second pass of common sub-expression elimination completes. The third and fourth phases both end in function "life_analysis", which determines the set of live registers at the start of each basic block and propagates this information inside the basic block. The two phase markers are separated by an analysis pass, which examines each basic block, deletes dead stores, generates auto-increment addressing, and records the frequency at which a register is defined, used, and redefined. The fifth phase ends in function "schedule_insns", which schedules instructions block by block. The sixth ends at the end of function "global_alloc", which allocates pseudo-registers. The seventh ends in the same function as the fifth marker, "schedule_insns". However, the two phase markers are in different branches, and each invocation triggers one sub-phase but not the other. The two sub-phases are executed through two calls to this function (only two calls per compilation of a function), separated by the sixth marker in "global_alloc", among other function calls. The eighth phase ends in the middle of function "dbr_schedule", which places instructions into delay slots. The last phase covers the remainder of the compilation process. Given the complexity of the code, manual phase marking would be extremely difficult for someone who does not know the program well. Even for an expert in *GCC*, it might not be easy to identify sub-phases that occupy large portions of the execution time, of roughly equal magnitude.

The variation of IPC for three executions is shown in Figure 5. If we count the number of instances of the outermost phase, the input *scilab* has 246, the input *200* has 211, and the input *166* has just 11. We show the execution of the outermost phase marker by



(a) GCC on *scilab* enlarged

(b) GCC on *166*

(c) GCC on *200* without phase markers for legibility

(d) GCC on *200* enlarged

Figure 5: The IPC curves of different *GCC* runs with phase instances separated by solid vertical lines (for the outermost phase) and dotted vertical lines (for inner phases). The length of instances varies greatly but the length distribution will be similar as shown in Figure 6.

solid vertical lines and the execution of inner phase markers by dotted vertical lines. For legibility, we show the full execution of *200* without line marks and an enlarged segment with the lines.

Active profiling accurately captures the variation and repetition of program behavior, even when the shape of the curve is not exactly identical from instance to instance or from input to input. The analysis is done once using regular and real training inputs. The phase marking in other inputs does not require additional analysis.

The length of phase instances varies greatly both within the same input and across inputs. Figure 6 shows histograms of the length (on a logarithmic scale) and IPC (on a linear scale) for the 246 instances of the input *scilab.i* in the top two graphs, the 211 instances of the input *200.i* in the middle two graphs, and the 11 instances of the input *166.i* in the bottom two graphs. The execution length ranges from 6 million instructions to 10 billion instructions, while the IPC ranges between 0.5 and 1.0. The center of the distribution is similar, showing that most functions in the input file take around 30 million instructions to compile and have an IPC of 0.7.

Figure 6 shows that despite of the wide variation, the lengths and IPC have a similar distribution especially in the two inputs that have a large number of phase instances. This cross-input regularity can also be seen by the average IPC, shown in Figure 7. The IPC of 9 inner-phases ranges from below 0.5 to over 1.2 but the difference from input to input is less than 0.1 for the same sub-phase. The average of the whole execution is almost identical.

Next we show similar measurements for the other four programs we have tested. Figure 8 shows phase behavior in a Lisp interpreter, *LI*, and an English parser, *Parser*. Unlike *GCC*, the two programs do not have clear sub-phases with different IPC. The 271 phase instances of *LI* have highly varied length and IPC, but the 43 instances of *Parser* have mostly the same length and the same IPC, as shown by the histograms in Figure 8. Finally, Figure 9 shows the IPC curves of *Compress* and *Vortex*, with two sub-phases in the former and 13 sub-phases in the latter.

Without phase markers, it seems unlikely that one could recognize repeating behavior in these programs. We would almost certainly overlook similarity between periods of 6 million and 10 billion instructions. If we just measured the average behavior of the execution, we would effectively capture only the behavior of the few largest processing steps and not the behavior variation of all steps. Sampling-based techniques would be similarly dominated by the longest phases.

4.3 Programmability

Though our active analysis tool is usually employed in a fully automated form (the user provides a regular input and a few real inputs, and the tool comes back with an instrumented binary), we can invoke the sub-tasks individually to explore specific aspects of an application. This is one of the unique features of active profiling.

As an example, consider the *Perl* interpreter. The installed version in our system directory has 27 thousand basic blocks and has been stripped of all debugging information. Perl interprets one program at a time, so it does not have outermost phases as other programs do. In hopes of exploring how the interpreter processes function calls, however, we created a regular 30-line input containing 10 identical calls. Given this input, the regularity checking tool (step 1 of Section 3.2) identified 296 candidate marker blocks. We then created a 10-line irregular program containing three calls to two different functions. The consistency checking tool (step 2) subsequently found that 78 of the 296 candidates appeared consistently. Choosing one of these blocks at random (number 5410, specifically), we tested a third input, written to recursively sum the numbers from 1 to 10 in 11 calls. Block 5410 was executed exactly 11 times. This experience illustrates the power of active profiling to identify high-level patterns in low-level code, even when subsumed within extraneous computation.

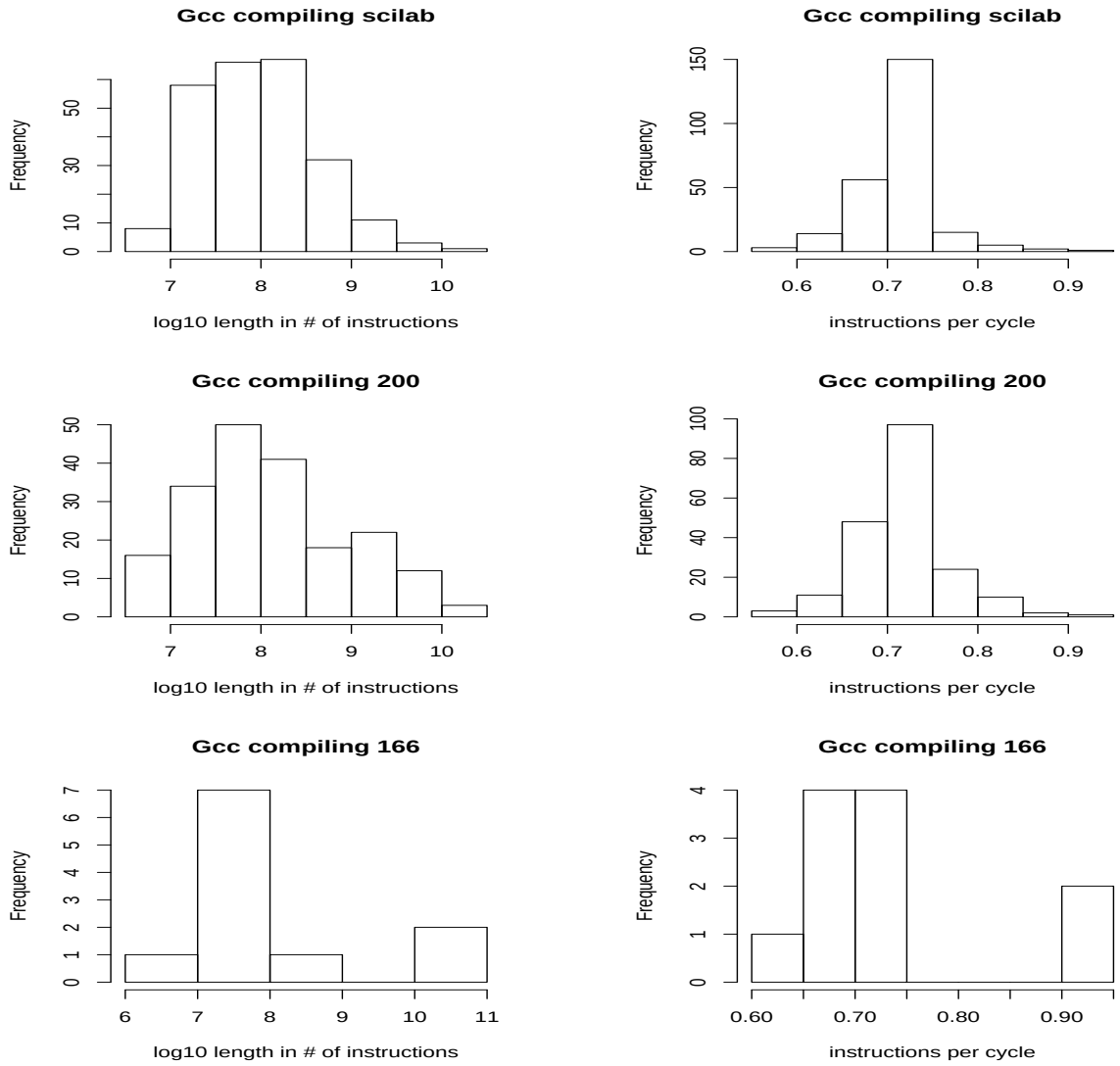


Figure 6: The number of instructions and the average IPC for the phase instances occurred in the three inputs, which have 246, 211, and 11 phase instances respectively. The numbers have similar ranges, and the distributions, although not identical, have their peaks in the same range.

A once-per-procedure marker in the interpreter could be used as a "hook" for various analyses related to procedures. It could, for example, enable subsequent analysis to measure time and memory overhead, monitor resource consumption, correlate results from different test runs, or localize the source of time, space, or other anomalies.

4.4 Comparison with procedure and interval phases

Program phase analysis takes a loop, subroutine, or other code structure as a phase [3, 17, 20, 23, 24, 25]. For the experiment reported here, we focus on procedure phases and follow the scheme of Huang et al., who picked subroutines using the thresholds θ_{weight} and θ_{grain} [20]. Assume the execution length is T. Their scheme picks a subroutine p as a phase if the cumulative time spent in p

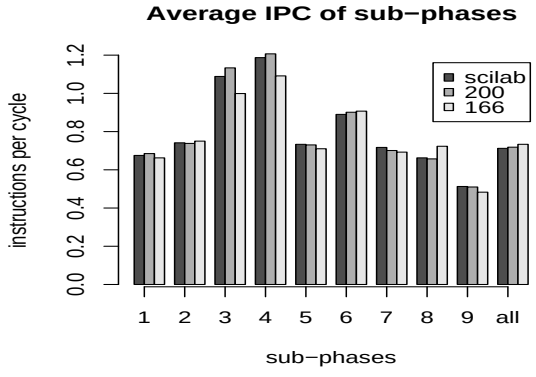
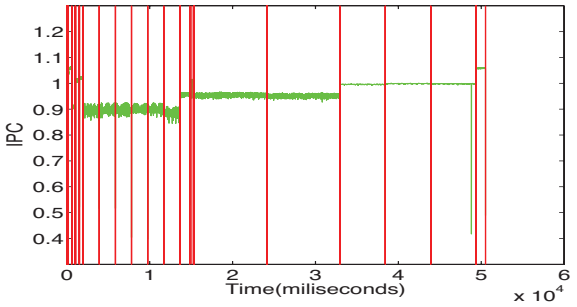
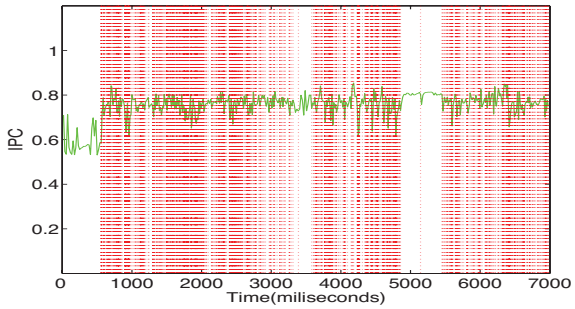


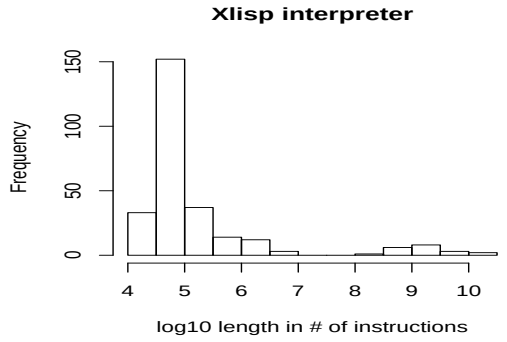
Figure 7: Average outer and inner phase IPC is consistent across three inputs.

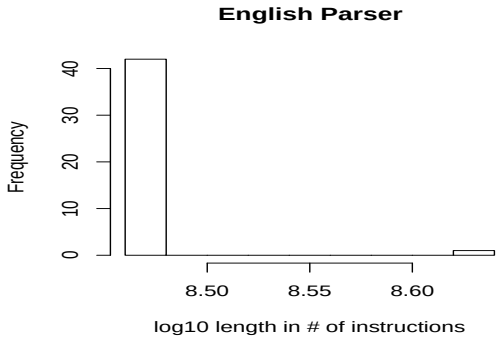


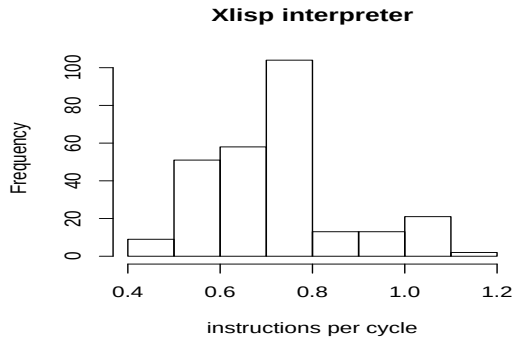
(a) XLisp Interpreter



(b) English Parser







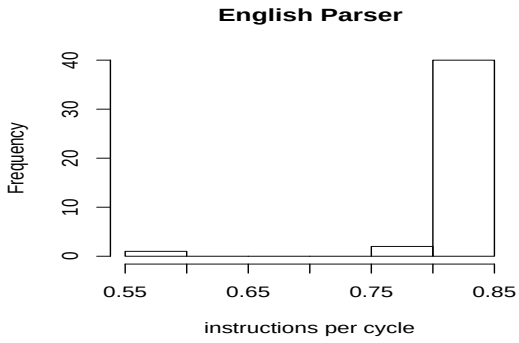
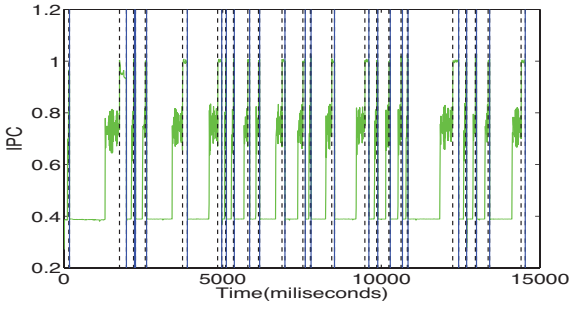
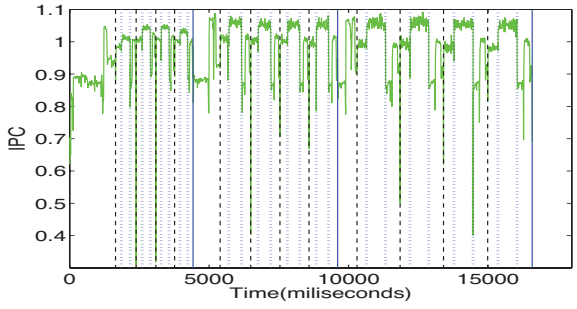


Figure 8: The IPC curves of the XLisp interpreter (*LI*) and the English parser (*Parser*) and the distribution of the instruction count and average IPC of 271 phase instances in *LI* and 43 phase instances in *Parser*.



(a) Compress



(b) Vortex

Figure 9: IPC curves of *Compress* and *Vortex* with phase markers.

(including its callees) is no less than $\theta_{weight}T$ and the average time per invocation no less than $\theta_{grain}T$. In other words, the subroutine is significant and does not incur an excessive overhead. Huang et al. used 5% for θ_{weight} and 10K instructions for $\theta_{grain}T$. Georges et al. made the threshold selection adaptive based on individual programs, the tolerable overhead, and the need of a user [17]. They studied the behavior variation of the procedure phases for a set of Java programs. Lau et al. considered loops and call sites in addition to subroutines, removed the code unit from consideration if the average size was below a threshold, and then selected code units whose behavior variation is within the average variation of all remaining code units [23].

Interval analysis divides an execution into fixed-size windows, classifies past intervals using machine or code-based metrics, and predicts the class of future intervals using last value, Markov, or table-driven predictors [3, 14, 16, 29]. Most though not all past studies (with the exception of [4]) use a fixed interval length for all executions of all programs, for example, 10 million or 100 million instructions.

Fully automatic techniques are rigid in that they are pre-programmed to look for specific patterns. Active profiling relies in part on the user input. As a result, it uses fewer empirically tuned thresholds—only the one commonly used for identifying outliers (described in Section 3.2). In our test set, the number of outermost phase instances ranges from 3 (corresponding to database queries) in *Vortex* to 850 (corresponding to input sentences) in *Parser*. We believe it is unlikely that an automatic method with a single set of thresholds could uncover phases with such wide variation in number. Without a priori knowledge about the number of phase instances, it is difficult to set a single appropriate value for θ_{grain} in procedure analysis or for interval length in interval-based analysis.

User control has two other important benefits. First, active profiling can be used to target specific components of a program. Second, it can identify phases in memory reference behavior that have no simple patterns. On the other hand, active profiling needs the manual efforts of a user to provide regular inputs. It is not suitable for the uses when full automation is desired.

We should note that the extension to procedure analysis by Lau et al. may improve the predictability of procedure phases. In fact, the phase markers found by the method of Lau et al. [23] for *GCC* include some of the markers found by active profiling. However, they found more phases (30 in *166*), and the markers were executed with a different frequency.

A quantitative comparison.

While behavior may vary widely within a phase, *average* behavior across all instances of a phase should be highly uniform. In our experiments we compare the cache hit rate and IPC of phase instances identified by different methods. Specifically, we compute the *coefficient of variation (CoV)* among phase instances. CoV measures how widely spread a normal distribution is relative to its mean, calculated as the standard deviation divided by the mean. If one uses the average of a phase's instances as its behavior prediction, the CoV is the expected difference between the prediction and the actual value of each phase.

It should be noted that different metrics—and thus different analysis techniques—may be appropriate for other forms of optimization (e.g., fine-grain tuning of dynamically configurable processors). Better prediction accuracy on some metrics does not imply a better program or a better system. Depending on the use, one type of phase may be better than another type.

The comparison includes adaptive profiling, the implementation of procedure analysis following [20], and an implementation of interval-based analysis. For purposes of comparison, we select the interval length for each program in our experiments so that the total number of intervals equals the number of inner behavior phase instances identified by active profiling. Instead of using specific interval phase methods, we calculate the upper bound across all possible methods using an optimal partitioning, approximated by k-means clustering. We measure most of the metrics using hardware counters on an IBM POWER4 machine. Since the counter results are not accurate for execution lengths shorter than 10ms, we excluded phase instances whose lengths are below 10ms.

Unlike the other methods, the results for procedure phases are obtained via simulation. Since some of the procedures are library routines, we would require binary instrumentation to obtain equivalent results from hardware counters. We use simulation because we lack an instrumentation tool for the IBM machine.

Figure 10 contains the CoVs of cache hit rates and IPCs.[1] Each program is represented by a group of floating bars. Each bar shows the CoV of a phase analysis method. When a program has multiple inner phases, the two end points of a bar show the maximum and minimum and the circle shows the average. The three potential bars in each group show the CoVs of behavior phases, procedure phases, and intervals with k-means clustering (the best possible prediction given the number of clusters).

Among all the approaches, adaptive profiling provides phases with the smallest average variations for 3 programs, *Compress*, *LI*, and *Parser*, on both cache hit rates and IPC. Its results on the other two programs, *GCC* and *Vortex*, are within 3% of the best ones from the clustered interval method. However, the variation range of the phases from the clustered interval method is much larger than that from active profiling for *GCC*. On average across all programs, the phases from active profiling have the smallest variation, while the basic interval method has the largest. Procedure-based phases show the largest variation ranges.

It is worth noting that *Compress* has two sub-phases, whose cache hit rates are always about 88% and 90% respectively. However, because different instances have different lengths, when divided into fixed-length intervals, the smallest and average CoV from interval phases are 0.7% and 0.9%, much larger than those of behavior phases, which are 0.15% and 0.21%. The reason that interval methods beat adaptive profiling on *Vortex* is because the interval length happens to suit this program.

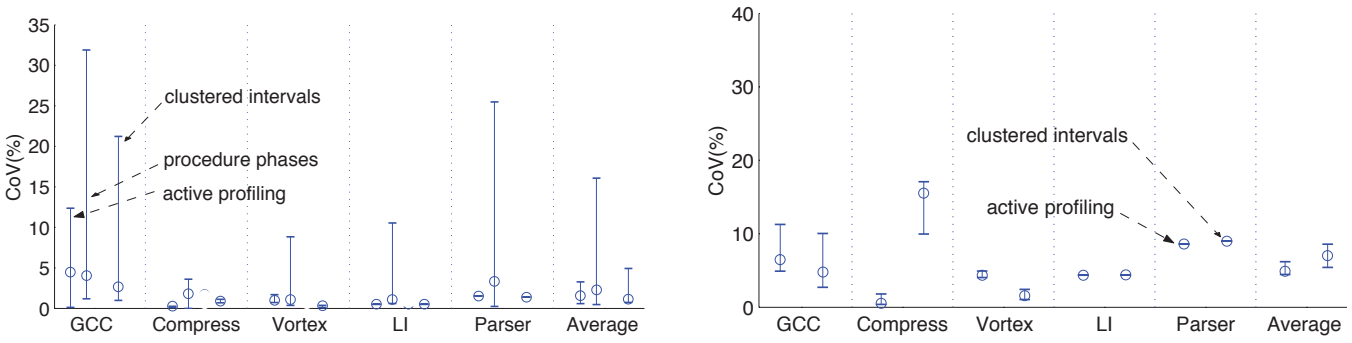
5. USES OF BEHAVIOR PHASES

Active profiling allows a programmer to analyze and improve high-level behavior of a program. In this section, we describe our preliminary results on preventive memory management and memory leak detection.

5.1 Preventive memory management

A behavior phase of a utility program often represents a memory usage cycle, in which temporary data are allocated in early parts of a phase and are dead by the end of the phase. This suggests that garbage collection will be most efficient when run at the end of a behavior phase, when the fraction of memory devoted to garbage is likely to be highest. Conversely, garbage collection should run in the middle of a phase only if the heap size reaches the hard upper bound on the available memory. This "preventive" scheme differs from typical reactive schemes, which invoke garbage collection (GC) when the heap size reaches a soft upper bound. By using

[1]We do not include the procedure-based method for IPC since it is based on simulation and therefore could not be directly compared to the real measurement of IPCs in the other three cases.



(a) Cache hit rate CoV

(b) IPC CoV

Figure 10: Behavior consistency of three types of phases, calculated as the coefficient of variance among instances of each phase. For each program, the range of CoV across all inner phases is shown by a floating bar where the two end points are maximum and minimum and the circle is the average. A lower CoV and a smaller range mean more consistent behavior. Part (b) shows the CoV of IPC.

phase information, preventive GC adapts to the natural needs of an application without requiring empirical thresholds.

We have implemented preventive garbage collection, applied it to the Lisp interpreter *LI*, and tested the performance on an Intel Pentium 4 workstation (2.8 GHz CPU, 1GB RAM). We used both the training and the reference inputs. The execution time of the entire program is shown in Table 2. Using preventive GC, the program outperforms the version using reactive GC by 44% for the reference input and a factor of 3 for the training input. For the reference input, the faster execution time is due mainly to fewer GC passes. Preventive GC passes are 3 times fewer than reactive ones for the training input and 111 times fewer for the reference input.

To be fair, we note that the (111 times) fewer GC passes in the reference input leads to (on average 36 times) more memory usage, as shown by the column "avg" in Table 2. Existing reactive garbage collectors may yield similar performance by giving the program as large a heap size. Still, the preventive scheme is simpler because it does not use empirically tuned thresholds. It is based on the high-level program behavior pattern. The training input shows an intriguing potential—the program runs 50% faster with preventive GC (despite its 47 collection calls) than it does without any GC. The faster execution is achieved with less than half of the memory, possibly due to better locality as a result of preventive GC.

The garbage collection hints by Buytaert et al. work in a similar way [9]. They use procedure-based phase analysis to insert GC hints at the Java method local minima in term of live data size. The live-data curves of the programs they used show a regular recurring pattern, which is not the case for the programs in this work. We believe that active profiling can be used to augment their system and to improve garbage collection for a broader class of applications.

5.2 Memory leak detection

Given that a behavior phase often represent a memory usage cycle, we apply it for memory leak detection. Through profiling, our analysis identifies all allocation sites that contribute to the long-term memory increase. We classify dynamic objects as either *phase local*, if their first and last accesses are within an (outermost) phase instance, *hibernating*, if they are used in a phase instance and then have no use until the last phase instance of the execution, or *global* if they are used by multiple phase instances.

If a site allocates only phase-local objects during profiling, and if not all its objects are freed at the end a phase, it is likely that the remaining objects are memory leaks. If a site allocates only hibernating objects, it is likely that we can avoid storing the object for more than one phase either by moving the last use early or by storing the object to disk and reloading it at the end. Less invasively, we can group objects that are likely to have no accesses for a common long period of time and place them on the same virtual memory page. Object grouping does not reduce the size of live data but it reduces the amount of physical memory usage because the unused page can be stored to disk by the operating system. Object grouping may also reduce energy consumption without degrading performance if memory pages can be placed in sleep mode.

Following is a sample report that shows a dynamic memory allocation site identified by its call stack. The middle line shows the number and size of freed and unfreed objects allocated from this site and the bottom part shows the object classes. This site allocates 18 objects, with 14 of them reclaimed later. Since all 18 objects are phase local in this execution, it is likely that the 4 remaining objects are memory leaks. Reclaiming these four objects would save 16KB memory without affecting the correctness of this execution. After testing many inputs, we found that all objects reclaimed by GCC are phase local objects. If an object is not reclaimed at the end of its creation phase instance, it will not be reclaimed by the program.

```
alloc. site: 44682@xmalloc<149684@_obstack_
             newchunk<149684@rtx_alloc<
             155387@gen_rtx<352158@gen_jump<
             84096@proc_at_0x120082260<
             83994@expand_goto<4308@yyparse<
             45674@proc_at_0x12005c390<
             48606@main<390536@__start<
4/16288 unfreed, 14/57008 freed.
                  freed          unfreed
phase local   14/     57008    4/     16288
hibernating    0/         0    0/         0
     global    0/         0    0/         0
```

Monitoring techniques have been used by both research and commercial systems to detect memory leaks. Chilimbi and Hauswirth developed sampling-based on-line detection, where an object is considered a possible leak if it is not accessed for a long time [10]. Bond and McKinley used one-bit encoding to trace leaked objects to their allocation sites [7]. Phase analysis can help these and other techniques by enabling them to examine not just physical time but also the stage of the execution. This is especially useful for utility programs because the length of phase instances may differ by orders of magnitude. In addition, phase-based techniques may help to recognize and separate hibernating objects, to reduce the size of active memory a program needs, and consequently to improve data locality and reduce resource and energy demand. Like all profiling techniques, of course, this method is incomplete: it cannot detect

Table 2: Comparison of the execution time between preventive and reactive GC

program inputs	GC methods	exe. time (sec) Pentium 4	per-phase heap size (1K)		total GC calls	total mem. (1K) visited by GC
			max	avg		
ref.	preventive	13.58	16010	398	281	111883
	reactive	19.5	17	11	31984	387511
	no GC	12.15	106810	55541	0	0
train	preventive	0.02	233	8	47	382
	reactive	0.07	9	3	159	1207
	no GC	0.03	241	18	0	0

leaks that don't appear in the profiling runs.

6. RELATED WORK

The key uniqueness of active profiling is that it connects users' high-level understanding of a program with automatic phase analysis by using controlled inputs. As a result, it is independent of parameters, customizable to users' needs, and robust to the complexity of behavior patterns. We discuss its detailed differences from other phase analysis below.

Locality phases Early phase analysis was aimed at virtual memory management and was intertwined with locality analysis. In 1976, Batson and Madison defined a phase as a period of execution accessing a subset of program data [6]. Bunt et al. measured the change of locality as a function of page sizes (called the locality curve) in hand marked hierarchical phases [8]. Using the PSIMUL tool at IBM, Darema et al. showed recurring memory-access patterns in parallel scientific programs [13]. These studies did not predict locality phases. Later studies used time or reuse distance as well as predictors such as Markov models to improve virtual memory management. Shen et al. used reuse distance to model program behavior as a signal, applied wavelet filtering, and marked recurring phases in programs [28]. For this technique to work, programs must exhibit repeating behavior. With active profiling, we are able to target utility programs, whose locality and phase length are typically input-dependent, and therefore not regular or uniform.

Program phases Balasubramonian et al. [3], Huang et al. [20, 24], and Magklis et al. [25] selected as program phases procedures, loops, and code blocks whose number of instructions exceeds a given threshold during execution. For Java programs, Georges et al. selected as phases those procedures that display low variance in execution time or cache miss rate [17]. It is not easy for a method that uses fixed thresholds to determine the expected size or behavior variance for phases of a utility program when one has no control over the input. For example, instances of a compilation phase may have very different execution length and memory usage.

Lau et al. considered loops, procedures, and call sites as possible phase markers if the variance of their behavior is lower than a relative threshold, which is the average variance plus the standard deviation [23]. The technique may capture regular repetitions more efficiently than the wavelet analysis [28]. The use of a relative threshold makes it flexible enough to capture phases with an input dependent length. It is also fully automatic. The analysis is general, but it does not target specific behavior cycles such as high-level data reuse [28] and user specified behaviors. Active profiling relies on user input but also permits a user to target specific behavior, for example, to find a code marker that signals the interpretation of a loop in a Perl program.

Allen and Cocke pioneered interval analysis to model a program as a hierarchy of regions [2]. Hsu and Kremer used program regions to control processor voltages to save energy. Their regions may span loops and functions and are guaranteed to be an atomic unit of execution under all program inputs [19].

In comparison to the above techniques, active profiling does not rely on the static program structure. It considers all program statements as possible phase boundaries. We found that in *GCC*, some sub-phase boundaries were methods called inside one branch of a conditional statement. In addition, active profiling permits a user to target specific behavior such as data reuse and memory allocation cycles. Finally, active profiling examines multiple inputs to improve the quality of code markers.

Interval phases Interval methods divide an execution into fixed-size windows, classify past intervals using machine or code-based metrics, and predict the behavior of future intervals using last value, Markov, or table-driven predictors (e.g., [3, 14, 16, 29]). Balasubramonian et al. [4] dynamically adjust the size of the interval based on behavior predictability/sensitivity. However, since the intervals don't match phase boundaries, the result may be an averaging of behavior across several phases. Duesterwald et al. gave a classification of these schemes [16]. Nagpurkar et al. proposed a framework for online phase detection and explored the parameter space [26]. A fixed interval may not match the phase length in all programs under all inputs. Our technique finds variable-length phases in utility programs. It targets program level transformations such as memory management and parallelization, so it is designed for different purposes than interval phase analysis is.

Training-based analysis Balasundaram et al. used microkernels to build a parameterized model in a method called training sets [5]. The model was used to select data partitioning schemes. Ahn and Vetter analyzed various program behavior metrics through statistical analysis including two types of clustering, factoring, and principle component analysis [1]. Different metrics were also correlated with linear models (with non-linear components) by Rodriguez et al. [27], queuing models by Jacquet et al. [22], and (logical) performance predicates by Crovella and LeBlanc [11, 12]. Ipek et al. combined observations automatically into predictive models by applying general multilayer neural networks on tens to thousands of training results [21]. Their model also predicts the accuracy of the prediction. Active profiling associates program points with input-dependent behavior and may use the existing models to predict the effect of different inputs.

7. CONCLUSIONS

The paper has presented active profiling for phase analysis in utility programs, such as compilers, interpreters, compression and encoding tools, databases, and document parsers[2]. Using deliberately regular inputs, active profiling exposes top-level phases, which are then marked via binary instrumentation and verified with irregular inputs. The technique requires no access to source code, no special hardware support, no user knowledge of internal application

[2]A Linux version of the software is available at http://www.cs.wm.edu/ xshen/Software/ActiveProf

structure, and no user intervention beyond the selection of inputs. By reliably marking large-scale program phases, active profiling enables the implementation of promising new program improvement techniques, including preventive garbage collection (resulting in improved performance relative to standard reactive collection) and memory leak detection.

Beyond the realm of behavior characterization and memory management, we have used active profiling to speculatively execute the phases of utility programs in parallel, obtaining nontrivial speedups from legacy code. As future work, we hope to explore additional optimizations, and to identify additional classes of programs amenable to profiling with intentionally crafted inputs.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments, which helped improve the presentation of the paper. We also thank Michael Huang, Brad Calder, and Jeremy Lau for answering our questions about their papers. This work was supported in part by National Science Foundation grants EIA-0080124, CCR-0204344, ECS-0225413, CCR-0238176, CNS-0411127, CNS-0509270, CNS-0615139, and CCF-0702505; a CAS fellowship from IBM; a grant from Microsoft Research; and equipment support from IBM, Intel, and Sun.

9. REFERENCES

[1] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of SC*, pages 1–16, 2002.

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

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd International Symposium on Microarchitecture*, Monterey, California, December 2000.

[4] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[5] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, Apr. 1991.

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

[7] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, 2006.

[8] R. B. Bunt, J. M. Murphy, and S. Majumdar. A measure of program locality and its application. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 1984.

[9] D. Buytaert, K. Venstermans, L. Eeckhout, and K. D. Bosschere. Garbage collection hints. In *Proceedings of The HiPEAC International Conference on High Performance Embedded Architectures and Compilation*, November 2005.

[10] T. M. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, USA, October 2004.

[11] M. Crovella and T. J. LeBlanc. Parallel performance using lost cycles analysis. In *Proceedings of SC*, pages 600–609, 1994.

[12] M. E. Crovella and T. J. LeBlanc. The search for lost cycles: A new approach to parallel program performance evaluation. Technical Report 479, Computer Science Department, University of Rochester, December 1993.

[13] F. Darema, G. F. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, May 1987.

[14] 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.

[15] C. Ding, C. Zhang, X. Shen, and M. Ogihara. Gated memory control for memory monitoring, leak detection and garbage collection. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Memory System Performance*, Chicago, IL, June 2005.

[16] 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.

[17] A. Georges, D. Buytaert, L. Eeckhout, and K. D. Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2004.

[18] J. Henning. Spec2000: measuring cpu performance in the new millennium. *IEEE Computer*, 2000.

[19] C.-H. Hsu and U. Kremer. The design, implementation and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[20] M. Huang, 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.

[21] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Proceedings of the Euro-Par Conference*, pages 196–205, 2005.

[22] A. Jacquet, V. Janot, C. Leung, G. R. Gao, R. Govindarajan, and T. L. Sterling. An executable analytical performance evaluation approach for early performance prediction. In *Proceedings of the International Parallel and Distribute Processing Symposium*, 2003.

[23] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *Proceedings of International Symposium on Code Generation and Optimization*, March 2006.

[24] W. Liu and M. Huang. Expert: Expedited simulation exploiting program behavior repetition. In *Proceedings of International Conference on Supercomputing*, June 2004.

[25] 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.

[26] P. Nagpurkar, M. Hind, C. Krintz, P. F. Sweeney, and V. Rajan. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2006.

[27] G. Rodríguez, R. M. Badia, and J. Labarta. Generation of simple analytical models for message passing applications. In *Proceedings of the Euro-Par Conference*, pages 183–188, 2004.

[28] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, Boston, MA, 2004.

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

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