

Characterizing Phases in Service-Oriented Applications

Xipeng Shen Chen Ding Sandhya Dwarkadas Michael L. Scott

Computer Science Department
University of Rochester

{xshen, cding, sandhya, scott}@cs.rochester.edu

ABSTRACT

The behavior of *service-oriented* programs depends strongly on the input. A compiler, for example, behaves differently when compiling different functions. Similar input dependences can be seen in interpreters, compression and encoding utilities, databases, and dynamic content servers. Because their behavior is hard to predict, these programs pose a special challenge for dynamic adaptation mechanisms, which attempt to enhance performance by modifying hardware or software to fit application needs.

We present a new technique to detect phases—periods of distinctive behavior—in service-oriented programs. We begin by using special inputs to induce a repeating pattern of behavior. We then employ frequency-based filtering on basic block traces to detect both top-level and second-level repetitions, which we mark via binary rewriting. When the instrumented program runs, on arbitrary input, the inserted markers divide execution into phases of varying length. Experiments with service-oriented programs from the Spec95 and Spec2K benchmark suites indicate that program behavior within phases is surprisingly predictable in many (though not all) cases. This in turn suggests that dynamic adaptation, either in hardware or in software, may be applicable to a wider class of programs than previously believed.

1. INTRODUCTION

Program and hardware adaptation to take advantage of dynamic changes in behavior is becoming increasingly important as a result of technological trends, including increases in clock frequency, pipeline complexity, the processor–memory speed gap, the depth of the memory hierarchy, and processor and system power density. A number of recent techniques divide an execution into phases and adapt hardware and/or the program on a per-phase basis. The benefit of the adaptation depends on the accuracy of phase partitioning and behavior prediction.

In this paper, we consider a class of dynamic programs whose behavior depends strongly on their input. An example is a compiler, in which behavior varies not only from one input to another but also across sections of the same input. Other examples include interpreters, compression and transcoding utilities, databases, and web servers. These applications share the common feature that they provide some sort of *service*: they accept, or can be configured to accept, a sequence of requests, each of which is processed more-or-less independently of the others. Because they are so heavily dependent on their input, service-oriented applications display much less regular behavior than does the typical scientific program. The question naturally arises: do they have *any* behavior sufficiently regular to make run-time adaptation worthwhile?

To address this question we introduce a new technique called *active profiling*, which exploits the following observation: if we

provide a service-oriented application with an artificially regular input, then its behavior is likely to be regular as well. If this regularity allows us to identify behavior phases and mark them in the program executable, then we may be able to delimit phases during normal operation as well, when irregular inputs make frequency-based phase detection infeasible. More specifically, 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. It then uses different real requests to capture common sub-phases and to verify the representativeness of the constructed input. In programs with a deep phase hierarchy, the analysis can be repeated to find sub-sub-phases, etc. Active profiling differs from traditional profiling in that its input is specially designed to expose desired behavior in the program. It requires access to the program binary only; no knowledge of loop or function structure is required. All basic blocks are considered as possible phase change points.

In a recent study, Shen et al. [15] used wavelet analysis on the program’s data reuse distance to identify and mark phases in regular applications. Because instances of a given phase within a single execution were nearly identical in length and behavior, early instances could be used to predict the length and behavior of later instances with high accuracy. In comparison, when execution was divided into intervals of uniform length, without regard to phase structure, behavior was much less predictable, because a given interval might contain pieces of more than one phase. Using knowledge of phases, Shen et al. were able to apply dynamic adaptation of cache size and memory layout to obtain significant improvements in application performance.

Unfortunately, the methodology of Shen et al. does not work for service-oriented programs, for two reasons. First, wavelet analysis depends on temporal regularity, which service-oriented programs do not display, due to irregular inputs. Second, Shen et al. predict the same behavior for all instances of a given program phase. This assumption does not hold in service-oriented programs. In a compiler, for example, compilation of a top-level function may constitute an outermost phase, but the length and behavior of the phase may depend on the length and semantic complexity of the function being compiled.

In this paper, we use active profiling and a simpler phase analysis technique to mark phase transitions in service-oriented programs. We then measure the extent to which instances of a given phase display similar behavior. We test our techniques on service-oriented applications from the Spec95 and Spec2K benchmark suites [1]. We measure the behavior similarity of instances of the same program phase when running on irregular production inputs, and compare this similarity to that of temporal intervals chosen without regard to phase boundaries, both in terms of phase length and as in

terms of phase behavior. Though less regular than scientific programs, we find that service-oriented programs often display a surprising degree of predictability, suggesting that dynamic adaptation techniques may be applicable to a significantly broader class of applications than has previously been believed.

2. ACTIVE PROFILING AND PHASE DETECTION

This section describes how we construct regular inputs and detect phases through profiling.

2.1 Constructing regular inputs

The behavior of service-oriented programs is highly input dependent due to the varying nature of the input requests, resulting in program phases of variable length and behavior. Input dependence is a challenge because we cannot pre-determine program behavior without knowledge of the program input. The key idea behind the detection of variable-length phases is to force repeatable behavior in the program by feeding it regular inputs that contain repeating requests. An example is *GCC* from the SPEC95/2K benchmark suite. The speed, measured in instructions per cycle (IPC), varies with the program being compiled. Figure 1 (a) shows the IPC curve for a reference input. However, we can induce regular behavior in *GCC* by letting it compile a sequence of identical functions. Figure 1 (b) shows the IPC curve on such a regular input. Solid and broken vertical lines indicate outer and inner phase boundaries. The fact that the program behavior repeats for a known number of times is critical to the phase detection described in the next section. The rest of this section discusses ways to construct regular inputs for service-oriented programs.

A service-oriented program provides an interface for specifying incoming 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 tuple. For example, a file compression program accepts a sequence of input files, together with such arguments as the level of compression and the name of the output file. The interface language can also be a full-fledged programming language. Examples are interpreters for Java or program simulators such as SimpleScalar [6].

Each request to a service-oriented program can be thought of as a mini-program written in the language specified by the service interface. To produce a sequence of repeating requests, we can often just repeat a request. This repetition is easy when 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 generally unaffected by the earlier ones. Repetition is more difficult when the service stores information about requests. A compiler generally requires that all input functions in a file have unique names; a database changes state as a result of insertions and deletions. For the stateful case, we need to consider these constraints when generating regular requests. For a compiler, we replicate the same function but give each a different name. For a database, we balance insertions and deletions or use inputs containing only lookups.

We use five benchmarks in our experiments. Two stateless programs are *Compress*, the Unix file compression utility from Spec95, and *Parser*, an English language parser from Spec2K. As distributed by SPEC, *Compress* is programmed to compress and decompress the same data 25 times, which is a regular input. We get an irregular input by applying it to different data of a random length. *Parser* takes a sequence of natural language sentences as its

input and parses them one by one. For regular input we created a file containing multiple copies of a single sentence.

The three other programs store a different degree of information when processing requests. *GCC*, from SPEC2K, is a version of the GNU C Compiler. As noted above, its regular input contains copies of identical functions with different names. The function itself contains multiple copies of a loop to simplify phase discovery. *LI* is a Lisp interpreter from SPEC95. It interprets expression by expression. For its regular input we provide multiple copies of an expression that calculates the sum of many copies of the same simple sub-expression. *Vortex* is an object-oriented database from SPEC2K. Its outer-level loop extends the database, then performs random lookup, deletion, and insertion operations. The input file to *Vortex* contains the initial database and the parameters controlling the number of various operations. To make its behavior regular, we comment out the delete and insertion operation and change the random generator such that the program does the same lookups on the same data every time.

Phase analysis and detection does not require a long execution of the program. We use 4 identical functions for *GCC*, 6 identical expressions for *LI*, 6 identical sentences for *Parser*.

Once a regular input is constructed, we measure such run-time statistics as IPC, and compare them with typical inputs. The goal is to ensure that behavior on the artificial input does not deviate significantly from behavior on real inputs. Our first regular input for *GCC* included repeated functions whose body contained a large number of copies of the statement “*a=0*”. The IPC curve was mostly flat, lacking the variations seen in the IPC curve of typical inputs. We subsequently constructed an input with repeated instances of a function copied from real input. The IPC curve then looked like that obtained from real, irregular inputs. While both inputs have the same number of top-level phases, the phases in the first input are hard to distinguish due to their uniform behavior. The higher variation in behavior within the phases in the second input allows easier determination of significant phases in typical executions.

The appropriate selection of regular inputs is important not only for capturing typical program behavior; it also allows targeted analysis for 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 sub-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.

2.2 Detecting and marking phases via frequency-based filtering

When run with a regular input, a service-oriented program should generate a series of mostly repeating phases. Phase detection examines the frequency of instructions being executed to identify candidates for inserting phase markers. It first determines the outermost phase and then the inner phases. As the marker for a given phase, we choose a basic block that is always executed at the beginning of that phase, and never otherwise.

2.2.1 Outermost phase detection

The instruction trace of an execution is recorded at the granularity of basic blocks. The result is a basic-block trace, where each element is the address of a basic block. Let f be the number of requests in the regular input. Then a block that marks the outermost phase should appear exactly f times in the trace. We therefore filter out all blocks that appear $k \neq f$ times. What remains is a

candidate trace. Many integer programs have a large number of small blocks, so the candidate trace may still contain thousands of different blocks. Although the remaining blocks should exhibit a repeating pattern, variations may occur due to initialization, finalization, and in some programs garbage collection. To separate the phase behavior, we check for the regularity of the appearances.

For each instance b_i of a block b , other than the first, we define the *recur-distance* r_{b_i} of b_i to be the number of dynamic instructions between b_i and the previous appearance of the block, b_{i-1} . We average across all instances of b to obtain the average and the standard deviation, \bar{r}_b and σ_{r_b} . Then we calculate the average across all non-initial instances of *all* blocks to get \bar{r} and σ_r . Finally, we calculate $\sigma_{\bar{r}}$, the standard deviation of \bar{r} across all blocks. Our experiments confirm that the probability is small that an initialization, finalization or garbage collection block will appear exactly as many times as there are requests in the input. Such a block is therefore likely to be a statistical outlier: it will have a very short average recur-distance (all instances occur during initialization, finalization, or a single GC pass), or it will show wide variation in recur-distance (because it is not tied to requests). Put another way, a block b that *does* represent a phase boundary should satisfy $\bar{r}_b \approx \bar{r}$, and $\sigma_{r_b} \approx \sigma_r$. Using three standard deviations as a threshold, we filter out blocks for which $\bar{r} - \bar{r}_b > 3\sigma_r$ or for which $|\sigma_r - \sigma_{r_b}| > 3\sigma_{\bar{r}}$. We search the trace in reverse order and choose the first block satisfying both conditions as the outermost phase end marker.

2.2.2 Inner phase detection

The outermost phase corresponds to the entire processing of a request. Many programs have sub-phases. For example, the compilation of a function may comprise many stages: parsing and semantic analysis, data flow analysis, register allocation, instruction scheduling. In contrast to the previous step, where we were interested in identical behavior, in this step we are interested in the common sub-phases of all behavior. Instead of using a regular input, we use irregular input to find these common sub-phases.

During the outermost phase detection, we obtain a trace in which each candidate block occurs the same number of times as the number of regular requests. After choosing one block as the phase-end marker, and instrumenting it via binary rewriting, we re-run the application on irregular input. During this run we record the blocks that appear exactly once in most instances of the outermost phase (90% in our experiments). These constitute the candidate blocks for inner-phase boundaries.

Figure 2 shows a trace of *GCC*. Each circle on the graph represents an instance of a inner-phase candidate block. The x-axis represents logical time (number of instructions executed); 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 2. From among these gaps we select the outliers, whose width is over $m + 3 * d$, where m is the average gap width and d is the standard deviation. We then define the basic block to the right of each such outlier to be an inner-phase boundary. In Figure 2, the boundaries are shown by vertical broken lines with inner-phase candidates shown as circles. Intuitively, the inner phases are large gaps in the graph.

The interface language for a service-oriented program may have a nested structure. 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 “sub-sub-phases” or program execution, 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 state-

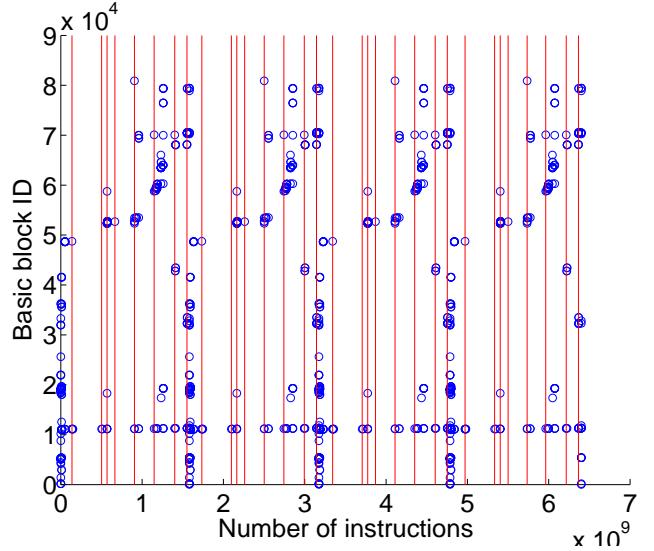


Figure 2: *GCC* inner-phase candidates with inner-phase boundaries

ments, and then mark the portions of each sub-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.

2.2.3 Marking phases for evaluation

Phase marking requires binary instrumentation. At present we rely on ATOM [17], running on Digital Alpha machines. We also collect measurements on a faster machine: a 1.3GHz IBM POWER4 pSeries. For these latter measurements we insert phase markers by hand, with help from debugging tools. Applications are first compiled with the “-g” debug option on the Alpha machines. ATOM is then used to determine the program counter of phase marker basic blocks. A simple manual search through the assembly code gives us the line number of the corresponding source code and the name of the source file, where we insert a phase marker. Sometimes a statement maps to multiple basic blocks and a marking basic block may not be the starting or ending basic block. A statement may call a subroutine, for example, with parameters that are themselves subroutine calls. The marking basic block may be between the two subroutine calls. This requires decomposing the statement into multiple simpler statements. While these issues are resolved by hand in our POWER4 experiments, it should be emphasized that the entire process can easily be automated on a machine with an ATOM-like tool, with no access required to source code, assembly code, or debugger symbol tables.

3. EVALUATION

In this section, we measure phase behavior in five programs, shown in Table 3, from the SPEC95 and SPEC2K benchmark suites: a file compression utility, a compiler, an interpreter, a natural language parser, and an object-oriented database. Three other service oriented programs—two more compression utilities and another interpreter—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.

Expanding on the description in Section 2.1, we construct regu-

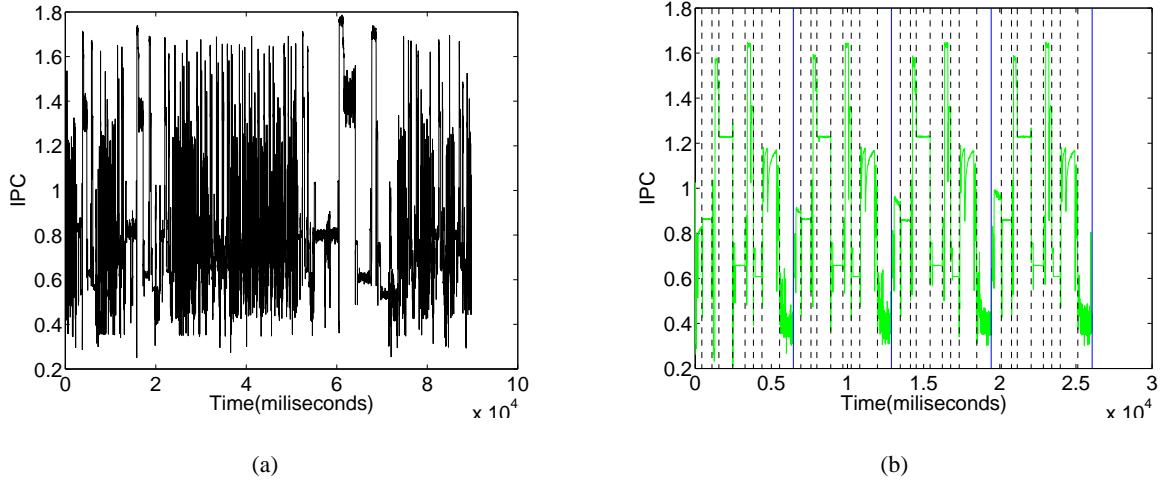


Figure 1: (a) IPC curve of *GCC* on a ref input *scilab*. (b) IPC curve of *GCC* on a regular input with top-level (solid vertical lines) and inner-level (broken vertical lines) phase boundaries.

Table 1: Benchmarks

Benchmark	Description	Source
Compress	common 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

lar inputs as follows. For *GCC* we use a file containing 4 identical functions, each with the same sequence of 119 loops. For *Compress*, which is written to compress and decompresses the same input 25 times, we provide a file that is 1% of the size of the reference input. For *LI* we provide 6 identical expressions, each of which contains 34945 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 three 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 [17] to instrument programs for the phase analysis on a decade old Digital Alpha machine, but measure program behavior on a modern IBM POWER4 machine. Due to the lack of a binary rewriting tool on the IBM machine, we insert phase markers into the Alpha binary, manually identify their location, insert the same markers at the source level, and then compile and run the marked program on the IBM platform.

We use IBM hardware performance monitoring facilities to record program behavior. POWER4 machines have a set of hardware events. The AIX 5.1 operating system provides a programming interface library—pmapi—to access those counters. By instrumenting the program with the library function calls, one can determine the count of the hardware events specified by the user at the instrumentation point. The instrumentation is also set to automatically generate an interrupt every 10ms so that the hardware counters are read at the 10ms granularity. We use a single run to avoid pertur-

bation from OS behavior. Not all hardware events can be measured simultaneously. We collect cache miss rates and IPCs at the boundaries of program phases and, within phases, at 10ms intervals.

The phase detection technique finds phases for all 5 benchmarks. *GCC* is the most complex program and shows the most interesting behavior. We describe it in the next section, and return to the remaining programs in a later section.

3.1 *GCC*

GCC comprises 120 files and 222182 lines of C code. The phase detection technique successfully finds its outermost phase, which begins at the compilation of an input function. Furthermore, the technique finds 8 inner phases inside the outermost phase through analysis at the binary level. We map the automatically inserted markers back to the source code and find 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 one is at a middle point of function “rest_of_compilation”, where the second CSE pass (Common sub-expression elimination) just 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 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 a middle point of function “dbr_schedule”, which places instructions into delay slots. The marker happens after the filling of delay slots. These automatically detected markers separate the compilation into 8 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 may not be easy to identify sub-phases that occupies a significant portion of the execution time.

GCC behavior varies with its input. Figure 1(a) shows the IPC curve of *GCC* on reference input *scilab.i*. Each point on the graph shows the average IPC of a 10ms interval. The points have been connected to form a continuous curve. The presence of a regular pattern is not obvious from visual inspection.

However, regularity emerges when we cut the execution into phase instances. Figure 3(a) shows the same curve marked with the boundaries of the outermost and inner phases, represented by solid and broken vertical lines respectively. For clarity, Figure 3(b) shows an enlarged part of the curve, in which there are three large phase instances and several tiny ones. Although they have a different width and height, they show a similar signal shape—with two high peaks in the middle and a declining tail. A related shape composes the IPC curves of *GCC* on other inputs, shown in Figure 3(c)(d)(e)(f). This shows that *GCC* has a consistent program pattern—the same complex compilation stages are performed on each function in each input file. The phase and sub-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. The phase marking is done off-line and requires no on-line measurement.

Figure 4(a) shows a distribution graph of IPC and cache hit rates for phase instances of *GCC*. Instances of different sub-phases are represented by different symbols. *GCC* has 57 instances of the outermost phase in the reference input. Each instance is divided into 8 inner phases. We have a total of 456 points in Figure 4(a). The 456 points cluster into 5 rough groups. The top group is the cluster of phase 3. It corresponds to the highest peak in the IPC curve, and is separated from the other phase instances. The cluster of phase 4 overlaps with the top of the cluster of phase 6, but separates from its major body. Phase 4 corresponds to the highland in the IPC curve, and phase 5 corresponds to the second highest peak with some low transition parts. The cluster of phase 8 corresponds to a short segment in IPC curve with low IPC and high cache hit rate. It separates from the other clusters well. The remaining large cluster contains the instances of phases 1, 2, 5 and 7. These four phases correspond to the 4 lowest IPC segments. They are close to each other but still separate mostly. Phase 2 has the highest cluster, phase 7 the lowest, with phase 1 in the middle. Phase 5 has the rightmost cluster with highest cache hit rate. Most of the 8 groups are very tight except for the values from phase 2. Even for this group, most points have almost the same IPC, and cache hit rates that vary by less than 0.2.

To quantify consistency, we measure the average value, the standard deviation, and the *coefficient of variance (CV)* of the behavior of phase instances. The CV, calculated as the standard deviation divided by the mean times 100%, shows the tightness of the normalized distribution. Assuming a normal distribution, a standard deviation of d means that 68% values fall in the range $[m-d, m+d]$ and 95% fall in $[m-2d, m+2d]$.

The measures we consider include cache hit rate, IPC, and the relative length of a sub-phase (percentage of its enclosing phase instance). Small standard deviations and co-variances imply a highly clustered distribution and precise phase-based behavior prediction. The first part of Table 2 shows the behavior variation of the 8 sub-phases and their average.

The sub-phases have different behavior. The average hit rate ranges from 82% in the 1st sub-phase to 100% in the 4th sub-phase. The average IPC ranges from 0.64 in the 7th sub-phase to 1.49 in the 3rd sub-phase. The distribution is very tight. The CV is 0 for three sub-phases and 4.4% on average, meaning that 95% of the

values are within 15% of the average. The relative phase length ranges from 4% to 41% of the total length, showing they are significant parts of the phase behavior. The CV is high, showing that the relative length is not as highly predictable as the hit rate or the IPC.

These measures of consistency compare favorably to those obtained without knowledge of phase structure. The last three rows in the *GCC* section of Table 2 measure the behavior distributions of all sub-phase instances (as opposed to instances of the same phase), plus coarse-grain and fine-grain fixed-length temporal intervals. The length of the coarse-grain interval is the total execution length divided by the number of phase instances, so it has the same granularity as the average phase instance. The fine-grain interval is 10ms in length, which is the smallest interval from the hardware counter.

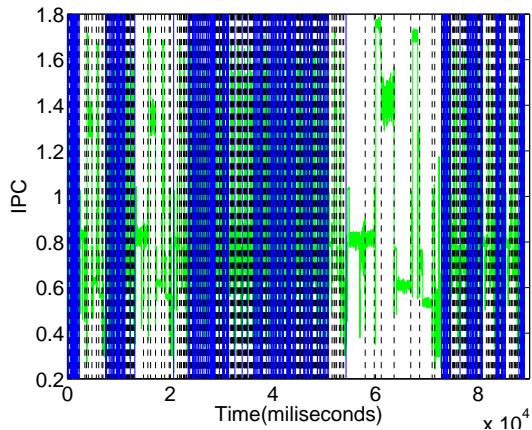
Without information about the sub-phase identity, the average hit rate is 91% with 9.6% CV, IPC 0.93 with 36% CV, and relative length 12% with 102% CV. The distribution is much wider because different behavior from sub-phases is included in one group. The distribution of the interval behavior is worse. The CV of the IPC is 35% for coarse-grain intervals and 44% for fine-grain intervals, showing that 68% of the values may vary by at least one third of the average. The results imply that analysis without knowing the phase boundaries would find significant variation but little consistency in program behavior.

From these results, we conclude that the program has very consistent phase behavior, with significant behavior variation across sub-phases. The high degree of consistency among corresponding sub-phases suggests that the hit rate, IPC, and relative sub-phase length could be predicted fairly accurately by off-line phase profiling and on-line behavior monitoring. It seems reasonable to expect that other behaviors—and in particular those that might be exploited by hardware or software adaptation—will show strong phase consistency as well.

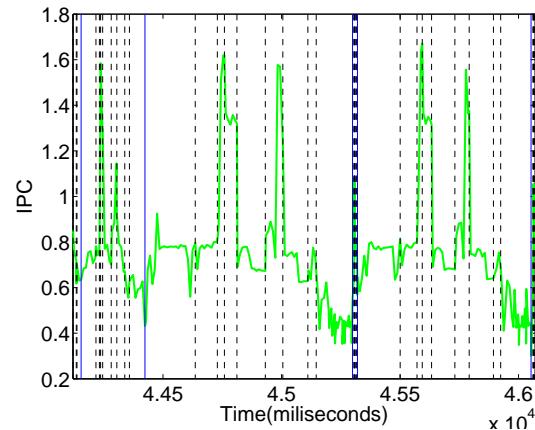
3.2 Compress, Vortex, LI, and Parser

Compress has two inner phases and shows even more regular sub-phase behavior than *GCC* did. Figure 5(a) shows the IPC curve with the phase and sub-phase markers in solid and broken vertical lines. All phase instances have two sub-phases even though different instances have different length, determined by the size of the file being compressed. Figure 4(b) shows that the behavior of the two sub-phases is highly consistent, with almost identical hit rate and IPC. Table 2 gives quantitative results. The same sub-phase always has the same hit rate, 88% and 90% respectively. The first sub-phase always has the same IPC, 0.49. 95% of the instances of the second sub-phase have an IPC between 0.96 and 1.02. The first sub-phase always takes 88% of the length of the outermost phase, the second sub-phase takes the remaining 12%. The behavior of the intervals shows a relatively tight distribution of the hit rate but greater variations in IPC, with the CV ranging from 26% to 41%. The CV for the relative length is 77%. Therefore, the behavior of *Compress* is precisely predictable with the sub-phase information; otherwise, IPC cannot be predicted accurately.

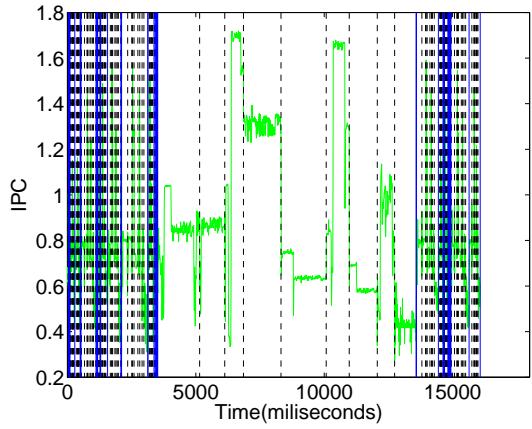
Vortex shows consistent but varying IPC among the instances of the outermost phase, as shown in Figure 5(b). Our detection method also finds two levels of sub-phases, shown by broken and dotted vertical lines. The overall phase behavior is consistent, shown by the tight cluster in Figure 4(c). Quantitatively, the hit rate is 97% on average with no variation, and the IPC is between 0.96 and 1.02 in 95% of the cases. Without phase information, the behavior of coarse-grain intervals is as consistent, although the fine-grain intervals have a much higher variation, as shown in Table 2. In this



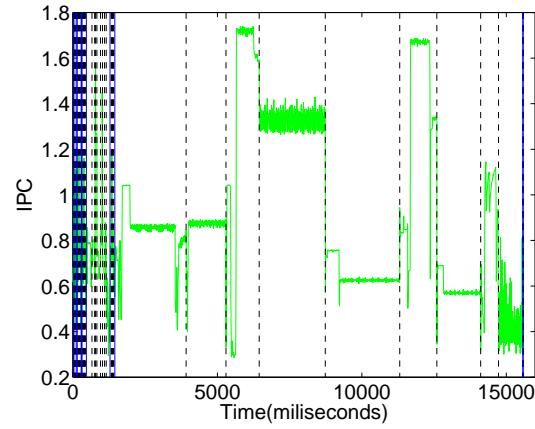
(a) scilab



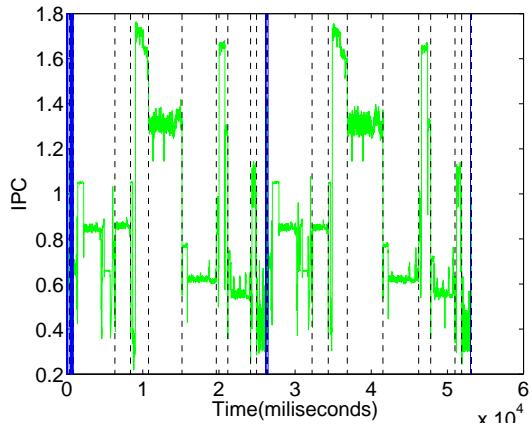
(b) scilab enlarged



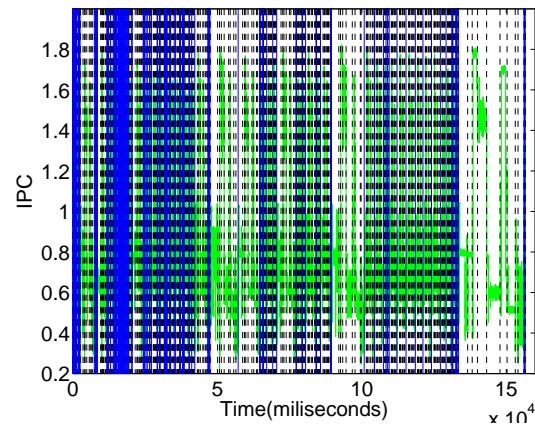
(c) expr



(d) integrate

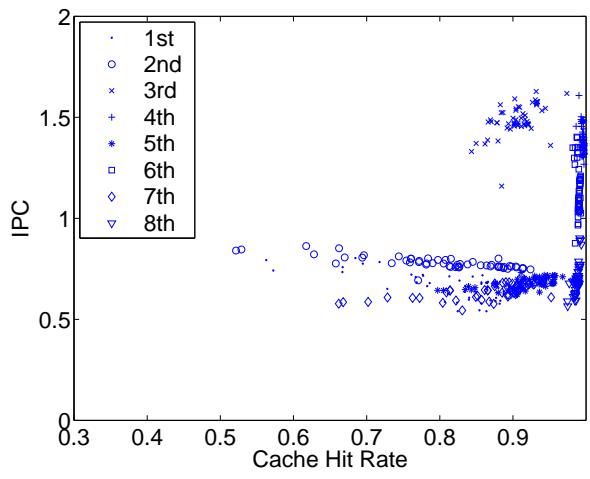


(e) 166

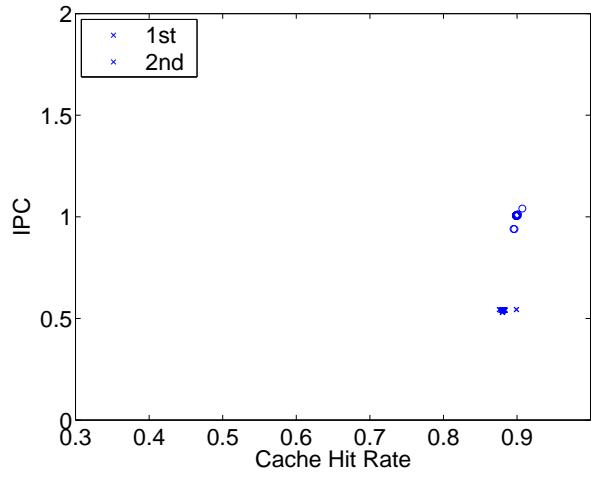


(f) 200

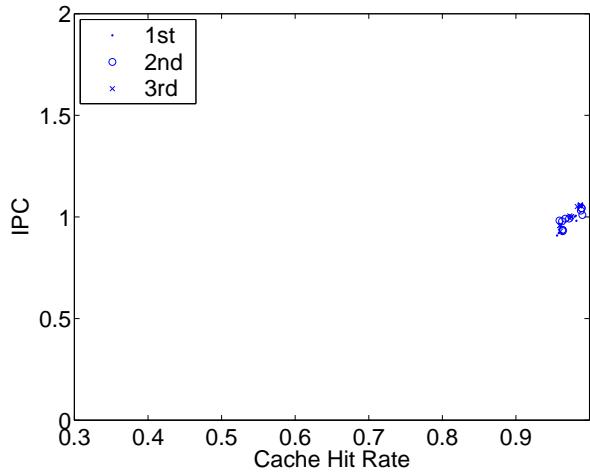
Figure 3: IPC curves of GCC on SPEC2K reference inputs with phase markers



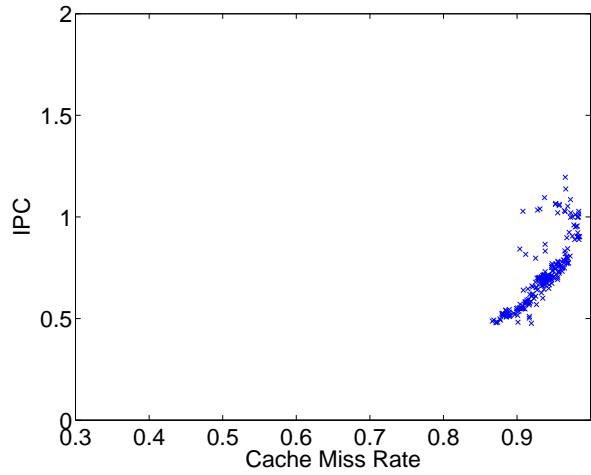
(a) GCC on all ref input: 456 instances of 8 phases



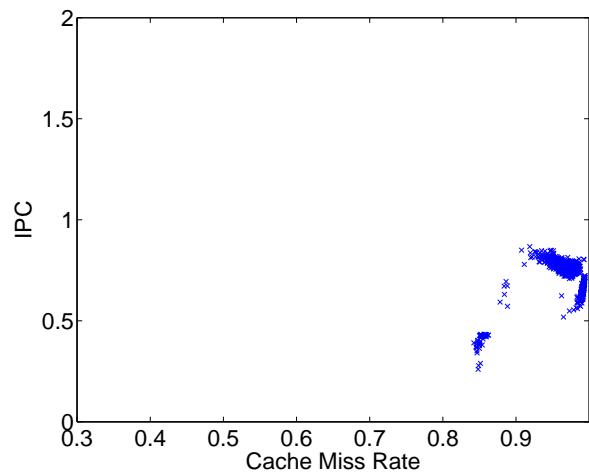
(b) Compress: 50 instances of 2 phases



(c) Vortex: 36 instances of 3 phases



(d) Li: 271 instances of 1 phase

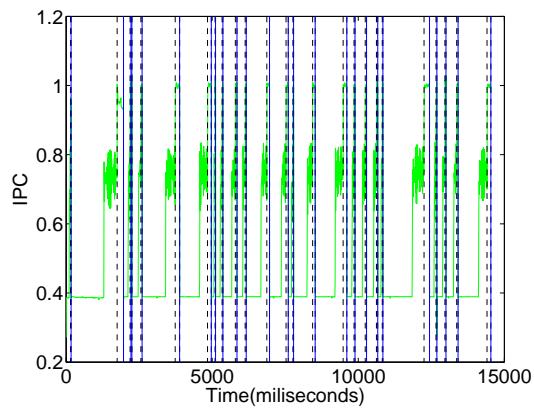


(e) Parser: 703 instances of 1 phase

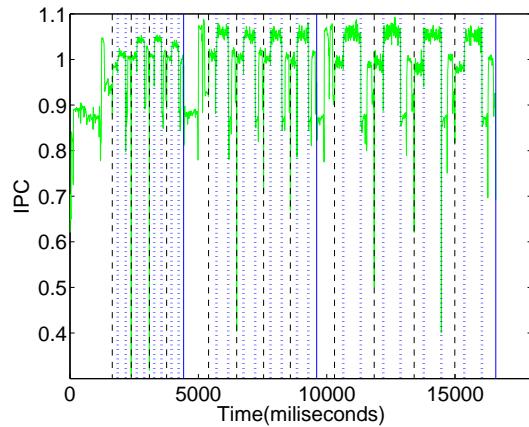
Figure 4: IPC and cache hit rate distribution graphs.

Table 2: The behavior distribution of phases, sub-phases, and intervals

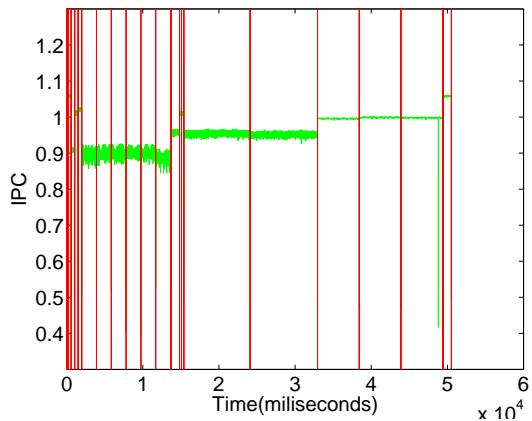
program	phase (sub-phases)	measurement unit	cache hit rate			IPC			length ratio		
			mean	std.	CV	mean	std.	CV	mean	std.	CV
GCC	1st	sub-phases with IDs	0.82	0.08	9.8	0.67	0.06	9.0	0.41	0.11	27.8
	2nd		0.80	0.09	11.2	0.78	0.03	3.8	0.08	0.03	38.6
	3rd		0.91	0.03	3.3	1.49	0.06	4.0	0.04	0.02	50.2
	4th		1.00	0.00	0.0	1.39	0.06	4.3	0.08	0.04	47.6
	5th		0.92	0.04	4.3	0.68	0.02	2.9	0.15	0.03	21.1
	6th		0.99	0.00	0.0	1.14	0.11	9.6	0.07	0.02	22.4
	7th		0.87	0.07	8.0	0.64	0.03	4.7	0.12	0.03	24.3
	8th		0.99	0.00	0.0	0.70	0.06	8.6	0.04	0.02	40.8
	avg.		0.91	0.04	4.4	0.93	0.06	6.5	0.13	0.04	28.5
	avg.		0.91	0.09	9.6	0.93	0.33	36.0	0.12	0.1219	101.6
Compress	1st	sub-phases with IDs	0.88	0.00	0.0	0.49	0.00	0.0	0.88	0.00	0.0
	2nd		0.90	0.00	0.0	0.99	0.01	1.0	0.12	0.00	0.0
	avg.		0.89	0.00	0.0	0.74	0.01	1.4	0.50	0.00	0.4
	avg.	sub-phases without IDs	0.89	0.01	1.2	0.74	0.25	34.4	0.50	0.3866	77.3
	avg.		0.89	0.02	1.7	0.55	0.14	26.3	*	*	*
	avg.		0.89	0.03	3.4	0.56	0.23	40.7	*	*	*
Vortex	1st	sub-phases with IDs	0.98	0.01	1.0	1.00	0.06	6.0	0.34	0.06	17.6
	2nd		0.97	0.01	1.0	0.99	0.04	4.0	0.32	0.08	25.0
	3rd		0.97	0.01	1.0	1.01	0.05	5.0	0.34	0.09	26.5
	avg.		0.97	0.01	1.0	1.00	0.05	5.0	0.33	0.08	23.0
	avg.	sub-phases without IDs	0.97	0.01	1.2	0.99	0.05	5.1	0.33	0.13	39.4
	avg.		0.97	0.01	1.1	0.98	0.05	5.0	*	*	*
	avg.		0.97	0.03	2.9	0.98	0.08	8.5	*	*	*
LI	avg.	phases	0.94	0.03	3.2	0.72	0.14	19.4	*	*	*
	avg.	coarse-grain intervals	0.98	0.01	0.5	0.96	0.04	4.4	*	*	*
	avg.	10ms intervals	0.98	0.01	1.3	0.95	0.08	8.1	*	*	*
Parser	avg.	phases	0.97	0.03	3.1	0.71	0.10	14.1	*	*	*
	avg.	coarse-grain intervals	0.98	0.01	1.4	0.75	0.07	9.0	*	*	*
	avg.	10ms intervals	0.97	0.03	3.2	0.71	0.12	16.8	*	*	*



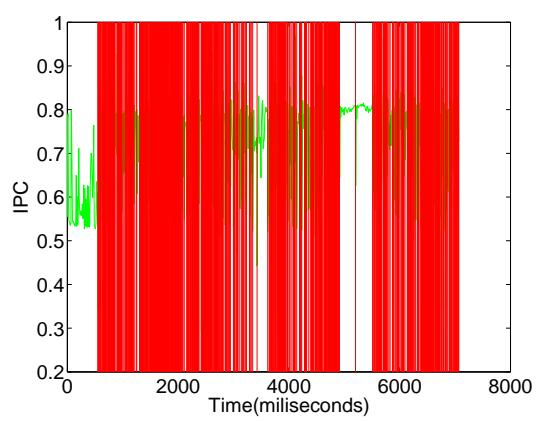
(a) Compress



(b) Vortex



(c) Li



(d) Parser

Figure 5: IPC curves of Compress, Vortex, Li and Parser with phase markers

Table 3: Correlation among metrics

	cache-IPC	cache-length	IPC-length
correlation coef. of mean	37.9	-41.1	-43.0
correlation coef. of CV	14.9	30.4	52.6

case, the phase information is less critical. A carefully picked interval length may capture a similar stable behavior. However, a phase-based method still has the advantage of not needing to pick an interval length.

LI shows three stages in the execution with a relatively constant performance at each stage, shown by Figure 5(c). The distribution of the hit rate and IPC shows a belt stretching along the diagonal direction, shown by Figure 4(d). Most points in the plot are in three clusters corresponding to the three stages of the execution. Though the program has 271 instances of the outermost phase, there is no specific correlation in the behavior between phases. The behavior of the present is closer to the behavior of the recent past than it is to an earlier phase. The quantitative measures in Table 2 are worse for phase instances because many instances are very small, and their behavior is not accurately measured by the 10ms intervals, the basic unit of the hardware counter.

Parser shows results similar to those of *LI*. Phase information does not lead to better characterization of program behavior. The IPC curve is shown in Figure 5(d), and the IPC hit rate distribution in Figure 4(e). Most points fall into three clusters. One cluster has significantly less variation in the miss rate than the IPC; another has the reverse. The quantitative measures in Table 2 are worse for phase instances than for coarse-grain intervals because of the most of the 703 phase instances are small and tend to exhibit more variation when measured by a coarse timer. However, the phase behavior is more consistent than the fine-grain intervals.

3.3 Correlation among metrics

Table 3 shows the correlation coefficients of the three metrics for our phases. The first row shows the results obtained on the mean values of phases, i.e. columns 4, 7 and 10 of Table 2. Less data access delay explains the positive correlation coefficient between cache hit rate and IPC. A shorter phase likely accesses fewer data, thus likely has higher cache hit rate, which explains the negative correlation coefficients between cache hit rate and phase length. Since higher cache hit rate likely happens with higher IPC, the IPC and phase length have negative correlation. The all positive correlation coefficients of the CVs show that if one metric of a phase varies much, the other two metrics likely vary much too. But the relation is weak between cache hit rate and IPC—only 15%. IPC and length ratio have the strongest relation, 53%. Cache hit rate has a 30% correlation coefficient with length ratio.

3.4 The temporal similarity of program phases

In the above sections we evaluated behavior similarity using the average cache hit rate, IPC and length of phase instances. Here we use two ad-hoc measures to compare the similarity of the temporal shape of the IPC curve of phase instances. The first is vertical similarity. For the IPC sequence of a phase instance, we divide the range of IPC into k equal segments, count the number of elements—10 ms intervals—whose IPCs fall in each range, and put the results in a k -cell vector. Then we divide the size of each vector cell with the total size. Each cell then represents the percentage

Table 4: Similarity of phase instances

Benchmarks	Vertical Similarity	Horizontal similarity		
		Horizontal similarity	Phase coverage	Time (%) coverage
Compress	96.8	97.1	24/25	99.8
GCC	87.0	71.2	57/57	100
LI	65.6	90.2	20/271	99.9
Parser	65.8	87.5	10/850	24.1
Vortex	78.7	99.1	30/30	100
Average	78.8	89.0	28/247	84.8

of the phase execution whose IPC falls in the given range. Since the last interval of a phase instance may be less than 10ms long, the elements are weighted by the interval length. After computing the vector for all phase instances, we calculate their average, weighted by the length of the instance. The difference between the vector of an instance and the average vector is the Manhattan distance between them, that is, the sum of the absolute difference of each cell. The average difference is the weighted average for all instances. The average vertical similarity, which we report in Table 4, is $1 - E/2$, where E is the average difference.

The vertical similarity shows, on average, how a phase instance differs from all other phase instances of the same subphase. It is a number between 0 and 1. The probability is α^{k-1} for two random vectors to have at least $1 - \alpha$ similarity, where k is the dimension of the vector [9]. In our experiment, k is chosen to be 5. As an example, *Compress* has a vertical similarity of 87%. The probability of two random vectors achieving such a similarity is 0.0003.

The similarity ranges, as shown in Table 4, from 66% to 97% across all programs. It is over 87% for *Compress* and *GCC*, 79% for *Vortex*, but only 66% for *Parser* and *LI*. The average is 79%. The results of *GCC* are for phase instances larger than 1 second across all program inputs. For *LI* we also consider all its reference inputs. A major reason for the sometimes low and often varying similarity is the effect of short phase instances. Because their length is close to the interval length of 10ms, the behavior variation is skewed when measured by intervals. Although *LI* has low vertical similarity, its IPC curve shows a very similar shape—namely, flat—across all phase instances. This reveals a problem of vertical similarity: when a signal has very small vertical variations, noise could have a large effect on the similarity value. Therefore, we also present horizontal similarity.

We measure horizontal similarity by dividing the length of each phase instance into k equal ranges, taking the average IPC for each range, and making a k -cell vector. Next we compute the relative vector, the average, the average difference, and the average horizontal similarity in the same way as we compute the vertical similarity. Not all phase instances are considered by this calculation because it needs at least k 10 ms intervals for the horizontal division. Table 4 gives the number of sufficient large phase instances and the number of all instances in the column marked as “Phase coverage”. These instances account for on average about 85% of the execution time, shown in the final column. *Parser* is the outlier, at 24%. This is because the top-level requests to *Parser* are small sentences, most of which require very little processing time. Excluding *Parser*, the average coverage becomes 99.9%.

Horizontal similarity is shown in the third column of Table 4, again for $k = 5$. It is over 97% for *Compress* and *Vortex*, and over 87% for *LI* and *Parser*. *GCC*'s similarity is the lowest, at 71%. This shows that large phase instances account for most of the execution

time, and they exhibit a very similar behavior pattern despite the difference in their length.

If we let $k = 10$ instead of 5, the vertical similarity numbers become lower, because of the finer partition. The horizontal similarity number becomes higher, because it excludes more smaller phases, and the benefit outweighs the loss of similarity due to the finer partition.

The temporal similarity measures confirm our previous observations. *GCC* and *Compress* have highly consistent phase behavior. *LI* and *Parser* have the least consistency. *Vortex* is between the two groups in terms of the behavior consistency.

4. RELATED WORK

Locality phases Early phase analysis, owing to its root in virtual-memory management, was intertwined with locality analysis. In 1976, Batson and Madison defined a phase as a period of execution accessing a subset of program data [5]. Later studies used time or reuse distance as well as predictors such as Markov models to improve virtual memory management. Recently, Shen et al. used reuse distance to model program behavior as a signal, applied wavelet filtering, and marked recurring phases in programs [15]. For this technique to work, the programs must exhibit repeating behavior. By using active profiling, we are able to target service-oriented programs, which typically do not have repeating behavior.

Program phases Allen and Cocke pioneered interval analysis to model a program as a hierarchy of regions [2]. For scientific programs, most computation and data accesses are in loop nests. A number of studies showed that inter-procedural array-section analysis accurately summarizes program data behavior. Recent work by 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 [11]. For general purpose architectures, Balasubramonian et al. [3], Huang et al. [12, 13], and Magklis et al. [14] selected as program phases procedures, loops, and code blocks whose number of instructions exceeds a threshold either during execution or in a profiling run. In comparison, our work does not rely on static program structure. It uses trace-based analysis to find the phase boundaries, which may occur anywhere and not just at region, loop, or procedure boundaries.

Interval phases Interval methods divide an execution into fixed-size windows, classify past intervals using machine or code-based metrics, and predict future intervals using last value, Markov, or table-driven predictors [3, 4, 7, 8, 10, 16]. Balasubramonian et al. searched for the best interval size at run time [4]. These studies showed benefits from interval-based adaptation for service-oriented programs. By accurately identifying phase boundaries, our work has the potential to improve prediction in service-oriented programs with varying but similar behavior across instances of different program phases.

5. CONCLUSIONS

This paper has presented a technique for marking program phases in service-oriented programs. By using a regular input, it induces repeating behavior. It then uses frequency-based filtering to identify phase boundaries and binary rewriting to insert phase markers. The markers divide a program execution into a sequence of dynamic phases.

Using our technique, we examined five representative service-oriented applications. Although phase lengths vary greatly even within a single execution, we find a high degree of correlation among corresponding phases on such time-adjusted metrics as IPC

and cache miss rate. In addition, phase lengths themselves are predictable with reasonable accuracy, allowing informed tradeoffs between optimization gains and adaptation overheads. These results suggest that program phases may provide valuable information for behavior prediction in service-oriented programs, extending the potential benefits of hardware and software adaptation to a significantly larger class of applications.

6. ACKNOWLEDGMENT

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.

7. REFERENCES

- [1] <http://www.specbench.org/>.
- [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] 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.
- [6] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Dept. of Computer Science, University of Wisconsin-Madison, June 1997.
- [7] A. S. Dhadapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working-set analysis. In *Proceedings of International Symposium on Computer Architecture*, Anchorage, Alaska, June 2002.
- [8] A. S. Dhadapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of International Symposium on Microarchitecture*, December 2003.
- [9] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
- [10] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.
- [11] C.-H. Hsu and U. Kermér. 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.
- [12] W. Liu and M. Huang. Expert: Expedited simulation exploiting program behavior repetition. In *Proceedings of International Conference on Supercomputing*, June 2004.
- [13] M. Huang and J. Renau and J. Torrellas. Positional adaptation of processors: application to energy reduction. In

- Proceedings of the International Symposium on Computer Architecture*, San Diego, CA, June 2003.
- [14] 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.
 - [15] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.
 - [16] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of International Symposium on Computer Architecture*, San Diego, CA, June 2003.
 - [17] 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.