Analysis of Input-Dependent Program Behavior Using Active Profiling [*]

Xipeng Shen[†] Chengliang Zhang[‡] Chen Ding[‡]
Michael L. Scott[‡] Sandhya Dwarkadas[‡] Mitsunori Ogihara[‡]

[†]Computer Science Department, The College of William and Mary
xshen@cs.wm.edu
[‡]Computer Science Department, University of Rochester
{zhangchl,cding,sandhya,scott,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. Experiments with five programs from the SPEC benchmark suites show that phase behavior is surprisingly predictable in many (though not all) cases. This predictability can in turn be used for optimized memory management leading to significant performance improvement.

1. Introduction

Complex program analysis has evolved from the static analysis of source or machine code to include the dynamic analysis of behavior across all executions of a program. We are particularly interested in patterns of memory reference behavior, because we can use these patterns to improve cache performance, reduce the overhead of garbage collection, or assist memory leak detection.

A principal problem for behavior analysis is dependence on program input. Outside the realm of scientific computing, 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 or impossible 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.

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. A C compiler, SPEC CPU2000 GCC for example, has a phase in which it compiles a single input function [17]. It runs this function through the traditional tasks of parsing and semantic analysis, data flow analysis, register allocation, and instruction scheduling, and then repeats for the following function.

Most of the applications mentioned above, as well as compression and transcoding filters, have repeating *behavior phases*, and often subphases as well. We refer to such programs as *utilities*. They 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. 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.[1]

Figure 1(a) shows the IPC (Instruction Per Cycle) curve of GCC on input *scilab*, which comprises the compilation of 274 C functions. Figure 1(b) shows a zoomed view. Vertical lines in the graph indicate phase boundaries, separating the compilations of different functions. The Figure suggests that something predictable is going on: IPC in each phase instance has two high peaks in the middle and a declining tail. But the width and height of these features differs so much that an automatic technique may not reliably identify the pattern[30]. In addition, the phase boundaries found for one execution may not exist in executions from other inputs.

Often a user is interested in the memory usage of an application. Figure 2 illustrates an opportunity provided by behavior phases. Though the volume of live data in the compiler may be very large

[*] This work was supported in part by National Science Foundation grants EIA-0080124, CCR-0204344, CCR-0219848, ECS-0225413, CCR-0238176, and CNS-0411127; Department of Energy grant DE-FG02-02ER25525; and equipment or financial grants from IBM, Intel, and Sun.

[1] Note that different authors define "phase" in different ways. We use it to refer to a span of program execution whose behavior, while potentially very nonuniform, is *predictable* in some important respect, typically because it resembles the behavior of some other execution span. Some authors, particularly those interested in fine-grain architectural adaptation, define a phase to be an interval whose behavior is *uniform* in some important respect (e.g., instruction mix or cache miss rate).

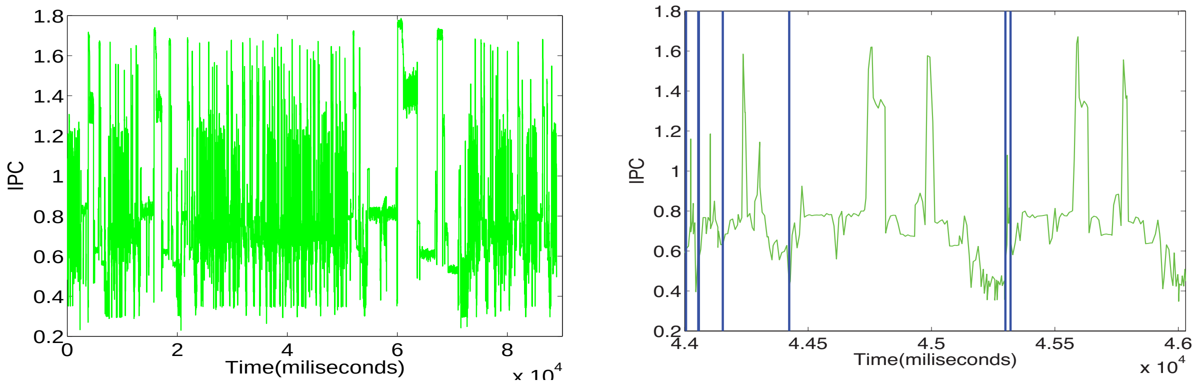


Figure 1. (a) IPC curve of *GCC* on input *scilab* and (b) an enlarged random part. Compilation boundaries are shown as solid vertical lines.

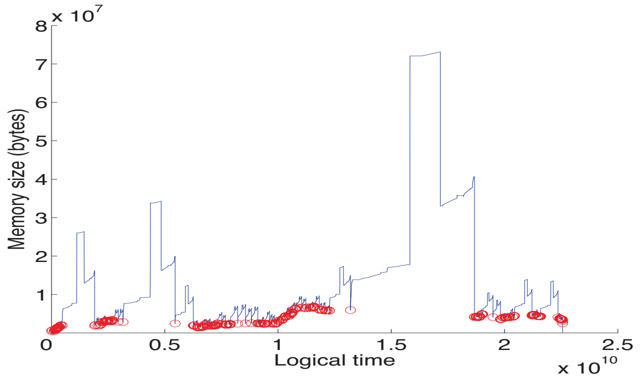


Figure 2. The curve of the minimal size of live data during the execution of GCC on input *scilab* with a circle marking the beginning of the compilation of a C function. Logical time is defined as the number of memory accesses performed so far.

while compiling an individual function, it always drops to a relatively low value at function compilation boundaries. The strong correlation between phases and memory usage cycles suggests that the phase boundaries are desirable points to reclaim memory, measure space consumption to predict memory usage trends, and classify object lifetimes to assist in memory leak detection. Automatic analysis is difficult because phases differ greatly in both length and memory usage.

In this paper 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 force it to display an artificially regular pattern of behavior that exposes the relationship between phases and fragments of machine code. Active profiling uses a sequence of identical requests to induce behavior that is both representative of normal usage and sufficiently regular to identify outermost phases (defined in Section 2.1). It then uses different real requests to capture inner phases and to verify the representativeness of the constructed input. In programs with a deep phase hierarchy, the analysis can be repeated to find even lower level phases. We can also design inputs to target specific aspects of program behavior, for example, the compilation of loops.

Many automatic techniques have been developed for phase analysis, as we will review in Section 5. 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. The analysis shows different types of behavior variation in these commonly used programs. We also compare with phases based on static program structure (functions and loop nests) and on run-time execution intervals. Finally, we demonstrate the use of phase information to monitor memory usage, improve the performance of garbage collection, and detect memory leaks.

2. Active Profiling and Phase Detection

2.1 Terminology

Program phases have a hierarchical structure. For utility programs, we define an *outermost phase* as the processing of a request, such as the compilation of a function in a C compiler, the compression of a file in a file compressor, or the execution of a query on a database. An *inner phase* is a computation stage in the processing of a request. Compilation, for example, typically proceeds through parsing and semantic analysis, data flow analysis, register allocation, and instruction scheduling. A *phase marker* is a basic block that is always executed near the beginning of that phase, and never otherwise.

2.2 Constructing regular inputs

In utility programs, phases have variable length and behavior as shown in Figure 1. We can force regularity, however, by issuing a sequence of identical requests—in GCC, by compiling a sequence of identical functions, as shown in Figure 3. Solid and broken ver-

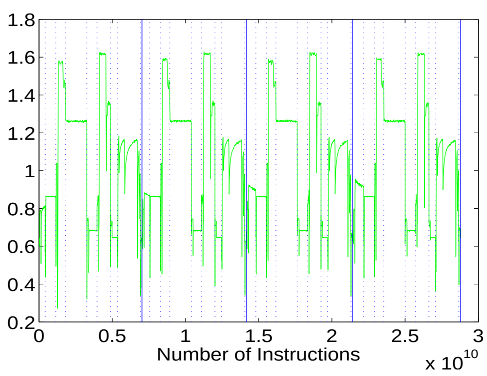


Figure 3. IPC curve of *GCC* on an artificial regular input, with top-level (solid vertical lines) and inner-level (broken vertical lines) phase boundaries.

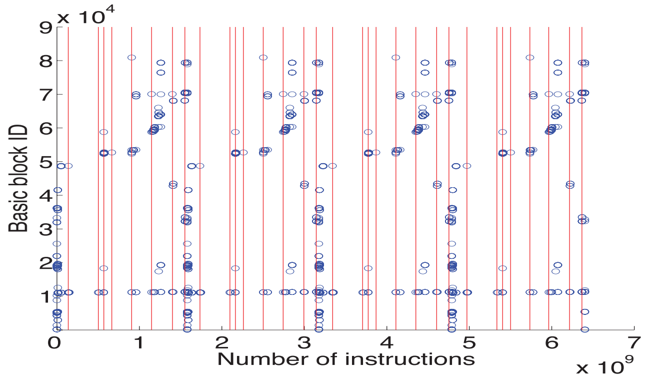


Figure 5. *GCC* inner-phase candidates with inner-phase boundaries.

tical lines indicate outermost and inner phase boundaries, 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. A request consists of data and requested operations. The interface can be viewed as a mini-language. It can be as simple as a stream of bytes and a small number of command-line arguments, as, for example, in a file compression program. It can also be as complicated as a full-fledged programming language, as for example, in a Java interpreter or a simulator used for computer design.

To produce a sequence of repeating requests, we can often just repeat a request if the service is stateless—that is, the processing of a request does not change the internals of the server program. File compression, for example, is uniformly applied to every input file; the compression applied to later files is unaffected by the earlier ones. 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.

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 repeated functions that have 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 3.2.

2.3 Selecting phase markers

Active profiling finds phase markers in three steps. The first step searches for regularity in the basic-block trace induced by a regular input. The second and third steps use real inputs to check for consistency and to identify outmost and inner phases.

Using a binary instrumentation tool, we modify the application to generate a dynamic trace of basic blocks. Given a regular input containing f requests, the trace should contain f nearly identical subsequences. The phase markers must be executed f times each, with even intervening spaces.

We first purify the block trace by selecting basic blocks that are executed 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. We therefore measure the mean and standard deviation of distance between occurrences, and discard blocks whose values are outliers (see 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 regular input may contain a single branch statement. Code to parse a branch may thus occur once per request with this input, but not with other inputs. In step two we check whether a block occurs consistently in other inputs. We use a real input containing g (non-identical) requests. We measure the execution frequency of the candidate blocks and keep 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 increase confidence. We pick the last block in the remaining block sequence as the outmost phase marker.

The rest blocks, always occurring exactly once per outermost phase, may actually mark interesting points *within* an outermost phase. Compilation, for example, typically proceeds through parsing and semantic analysis, data flow analysis, register allocation, and instruction scheduling. We call these *inner phases*. Each is likely to begin with one of the identified blocks.

In step three we select inner phases of a non-trivial length and pick one block for each phase boundary. Figure 5 shows a trace of *GCC* on regular input. Each circle on the graph represents an instance of a candidate inner phase marker. The x-axis represents logical time (number of memory accesses); the y-axis shows the identifier (serial number) of the executed block. We calculate the logical time between every two consecutive circles: the horizontal gaps in Figure 5. From these we select the gaps whose width is more than 3 standard deviations larger than the mean. We then designate the basic block that precedes each such gap to be an inner-phase boundary marker.

The phases of a utility program may also nest. For example, the body of a function in the input to a compiler may contain nested statements at multiple levels. This nesting may give rise to deeply nested phases, which our framework can be extended to identify, using a sequence of identical sub-structures in the input. In the case of the compiler, we can construct a function with a sequence of identical loop statements, and then mark the portions of each inner phase (compilation stage) devoted to individual loops, using the same process that we used to identify outermost phases in the original step of the analysis.

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 of phase marker selection and procedures for recurring-distance filtering.

3. Evaluation

We test six programs, shown in Table 3, from the SPEC95 and SPEC2K 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 two 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.

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 34945 identical subexpressions. 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 [32] to instrument programs for the phase analysis on a decade-old Digital Alpha machine, but measure program behavior on a modern IBM POWER4 machine through its hardware performance monitoring facilities. POWER4 machines have a set of hardware counters, which are automatically read every 10ms. Not all hardware events can be measured simultaneously. We collect cache miss rates and IPCs (in a single run) at the boundaries of program phases and, within phases, at 10ms intervals.

The phase detection technique finds phases for all 6 benchmarks. *GCC* is the most complex program and shows the most interesting behavior. *Perl* has more than one type of phase. We describe these in the next two subsections, and the remaining programs in the third subsection.

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

Table 1. Benchmarks

3.1 Behavior variation in GCC

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 8 inner phase markers. Though the analysis tool never considers the source, we can, out of curiosity, map the automatically inserted markers back to the source code, where we discover that the 8 markers separate different compilation stages.

The first marker is at the end of function "loop_optimize", which performs loop optimization on the current function. The second marker is in the middle point of function "rest_of_compilation", where the second pass of common sub-expression elimination completes. The third and fourth markers are both in function "life_analysis", which determines the set of live registers at the start of each basic block and propagates the life information inside the basic block. The two markers are separated by an analysis pass, which examines each basic block, deletes dead stores, generates

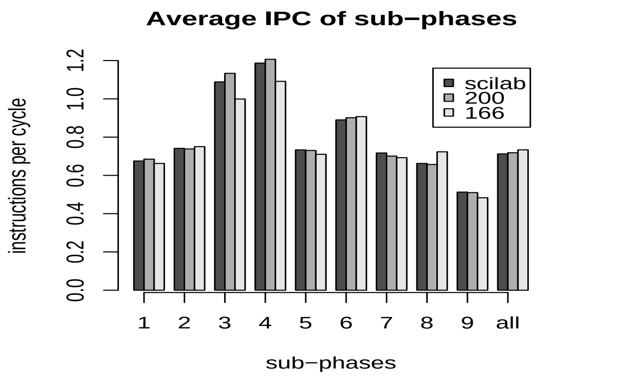


Figure 7. Average outer and inner phase IPC is consistent across three inputs

auto-increment addressing, and records the frequency at which a register is defined, used, and redefined. The fifth marker is in function "schedule_insns", which schedules instructions block by block. The sixth marker is at the end of function "global_alloc", which allocates pseudo-registers. The seventh marker is in the same function as the fifth marker, "schedule_insns". However, the two 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 last marker is in the middle of function "dbr_schedule", which places instructions into delay slots. These automatically detected markers separate the compilation into 9 major stages. 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.

GCC behavior varies with its input. Regularity emerges, however, when we compare the average IPC across the three inputs. As shown by Figure 7, the IPC of 9 sub-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. Note that this is the average for all phase instances. We will show the distribution later in Figure 8.

Temporal patterns also become visible when we cut the execution into phases. Figure 6(a) shows the same curve as Figure 1(b) (in the introduction section) with markings for outermost (solid) and inner (broken) phases. Both outermost and inner phases show similar signal curves across phase instances. The IPC curves of *GCC* on other inputs have a related shape, shown in Figure 6(b)–(d). This shows that *GCC* displays a recurring execution pattern—the same complex compilation stages are performed on each function in each input file. The outermost phase and inner phase markers accurately capture the variation and repetition of program behavior, even when the shape of the curve is not exactly identical from function to function or from input to input. Note that while we have used IPC to illustrate behavior repetition, the phase marking itself is performed off-line and requires no on-line instrumentation.

Next we examine the relation between the length of the phase instances and the IPC. Figure 8 shows the histogram of the length (in a logorithmic scale) and the histogram of the IPC (in 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 9 shows the phase behavior in a lisp interpreter, *Li*, and an English parser, *Parser*. Unlike *GCC*, the two programs do not have clear sub-phases with a different IPC. The 271 phase instances of *Li* have highly varied length and IPC, but the 43 instances of *Parser* has mostly the same length and the same IPC. Finally, Figure 10 show the IPC curves of *Compress* and *Vortex*, showing two sub-phases in the former and 13 sub-phases in the latter.

We note that such analysis would not be possible without knowing the phase markers. Without them we cannot tell for example that a period of 6 million instructions is repeating the same behavior cycle as a period of 10 billion instructions. If we just measure the average behavior of the execution, we would effectively measure only the behavior of the few largest processing steps and not the behavior variation of all steps. The same limitation holds for sampling-based techniques, if they do not know the phase markers.

3.2 Perl

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.

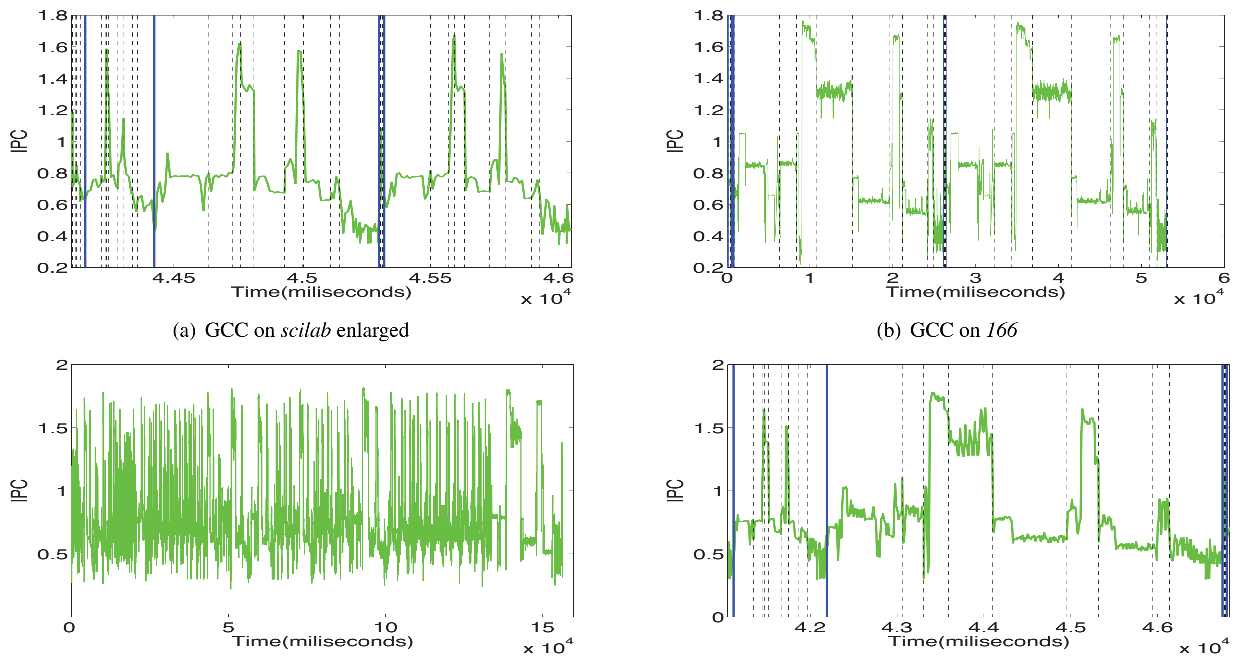
As an example, consider the *Perl* interpreter. The installed version in our system directory has 27 thousand basic blocks and has been stripped of all debugging information. Perl interprets one program at a time, so it does not have outermost phases as other programs do. In hopes of exploring how the 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 2.3) 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.

Such phase markers can be used to analyze how well the current implementation handles procedure calls and other features that a user is interested in. It would enable subsequent analysis for measuring the average time and memory overhead, monitoring for resource consumption, and detecting patterns and anomalies.

3.3 Comparison with procedure and interval phase analysis

In this section, we compare the ability of different analysis techniques—active profiling, procedure analysis, and interval analysis—to identify phases with similar behavior. In Section 4 we will consider how to use these phases to optimize memory management. Different metrics—and thus different analysis techniques—may be appropriate for other forms of optimization (e.g., fine-grain tuning of dynamically configurable processors). It should be noted that 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.

Program phase analysis takes a loop, subroutine, or other code structures as a phase [3, 16, 20, 23, 25, 26]. For this experiment, we mainly consider procedure phases and follow the scheme given by Huang et al., who picked subroutines by two thresholds, θ_{weight}



(a) GCC on *scilab* enlarged

(b) GCC on *166*

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

(d) GCC on *200* enlarged

Figure 6. The IPC curves of different *GCC* runs with phase markers

and θ_{grain} [20]. Assume the execution length is T. Their scheme picks a subroutine p if the cumulative time spent in p (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 [16]. 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]. In this experiment, we use the fixed thresholds from Huang et al.

The extension by Lau et al. may reduce the behavior variation seen by in our experiments. In fact, the phase markers of Lau et al. include some of the markers found by active profiling. We are looking into a more direct comparison, which is tricky. Their method identifies finer phases than active profiling does. Its parameters can be adjusted to find phases of different granularities, but it is unclear what parameter values would produce what number of phases and how the number depends on the training input.

Active profiling allows the user to target the analysis in ways that may not be possible with some general parameters. In our test set, the number of outermost phase instances ranges from 3 (corresponding to input queries) in *Vortex* to 850 (corresponding to input sentences) in *Parser*. It seems unlikely that a single set of thresholds could uncover phases with such wide variation in number. Without a priori knowledge about the number of phase instances, how would one set θ_{grain}? Finally, active profiling can identify phases in memory reference behavior that have no obvious pattern. Such phases can be valuable for memory management, as we show in Section 4.

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, 13, 15, 31]. 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. 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. Space limitations do not permit us to consider all possible prediction and clustering methods. We calculate the upper bound of all possible methods using this interval length by applying optimal partitioning (approximated by k means in practice) on the intervals of an execution. We further assume perfect prediction at run-time—we assume knowledge of the number of clusters, the behavior, and the cluster membership of each interval before execution.

Though phases are not in general expected to have uniform internal behavior, different instances of the same phase should have similar *average* behavior. In our experiments we consider cache hit rate and IPC as measures of behavior. Quantitatively, we compute the *coefficient of variation (CoV)* among phase instances. Statistically CoV is the measure of 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. The results from our hardware counters are not accurate for execution lengths shorter than 10ms, so we excluded phase instances whose lengths are shorter than 10ms.

Figure 11(a) shows the CoVs of cache hit rates. Each program is shown 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 four bars in each group show

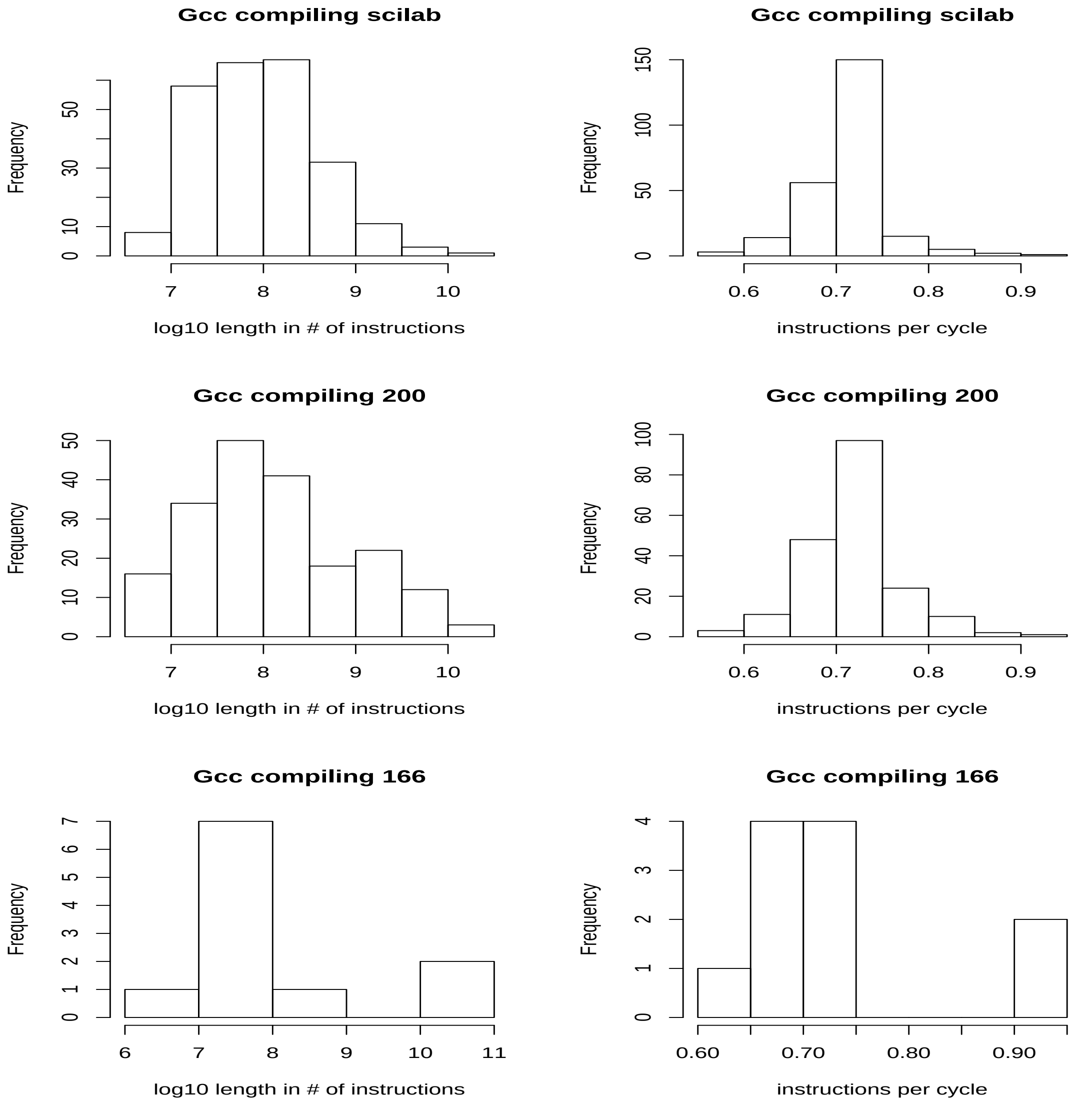


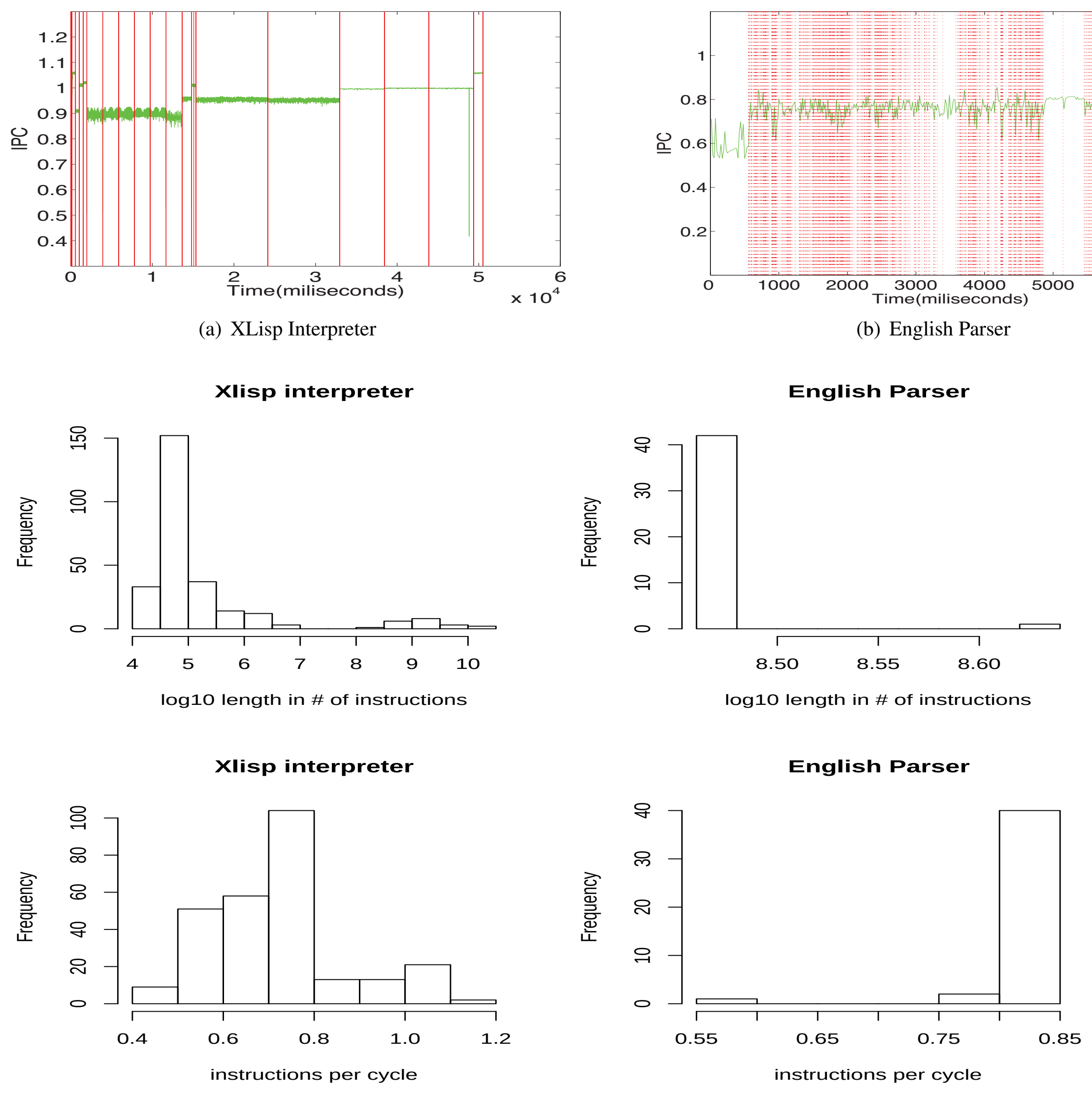
Figure 8. The number of instructions and the average IPC for the phase instances occurred in the three inputs, which have 246, 211, and 11 phase instances respectively. The numbers have similar ranges, and the distributions, although not identical, have their peaks in the same range.

the CoVs of behavior phases, procedure phases, intervals with no clustering (all intervals belong to the same group), and intervals with k-means clustering (the best possible prediction given the number of clusters).

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.

GCC has 9 behavior sub-phases, with a CoV between 0.13% and 12% (average 4.5%). The CoV for procedure phases ranges from 1.2% to 32% with an average of 4%. When cutting the execution into the same number of fixed length intervals as the number of inner phase instances, the CoV is 16%. When the intervals are clustered into 9 groups, the CoV ranges from 1% to 22% with an average of 2.7%. The average CoV for procedure phases and interval phases is lower than that of the behavior phases. However, the procedure phases do not cover the entire execution, and the interval results assume perfect clustering and prediction. In addition, the behavior phase that has the highest consistency (0.13% CoV) is the 4th subphase, which represents 8% of the program execution. The boundaries of this sub-phase are not procedure boundaries.

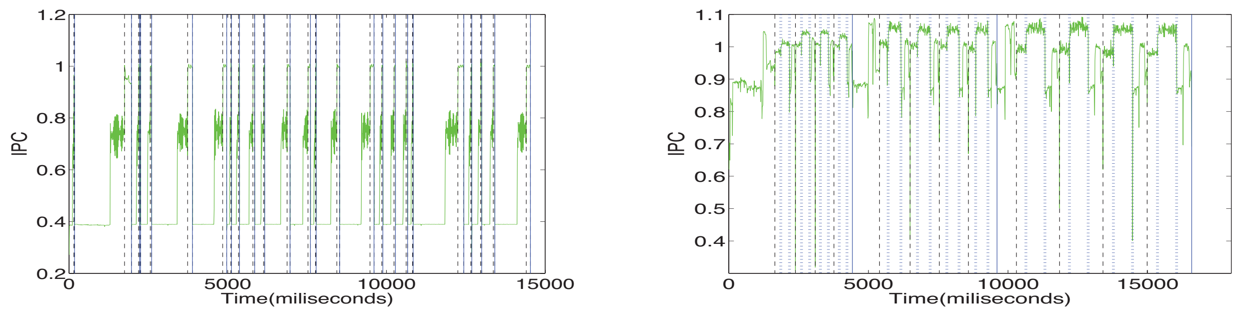
Compress has two sub-phases. The cache hit rate is always 88% for instances of the first sub-phase and 90% for those of the second sub-phase, despite the fact that the instances have different lengths, as shown in Figure 10(e). The relative length ratio is constant. In each outermost phase, the first sub-phase takes 88% of the time and the second takes the remaining 12%. The CoVs of the two sub-phases are 0.15% and 0.21%, barely visible in Figure 11 (a). When divided into two clusters, the smallest and average CoV from interval phases is 0.7% and 0.9%. This program shows the value of variable-length phases: even the ideal clustering of fixed length intervals cannot be as accurate.



(a) XLisp Interpreter

(b) English Parser

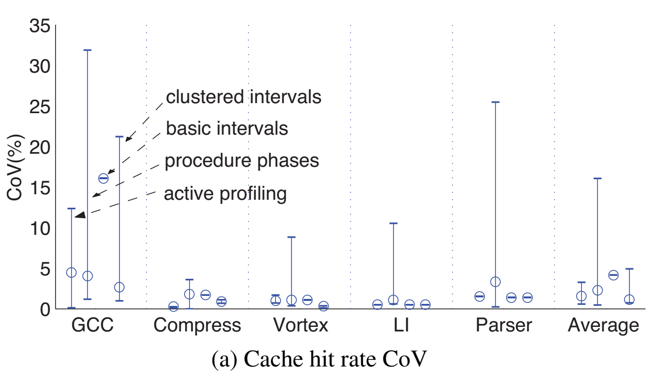
Figure 9. The IPC curves of the XLisp interpreter (*Li*) and the English parser (*Parser*) and the distribution of the instruction count and average IPC of 271 phase instances in *Li* and 43 phase instances in *Parser*.



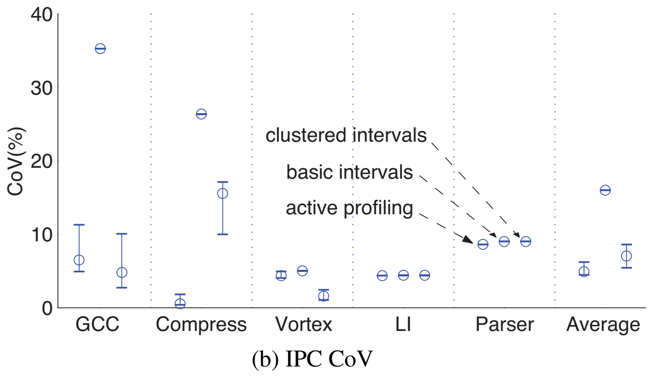
(a) Compress

(b) Vortex

Figure 10. IPC curves of *Compress* and *Vortex* with phase markers



(a) Cache hit rate CoV



(b) IPC CoV

Figure 11. Behavior consistency of four 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.

Vortex has less behavior variation than the previous two programs. The best case procedure and interval phase results are 0.3% CoV, better than the 0.7% minimum CoV of behavior phases. The highest CoV, 8.9%, occurs in a procedure phase. For predicting the cache hit rate, the behavior phase information is not very critical. A carefully picked interval length may capture a similar stable behavior. However, behavior phases still have the advantage of not needing to pick an interval length.

LI shows very few performance changes, as seen in Figure 10(g). Except for procedure phases, all methods have a CoV of less than 1%. The worst procedure, however, shows an 11% CoV. *Parser* is similar. The CoV is below 2% except for procedure phases, which have a CoV of 3% on average and 26% in the worst case. The two programs show that the behavior variation for a procedure can be large even for a program with relatively constant overall behavior.

The results also show the difficulty of setting thresholds in procedure and interval phase analysis. A CoV of 1% may be too large for *LI* but too small for programs such as *GCC*. The techniques of Lau et al. [24] and Georges et al. [16] are adaptive based on the average CoV of all candidate phases in a program. Their best possible result is shown by the lower bound CoV in Figure 11. However, these techniques may still run into a problem if the program has two types of phases that have very different CoVs.

The CoVs of the programs' IPC are shown in Figure 11(b). 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. Between the behavior and interval phases, the qualitative results are the same for IPC as for the cache hit rate. On average across all programs, the CoV is 4.9% for behavior phases and 7.1% for intervals with optimal clustering and prediction.

The five programs show a range of behavior. *Compress* is at one extreme, with behavior that is highly varied but consistent within sub-phases. *LI* is at the other extreme, with behavior that is mostly constant and that does not change between phases.

4. 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, memory usage trend analysis, and memory leak detection.

4.1 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 new "preventive" scheme differs from typical reactive schemes, which invoke garbage collection (GC) when the heap size reaches a soft upperbound. By using 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 and 1GB memory). 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 results in this and the following two Subsections were presented at a workshop in mid 2005 [14]. Later that year Buytaert et al. published a systematic scheme they called garbage collection hints [8]. By profiling SPECjvm98 programs, they inserted GC hints at the Java methods whose invocation corresponds to the recurring local minima in the size of live data, measured using the Merlin trace analyzer [18]. They used a cost model to decide when to invoke the garbage collection and reduced the garbage collector time by up to 30X, resulting in overall execution time improvement of more than 10%. The study shows the need for more sophisticated methods than our preventive GC but at the same time it demonstrates that behavior phases are valuable for memory management.

Buytaert et al. used procedure based phase analysis. The live-data curves of SPECjvm98 showed a regular recurring pattern. This

program inputs	GC methods	exe. time (sec) Pentium 4	per-phase heap size (1K) max	per-phase heap size (1K) avg	total GC calls	total mem. (1K) visited by GC
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

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

is not the case for our programs under default inputs. Indeed, as shown in Section 3, procedure phases do not capture the recurring cache and IPC behavior as well as active profiling does. We believe that active profiling can be used to augment their system and to improve garbage collection for a broader class of applications. It helps to bridge the gap between high-level program information and dynamic memory management. In the late '80s, Wilson et. al. [33] observed that at compute-bound phase boundaries, the amount of live data is likely to be relatively small, and proposed scavenge scheduling, but without a quantitative evaluation.

4.2 Memory usage trend monitoring

For a long running program, it is important to control the cumulative data size, so the memory usage either does not increase linearly with execution time or increases at a manageable rate. Phase analysis can be used to estimate the long-term memory usage trend. Since behavior phases represent a memory usage cycle, one can measure the heap size at the end of each phase instance to estimate the amount of permanent heap data. In addition, a user may test the memory usage of specific behavior, for example, the loading of files in an editor. The user can use active profiling to target the analysis to such actions. We illustrate memory usage trend monitoring using *GCC*.

The memory usage of the test input of *GCC* is shown in Figure 12(a). Using active profiling, we find the compilation of each function as a phase instance. By measuring the heap size at the end of each phase instance, we observe that on average the heap size grows by 58KB for each compiled function. For machines with limited physical memory we can use this knowledge to estimate the memory needed to compile a particular program and warn a user about potential overflow. Alternatively, we may use active profiling to mark the compilation of loops or data structures and to estimate the memory usage trend as a function of smaller compilation units.

We also examine whether the long-term memory usage of GCC might be reduced by eliminating memory leaks. We discuss the preliminary results of memory leak detection in the next section. Here we measure the portion of permanent heap data that are used in later execution. It gives a lower bound on the size of live data. We note that the lower bound is conservative. A heap object is live if it may be used by a later request. Our measurement classifies it as not live if the training input happens not to use the object. The lower curve of Figure 12(a) shows the size of live data across phase instances. It shows an increasing trend until a drop at the end. The trend suggests that the program requires a slowly increasing amount of data. However, the two curves also show that the permanent heap data increases at a faster rate. The large gap may be due to persistent memory leaks.

In comparison, the memory usage trend is perturbed if we measure by logical time (in particular the number of memory references in the execution). The first curve in Figure 12(b) shows the size of the heap measured at every 30000 memory accesses. The steady growth of heap data is obscured by a series of spikes that are caused by the allocation and freeing of temporary objects in a handful of large phase instances. They dominate the time curve because most of the execution time is spent in these few phases. However, most of the 274 phase instances contribute to the increase in memory use. The lower curve shows the size of the live data over time. The trend is similarly obscured by the time-based measurement.

4.3 Memory leak detection

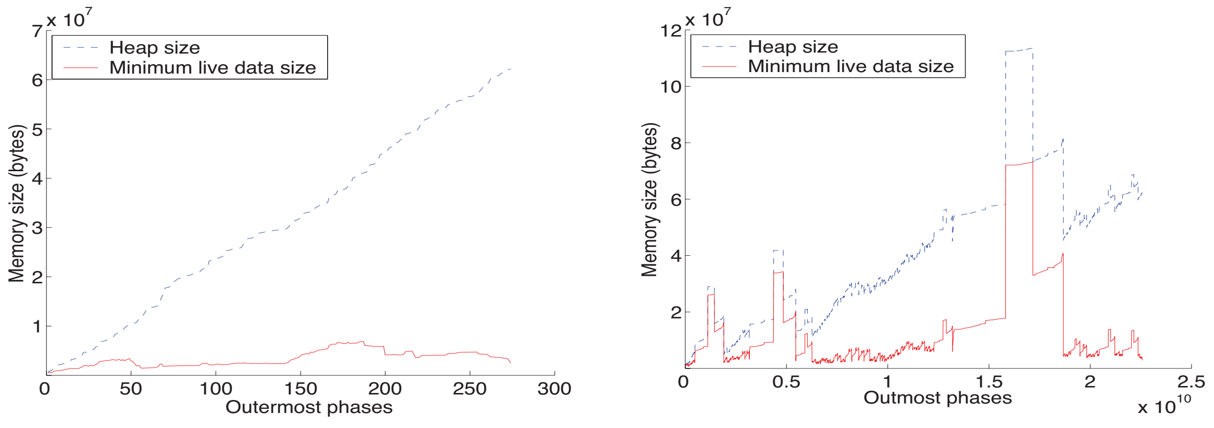
Based on the results of memory usage monitoring, we experiment with a scheme for memory leak detection. We classify dynamic objects as either *phase local*, if their first and last accesses are within a (outermost) phase instance, *hibernating*, if they are used in a phase instance and then have no use until the last phase instance of the execution, or *global* if they are used by multiple phase instances.

Through profiling, our analysis identifies all allocation sites that contribute to the long-term memory increase. We rank them by the rate of their contribution to the memory increase, so that a programmer can fix the most pressing problems and leave the mild ones to the run-time monitoring system.

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. In addition, 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 when their memory pages are 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
```



(a) Heap and live data size after every phase instance

(b) Heap and live data size after every 30000 references

Figure 12. The memory usage of GCC on input *scilab*. The phase-based measurement (on the left) shows the steady growth of the long-live heap data, while the trend is perturbed by temporary heap data in the time-based measurement.

Monitoring techniques have been used to detect memory leaks by research and commercial systems. 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 [9]. 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 the resource and energy demand.

5. Other Related Work

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 hierarchial phases [27]. Using the PSIMUL tool at IBM, Darema et al. showed recurring memory-access patterns in parallel scientific programs [12]. These studies did not predict locality phases. 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 [30]. 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, 25], and Magklis et al. [26] 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 [16]. 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 the 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 [24]. The technique can capture regular repetitions more efficiently than the wavelet analysis [30]. The use of 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 reuses in [30] and user specified behavior in active profiling. Active profiling relies on user input but also permits a user to target specific behavior, for example, finding the 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, 13, 15, 31]). 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 [15]. Nagpurkar et al. proposed a framework for online phase detection and explored the parameter space [28]. 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 matrics were also correlated with linear models (with non-linear components) by Rodriguez et al. [29], queuing models by Jacquet et al. [22], and (logical) performance predicates by Crovella and LeBlanc [10, 11]. Recently, Ipek et al. combined observations automatically into predictive models by applying the general multilayer neural networks on tens to thousands of training results [21]. The 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.

6. 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. 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), memory-usage monitoring, and memory leak detection.

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 structure, and no user intervention beyond the selection of inputs. The entire process is fully automated, from the scripting of profiling runs, through the collection and analysis of the resulting statistics, to the instrumentation of the program binary to mark application phases and perform the garbage collection or memory monitoring.

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.

References

[1] Dong H. Ahn and Jeffrey 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 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, April 1991.

[6] 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.

[7] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *Proceedings of ASPLOS*, 2006.

[8] Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Garbage collection hints. In *Proceedings of The HiPEAC International Conference on High Performance Embedded Architectures and Compilation*, November 2005.

[9] 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.

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

[11] Mark E. Crovella and Thomas 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] Matthew Hertz, Steve M. Blackburn, K. S. McKinley, J. Eliot B. Moss, and Darko Stefanovic. Generating object lifetime traces with merlin. *Transactions on Programming Language And Systems (TOPLAS) (to appear)*.

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

[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] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In *Proceedings of Euro-Par*, pages 196–205, 2005.

[22] Adeline Jacquet, Vincent Janot, Clement Leung, Guang R. Gao, Ramaswamy Govindarajan, and Thomas L. Sterling. An executable analytical performance evaluation approach for early performance prediction. In *Proceedings of IPDPS*, 2003.

[23] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. Technical Report CS2004-0804, UCSD, November 2004. *conference version to appear in CGO'06*.

[24] 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.

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

[26] 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.

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

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

[29] Germán Rodríguez, Rosa M. Badia, and Jesús Labarta. Generation of simple analytical models for message passing applications. In *Euro-Par*, pages 183–188, 2004.

[30] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architect ural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.

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

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

[33] P.R. Wilson and T.G. Moher. Design of the opportunistic garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 29–35, 1989.