SAND2022-9137C

IncProf: Efficient Source-Oriented Phase Identification for Application Behavior Understanding

Anonymous Anonymous
Anonymous Location
Anonymous Address
anonymous@email

***Abstract*—Long running applications often have varying behaviors, here called phases. While considerable work in computer architecture has been done in identifying application phases based on how the hardware is being exercised, comparatively less work has been focused on identifying application phases based on regions of source code being executed. In this paper we introduce a new methodology and an efficient tool framework, *IncProf*, for observing and capturing the time-varying source execution behavior of applications, and for then deducing application phases from the resulting data. Uses of this capability include simply better understanding the varying behavior of long running applications, and for efficiently tracking deployed application performance in the future by providing information to identify good instrumentation points.**

***Index Terms*—high performance computing, application monitoring, phase detection**

I. Introduction

Many applications cycle through periods of different behaviors that we can call *phases* [1]. These can be very large scale steps in an overall application, differing and configurable computations, or might be smaller scale behaviors that are repeated often. Being able to have, in deployment, tracking and performance introspection of these phases would help in providing developers and operations staff a better understanding of real-world application performance. A view into their application as it is used in deployment, broken down to the phase level, can aid in resource utilization and customer experience, and could elucidate places where application performance might be improved.

Our goal in this work is to automatically identify the phases of an application based on efficiently gathered data, and then to identify the parts of the source code that are representative of that phase. These source code places could then be used as places for efficient phase-identifying instrumentation, which would be usable in deployment and would provide the information needed to accomplish the above scenarios.

The main contribution of this paper is an algorithmic methodology and a tool framework, *IncProf*, for automatically identifying both the phases of an application and the representative source code places that could be instrumented to produce the phase logging data. This tool framework avoids the high slowdown factors of heavyweight program analysis tools, and thus can be used on runs of much more significant size than just “toy” sizes; this enables phase identification that will maintain its accuracy in production run sizes.

We also present a utilization of the phase discovery in a prototype *heartbeat* instrumentation framework, *AppEKG*, that efficiently collects data over the phases of an application. The resulting heartbeat data can then be used to analyze the performance of an application, or the system, on a per-phase level, thus gaining an understanding of which parts of an application are being used heavily, and which may need more tuning or optimization. We show that phase-oriented heartbeats can be utilized in production with very little overhead, with the proper instrumentation and data collection framework.

These ideas are applied and exemplified in this paper using applications in the realm of high performance computing (HPC). Monitoring the execution of scientific programs is crucial to the success of exascale computing, where the scale of resource and power usage demand more visibility about what is happening in the application and system during *production* runs; such visibility will help both users and administrators to work together to improve the usage and utility of their costly HPC infrastructure.

Sections II and III discuss the concepts of phases and heartbeats. Sections IV and V present the methodology, tools, and analysis framework. Section VI presents experimental results and their insight. Finally, Sections VII and VIII present related work, concluding thoughts, and areas of future work.

II. Phases

The idea of phases has taken different forms in the past, each of which entailed different fundamental conceptions of what a phase is. For example, hardware-based phase identification is useful for performance evaluation and especially for speeding up (“fast-forwarding”) hardware simulations for performance predictions of novel architectures [2], [3], [4], [5], [6]. In this case, a phase is distinguished by a pattern of how the hardware is exercised, observed using on-chip hardware performance counters.

Source-code phase identification, on the other hand, is useful for understanding the time-varying behavior of the application, for selecting instrumentation points that are efficient and yet cover the full execution, and for application-oriented performance evaluations. In this case, a phase corresponds to

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

a pattern of code execution. Some of the hardware-oriented work (e.g., [2]) uses application data (basic block sequences) but still orients towards hardware optimization, and incurs very high data collection overheads (e.g., slowdowns on the order of 10X).

We are taking the program-centric view of phases, that of code execution behavior, rather than the hardware-centric view. We desire the defined phases to represent unique patterns of the application, rather than unique profiles of hardware execution. Previous work has shown a degree of overlap [7], but in this work we need to be certain that we are identifying application-centric behavior patterns. Thus since our approach is to have phases that reflect application behavior instead of the hardware performance behavior, we define a **phase** as *a unique execution behavior that is reflected in a unique code execution profile*. This unique profile can then be used to identify instrumentation points in the application code that relate to phases, which can then produce heartbeats that can provide application phase performance information.

III. Heartbeats

Mechanisms and analyses for observing and evaluating the performance of systems and applications have been around since the beginning of computing, and in systems that are more complex and distant from the user—grid, cloud, and HPC clusters—observation capabilities are both more crucial and also harder to devise and deploy. For example, most HPC deployed monitoring systems focus on system performance and ignore any measure of application performance (e.g., [8]). Application heartbeats [9], [10], [11] were proposed as a lightweight mechanism for applications to be able to output information about their continued liveness, progress, and performance. In the original proposed form the heartbeat idea was connected with autonomic and autotuning ideas, where the application would publish target heartbeat rates and the autonomic environment would monitor the actual heartbeat and manage the application in order to try to keep it at the desired heartbeat rate.

Our idea here is much simpler—that applications would simply record heartbeats from their application phases, without any preconceived target rate or any autonomic computation steering. Monitoring tools will record at some level of detail the actual application heartbeat rates over the repeated use of the application by users, and as a history of an application is built up this data can be used to identify when the application is running poorly and when it is running well. Correlating the application heartbeat data with system data could help identify when system issues caused the poor performance. Others have also looked at using application heartbeats for application performance analysis [12].

In the HPC environment, many commonly used large applications do output their own logs, often including application-specific performance information. *The advantage of a heartbeat framework over this is that it is a generic concept across all applications and thus analyses can be developed that can be shared widely.*

A. Heartbeat Instrumentation

Our ultimate goal of this work is to have in-production observability of the performance of applications, at the phase level. Production-side instrumentation, by definition, must be minimally intrusive and low overhead. We have begun creating a heartbeat instrumentation framework for binary applications (i.e., compiled to native machine code) that is integrated into the LDMS data collection framework [13], which is used in high performance computing systems and is proven to be an efficient and scalable data collector. Our framework, *AppEKG*, is based on a heartbeat instrumentation API and can be used in a stand-alone fashion as well, without LDMS.

After experimenting with a simple, single "impulse" heartbeat capability, our design evolved into a two-step begin/end heartbeat API: a *beginHeartbeat(ID)* and an *endHeartbeat(ID)*; each unique heartbeat ID represents a unique phase of the application. Separating the begin/end events of a heartbeat allows analysis of heartbeat durations as well as the rate of heartbeat occurrences. Other than an initialization call, no other instrumentation is needed. The framework does not record every individual heartbeat but rather accumulates the number of heartbeats and their average duration during a specified collection interval; at the end of the interval, this data is then written out. Doing this controls the amount of I/O and allows the heartbeat framework to operate very efficiently, while still providing a record of the dynamic behavior of the software.

Such heartbeat data can be used in myriad ways, and our future work in *AppEKG* will involve researching effective ways of deriving performance results from this data. Section VI contains example figures and data from *AppEKG*.

IV. IncProf: An Incremental Profiling Tool

Observing and capturing the time-varying behavior of an application is often a very costly exercise. Quite a number of development-oriented tools can capture extremely detailed trace data regarding a program, but this often comes at a very high cost. For example, Pin [14] is widely used for detailed data collection, but Pin analyses typically have a 60-100X slowdown of the application. With this kind of overhead, obviously only very small runs can be used to capture data. Valgrind [15] is similar when detailed analyses are being performed. Various instrumentation techniques can instrument functions/methods for call tracing, and while more efficient than the above tools, these can generate very voluminous program traces.

On the other hand, some very efficient tools have been around for a long time. These tools, for efficiency reasons, do not capture detailed data but still capture useful insight into program execution. One of these is *gprof* [16], an efficient and ubiquitous call graph-based program profiling tool. Although the work in this paper uses *gprof* for its data collection, the methodology developed can be applied to data collected from other tools.[1]

[1] E.g., we have created proof-of-concept implementations for both the *gcov* and *JaCoCo* tools, and other profiling tools such as *OProfile* may be usable.

Gprof uses program counter sampling and function entry instrumentation to efficiently construct an accurate model of per-function performance, also breaking this down into parent and child function call performance statistics. *Gprof* has known limitations due to sampling (and sampling rate) and its blindness to I/O and other such performance effects, yet it continues to provide many users valuable insight into their applications.

Manual inspection of the Gnu implementation of *gprof* revealed that its runtime instrumentation support is included in the standard C library, and that it has one function that is responsible for writing out the data file.[2] We utilize this function by creating a preloadable shared library, that we call *IncProf*, that runs its own thread in a sleep/wakeup cycle, and at each wakeup it calls the *gprof* write function, renames the file to a unique sample name, and goes back to sleep. Figure 1 shows the process for collecting and reducing application profile data.

The application, of course, must be compiled with the "-pg" compiler option for *gprof* to be active, but other than this no further application instrumentation is needed during the data collection process. The application incurs the typical *gprof* overhead, which is very low, plus the overhead of our thread waking up, writing out the data file, and renaming the data file with an interval identifier. Overall, this overhead has always been less that 10% in our experiments, with a data write-out rate of once per second.

Although there is some documentation on the format of the binary files that *gprof* writes out, we found it easier to just invoke the *gprof* command line tool to convert the data into standard *gprof* textual reports, and then process those. *Gprof* produces both a flat per-function profile table, and then a table relating function profiles to particular calling contexts. The analysis presented here only uses the flat profile, though we have ongoing experiments with using the call-graph profile data to improve the results.

V. Overall Process Flow for Detecting Phases

With the above data resulting from our *IncProf* collection framework, we can now use this data to detect application phases and identify instrumentation sites. The tools that embody these analyses are distributed with and are considered part of the *IncProf* framework.

A. Identifying Phases

The incremental profile data is written out by *gprof* as totals since the beginning of the program, so the first step is to subtract the previous interval from each interval to create interval profile data. Each interval is then represented as a tuple of function execution times (the *gprof* 'self' time), where each unique function is an attribute dimension of the data.[3]

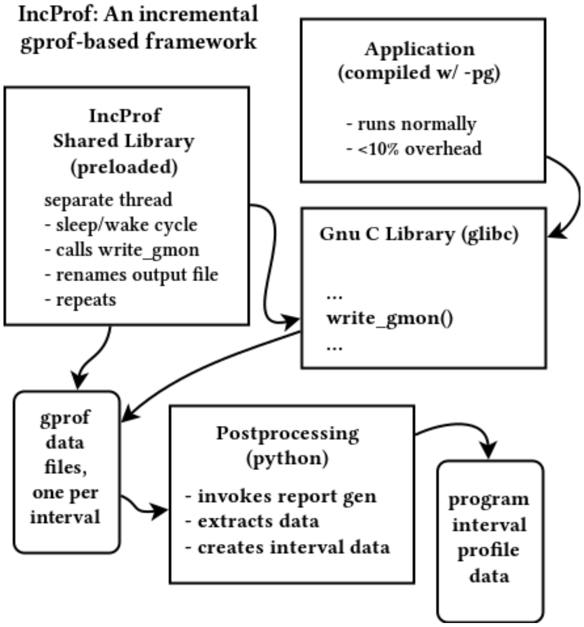


Fig. 1. Collecting Interval Data with *IncProf*.

Interval data is then clustered using the *k-means* clustering algorithm, and each cluster is interpreted as a phase of execution. However, since *k-means* requires that one know k ahead of time, we run *k-means* for $k = 1..8$, and then use the *Elbow* method to select the best number of clusters. These then are the phases. We have not had any applications where the number of phases discovered is greater than five, so eight as a maximum has worked well.

We have experimented with including or using other profiling data (number of calls, execution time of children, etc.) but have not found these to improve the results, and sometimes to worsen them; however we do continue to evaluate other combinations of data available from the profiling output. We have also experimented with other clustering algorithms (e.g., DBSCAN) but also have not seen improvements. We expect clusters over the profile data to have simple shapes and so algorithms that identify complex cluster shapes have not been found necessary for this context.

B. Identifying Instrumentation Points

As we create the interval function time tuple data, we also record the number of times each function is called in each interval. After finding the phases (clusters of intervals), the next step is to find places in the source code where phase heartbeat instrumentation could be placed—in other words, source code places that are active in, and representative of, the phase. Since our data is at the function level, this reduces to identifying functions.

[2] Although the code is in the C library, the symbol (function name) is not, and thus the code is hidden. We crafted an automated mechanism based on disassembling the library code to consistently find the data output function across different OS installations.

[3] Not all functions in a program end up being represented in the profile data, so this is not all of the functions in the program.

Algorithm 1 Instrumentation Identification Algorithm

```
    Input: Set $C$ of clusters over interval data set; each $C_i \in C$ is
    the set of intervals in cluster $i$
    Input: Function call count set $F$
    Input: Function rank set $R$
    Output: The set $P$ of phases, each $P_i \in P$ is a set
            of tuples $<function\ id, instrumentation\ type>$
 1: $P \leftarrow \{\}$
 2: for each $C_i \in C$ do
 3:     Sort $I$ in $C_i$ by distance to the cluster centroid
 4:     $P_i \leftarrow \{\}$
 5:     for each $I \in C_i$ do
 6:         {I is interval's tuple data (functions' self time)}
 7:         if $\exists f, f \in I \ \wedge f \in P_i$ then
 8:             continue
 9:         end if
10:         Using $F$ and $R$, sort $I$ by number of calls (ascending)
11:                         then rank (descending)
12:         $f \leftarrow$ first element in $I$
13:         if $f.calls > 0$ then
14:             $f.inst \leftarrow$ Body
15:         else
16:             $f.inst \leftarrow$ Loop
17:         end if
18:         if $<f.id, f.inst> \notin P_i$ then
19:             $P_i.add(<f.id, f.inst>)$
20:         end if
21:     end for
22:     $P.add(P_i)$
23: end for
24: Output $P$
```

To avoid low-level, library-type functions, we preference functions that have a lower number of calls in the interval. We also want to preference functions that are active in many intervals in the phase, so we create a per-function, per-phase *rank*, which is the fraction of intervals in the phase that the function is active in (i.e., has a non-zero execution time).

Using the clustered intervals, function call counts, and ranks, Algorithm 1 selects the instrumentation sites for each phase. Instrumentation sites are functions and are tagged by one of two designations: *body*, which means that the function body can be instrumented (essentially that the instrumentation can be inserted at the start and end of the function), or *loop*, which means that a loop within the function body needs instrumented (the instrumentation point should be within the body of a loop in the function). Since *gprof* only records function-level data, our algorithm cannot identify exactly which loop in the function that needs to be instrumented, and so this detail is still future work, and is discussed later.

Algorithm 1 identifies instrumentation points for each phase, and is described as follows:

- the outer loop iterates through all clusters (all phases);
- line 3 sorts the intervals in the cluster by their distance to the centroid of the cluster; this places intervals that are most representative of the cluster first in processing order;
- the inner loop (line 5) iterates through the intervals in the cluster (phase);
- lines 7-9 checks to see if this interval is already covered by some previously selected instrumentation site for this phase; if so, it can be skipped;
- line 10 takes the current interval's function data and sorts the functions first by the number of calls (ascending) and then by rank (descending);
- line 11 then takes the topmost function from this sort as the function to instrument in order to cover this interval;
- lines 12-16 tag this function's instrumentation type as *body* if there are any calls to the function in the interval, or *loop* if there are no calls (meaning the function has continued to execute from being invoked previously);
- lines 17-18 add this function's instrumentation type to the set of output instrumentation sites for this phase, if it is not already in it;
- line 21 adds the new phase to the output phase set.

At the end of this algorithm, the phases will be identified and each phase will have a set of instrumentation sites identified that fully cover the intervals that are included in that phase (our implemented algorithm does allow a coverage threshold, to skip outliers; in our results we use a 95% threshold). This algorithm is essentially a greedy algorithm that does not backtrack to try and make different decisions, but there are two keys that help in selecting good instrumentation sites for phases. One, by starting with intervals that are closest to the center of the cluster, it is expected that the most representative instrumentation sites will be selected first, and these should cover the most number of intervals as they are picked; later we discuss the issue of alternatives for dealing with outlier intervals. Two, by sorting active functions within an interval by the number of calls it has in the interval, we are preferencing functions with fewer calls and thus longer execution times. The goal of this is to avoid selecting very short non-distinguishing functions, such as getters and setters or other utility functions that might be called many times, even in one interval.

VI. Evaluation and Analysis

Our initial exploration of this research has been performed in the context of understanding the time-varying behavior of high performance computing applications. Thus, in this section, we discuss the experimental results for 5 scientific applications (two real and three proxy/benchmark), and we show how our phase identification captures the execution phases for each application. Phase identification is shown by the time-varying activity of the heartbeats from the individual heartbeat instrumentation sites. Instrumentation sites are presented individually, even if they belong to the same phase, which allows us to evaluate and understand how multiple instrumentation sites may together represent a single phase.

All of these applications are highly parallel, using MPI to communicate among distributed processes (MPI ranks). In the discussion below we are processing the profile samples, and showing heartbeat data, from just one representative process (MPI rank) in the application; our framework does produce profiles and heartbeats from all processes in an application, but at present we only use all the data for aggregate descriptive

TABLE I
EXPERIMENTAL OVERVIEW: SETUP & OVERHEAD

App	Procs / Nodes	Uninstr Runtime (sec)	IncProf Ovhd (%)	Heartbeat Ovhd (%)	# Phases Discov.
Graph500	1 / 1	188	10.1	1.6	4
MiniFE	16 / 2	617	-6.2	1.1	5
MiniAMR	16 / 2	459	1.5	0.2	2
LAMMPS	16 / 2	307	7.5	8.1	4
Gadget	16 / 2	421	6.4	1.0	3

statistics. All of the applications being used are symmetrically parallel and thus all processes behave similarly.

Table I shows the configuration data describing the application runs that we evaluated, and the general overhead and phases detected results. We selected input and configurations that resulted in runs on the order of 5-10 minutes. Experiments were conducted on a heterogeneous multi-use cluster; we used 2 homogeneous nodes, each having two AMD EPYC 7282 2.8GHz CPUS with a total of 32 cores, and 512GB of RAM. The interconnect is Infiniband HDR.

Our *IncProf* sampling rate was set to one second, in order to achieve 1-second intervals and produce an analysis that results in instrumentation sites valid at this fine-grained level. Even with all MPI ranks writing out incremental profile data files, the overhead of using *IncProf* is 10% or less[4] (Table I), which is much better than many program analysis tools. This overhead is only during profile collection for use in phase detection; it does not occur during the heartbeat experimentation step. We applied a threshold cutoff for instrumentation site selection of 95%, meaning that once selected sites covered that much of the intervals in a phase, no further site selection was done.

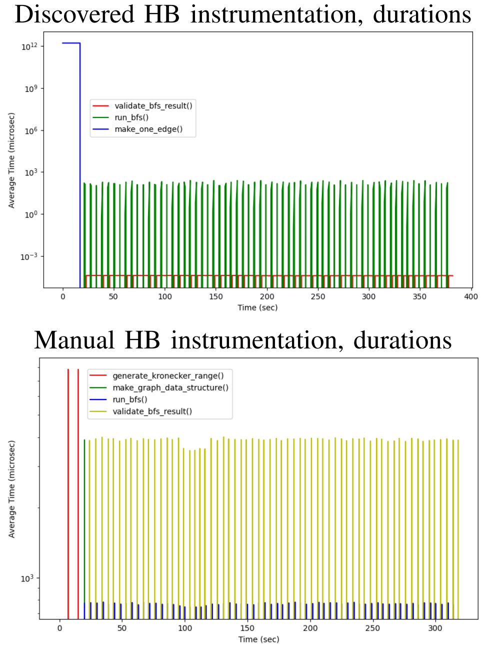
In the results below, each instrumentation site is shown with the amount of the phase it covers, and the amount of the entire profiled application run it covers. This gives a measure of how significant the site is.

We used *AppEKG*, our heartbeat instrumentation framework, in two ways: one, we inspected the application code and instrumented each application with what we consider to be the "best" heartbeat instrumentation places. Two, we instrumented the sites chosen by our phase discovery methodology. This offers a comparison for how well the discovery methods work as compared to a human understanding of the application. In this paper we present raw heartbeat plots to visualize how the discovery process performed, we do not present any heartbeat performance analysis, which is outside the scope of this paper. The heartbeat overhead in Table I reflects the overhead for the manual "best" heartbeat instrumentation. All applications but LAMMPS has extremely low overhead, and LAMMPS is only at 8%; in-development *AppEKG* modifications can lower this significantly, but such preliminary results are not shown here.

The following subsections discuss the results for each application, including the phases detected, their selected instrumentation sites, and instrumentation characterization.

TABLE II
GRAPH500 INSTRUMENTED FUNCTIONS

Phase ID	HB ID	Discovered Site Function	Phase %	App %	Inst. Type
0	1	validate_bfs_result	98.1	62.2	loop
1	2	run_bfs	100	13.2	body
2	3	run_bfs	100	12.3	loop
3	4	make_one_edge	97.2	10.8	body
Manual Instrumentation Sites					
		make_graph_data_structure			body
		generate_kronecker_range			body
		run_bfs			body
		validate_bfs_result			body


Discovered HB instrumentation, durations

Manual HB instrumentation, durations

Fig. 2. Graph500 Phase Heartbeats.

A. Graph500

Graph500 [17] is a benchmark developed to evaluate the performance of HPC systems on graph problems; in this experiment we used the *mpi_simple* version of the the 2.1.4 version of the benchmark. This version creates a large graph data structure, and then performs breadth-first searches over the graph, and checks (validates) the result of the searches.

Table II shows our phase discovery results. Our phase analysis discovers four phases in Graph500, each with one instrumentation site, but two of the phases have the same function as an instrumentation site, with one phase designating it as a body-type site, while the other designates it as a loop-type. For the main computations (search and validate), the discovered sites are the same as our manual instrumentation selection (modulo a body/loop choice). For the initialization portion of the run, the discovered site is a lower-level function than the two that we selected.

[4] For MiniFE, compiler optimization levels change its overhead significantly, but O3 consistently produced negative overhead; we are investigating.

TABLE III
MINIFE INSTRUMENTED FUNCTIONS

Phase ID	HB ID	Discovered Site Function	Phase %	App %	Inst. Type
0	1	sum_in_symm_el...	100	19.5	body
1	2	cg_solve	100	43.7	loop
2	3	init_matrix	93.2	10.1	loop
2	4	generate_matrix_structure	6.8	0.7	loop
3	5	impose_dirichlet	100	4.4	loop
4	2	cg_solve	94.7	20.5	loop
4	6	make_local_matrix	2.7	0.6	loop
Manual Instrumentation Sites					
		cg_solve			loop
		perform_elem_loop			loop
		init_matrix			loop
		impose_dirichlet			loop
		make_local_matrix			loop

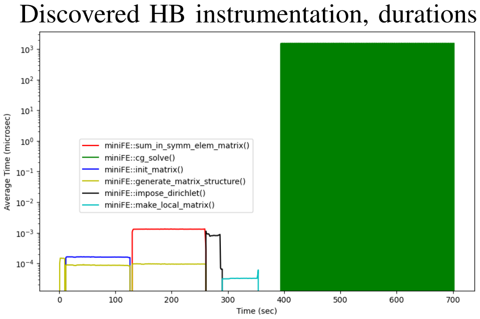


Fig. 3. MiniFE Phase Heartbeats.

TABLE IV
MINIAMR INSTRUMENTED FUNCTIONS

Phase ID	HB ID	Function	Phase %	Inst. Type	Rate Factor
0	1	check_sum	100	89.1	body
1	2	allocate	33.8	3.7	loop
1	3	pack_block	32.4	3.5	body
1	4	unpack_block	26.5	2.9	body
Manual Instrumentation Sites					
		check_sum			body
		stencil_calc			body
		comm			body

The heartbeat plots in Figure 2 help explain the differences; shown are the average heartbeat duration in each interval, for both the discovered instrumentation sites and the manual ones.

Essentially, the manual heartbeat sites selected all typically run longer than our interval size (one second). Due to the interval-sampling data collection mechanism, these heartbeats do not show up in all the intervals, only those that they finish in, and our manual-site heartbeat counts are never more than one in any interval. In the initialization phase, the manual sites thus have gaps, and the discovered heartbeat site, being lower-level and faster, does not. Also, phase discovery separated as two distinct phases those intervals that *run_bfs* was called in and those where it simply continued to run; this is the reason for one cluster indicating the function body should be instrumented, while the other indicates a loop in the function should be.

Arguably, the discovered sites better capture the behavior than our manual sites. Thus overall, we deem this result good, and yet it points out areas of improvement: our manual selection perhaps should have instrumented loops, our phase discovery might need some postprocessing to combine phases which have the same instrumentation sites, and better analysis or plotting of heartbeats that are longer than data collection intervals could be done.

B. MiniFE

MiniFE is a finite element mini-application developed to represent an implicit finite-element application kernel, and is part of the Mantevo mini-application suite [18]. MiniFE, as described in its documentation, uses four kernels: the first generates the matrix/vector mesh structures, the second assembles the mesh into sparse matrices, the third performs sparse matrix operations during a conjugate-gradient solver, and the fourth performs various vector operations.

As shown in Table III, our phase analysis finds five phases, and selects six different instrumentation sites (one repeated) to represent those phases. Our manual heartbeat instrumentation identified five sites, four of which are also selected by the phase discovery; Figure 3 only shows the discovered heartbeats, since they are nearly identical. The *cg_solve* heartbeat captures the main solver computation, with the other four being preparation phases (in our runs, the main computation is not run for a long, extended period of time, so the initialization phases appear fairly large). All heartbeats are relatively stable in behavior, although some have some spiking at various ends. The graph region with *cg_solve* active appears almost solid; this is because the heartbeat oscillates between 0 and 1 in successive interval data.

The discovered *generate_matrix_structure* heartbeat site is active across other varying heartbeats, and so is not considered by us to be a good selection; the *sum_in_symm_elem_matrix* heartbeat is invoked from and is essentially equivalent in behavior to our manual *perform_element_loop* heartbeat; extending the discovery analysis to use the call-graph structure might be a way to improve it and select our site, which is higher up in the call graph.

C. MiniAMR

MiniAMR is a proxy that represents applications based on adaptive mesh refinement; it applies a stencil computation over a mesh that adaptively refines and coarsens as objects move through it [19].

Table IV shows the phases and instrumentation sites for the two discovered phases, with the second phase having three discovered heartbeat instrumentation sites. Only the first phase, along with its heartbeat site, matches our manual heartbeat instrumentation selection, but this phase covers almost 90% of the execution. We should note that the *check_sum* heartbeat site is not a function that performs a simple mathematical checksum but rather embodies more involved matrix computations.

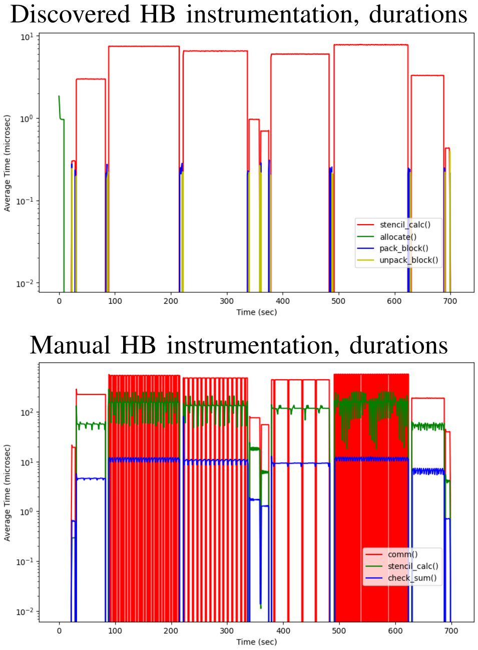


Fig. 4. MiniAMR Phase Heartbeats.

Figure 4 shows the time varying behavior of the heartbeats. In comparing the plots of the discovered heartbeat sites and the manual sites, our three manual sites are simultaneously active, not really capturing different phase behavior. On the other hand, the discovered heartbeats for phase 1 are clearly capturing the periods in the execution where the "normal" computation is deviated from. The large and varied deviation in the middle is a mesh adaptation, while the smaller periodic deviations are large communication steps. In our manual instrumentation, we observed in the code an iteration over the three main functions we chose as heartbeat sites, but these are all active together and so our discovery analysis, which tries *not* to overlap heartbeats, would not select all of them. Still, an application developer would probably be interested in statistical analysis of the performance of these three heartbeats as they seek to improve their application.

Overall, the phase discovery did well in selecting phases that distinguish application behavior, but also missed what manual code inspection saw as important (interleaved) application steps. This points to future work in perhaps integrating whole-execution summary data, to include overlapping high-time program regions, with the interval data.

D. LAMMPS

The Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) is a classical molecular dynamics application [20] that is used in much active research. LAMMPS is a large application that can be used in several different modes that simulate a variety of combinations of molecules and force types. In this work we chose the metal type atoms with the Lennard-Jones (LJ) force model. After initialization and atom creation, the application has one main core computation, that of using the LJ force computation algorithm to simulate the interaction between atoms. We recognize that our analysis here does not capture what would be needed to recognize phases in, and find instrumentation sites for, other modes of LAMMPS. For full heartbeat instrumentation of LAMMPS, further analyses would be needed, and large multi-mode applications like LAMMPS should really be thought of as a collection of related applications, each having unique but related phase behavior.

TABLE V
LAMMPS INSTRUMENTED FUNCTIONS

Phase ID	HB ID	Discovered Site Function	Phase %	App %	Inst. Type
0	1	PairLJCut::compute	100	55.7	loop
1	2	NPairHalf...::build	100	7.7	loop
2	1	PairLJCut::compute	100	34.1	loop
3	2	NPairHalf...::build	50	1.3	body
3	4	Velocity::create	42.9	1.1	loop
Manual Instrumentation Sites					
		PairLJCut::compute			body
		NPairHalf...::build			body

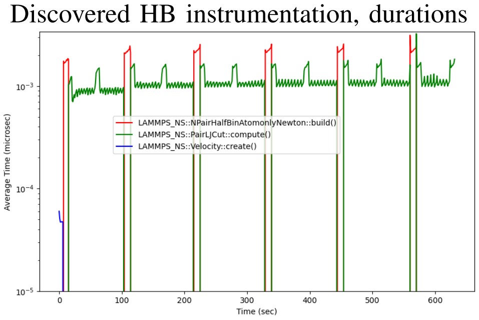


Fig. 5. LAMMPS Phase Heartbeats.

Table V shows our phase analysis finding four phases, but only three different instrumentation sites. The two doubly-indicated sites are the same as our manual instrumentation effort selected. Phases 0 and 2, with the *PairLJCut::compute* site, make up almost 90% of the execution, and should really be identified as a single phase. Phase 3 is only 2.4% of the execution, and half of it is covered by the *NPairHalf::build* site, which is also covering all of phase 1.

Figure 5 shows that the application is dominated by phase 0/2 and short intervals of the phase 1/3 heartbeat *NPairHalf::build*, with the other phase 3 heartbeat (*Velocity::create*) only occurring at the beginning, and thus being an initialization function. Inspecting the application source indicates that *PairLJCut::compute* is the main point for re-computing the LJ force model used in this execution, while *NPairHalf::build* occurs in preparing data for parallel communication.

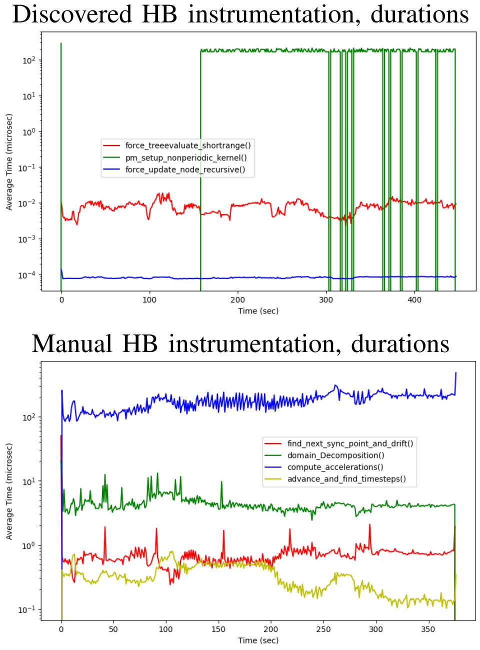
Figure 5 shows just the discovered heartbeat plot, since it subsumes the manual sites that were selected.

E. Gadget2

Gadget2 is a simulation for a cosmological N-body/SPH problem; it computes gravitational forces using a hierarchical

TABLE VI
GADGET2 INSTRUMENTED FUNCTIONS

Phase ID	HB ID	Discovered Site Function	Phase %	App %	Inst. Type
0	1	force_treeevaluate_shortrange	100	44.9	body
1	2	pm_setup_nonperiodic_kernel	93.8	28.6	body
1	3	force_update_node_recursive	5.9	1.8	body
2	1	force_treeevaluate_shortrange	100	24.7	body
		Manual Instrumentation Sites			
		find_next_sync_point_and_drift			body
		domain_decomposition			body
		compute_accelerations			body
		advance_and_find_timesteps			body


Discovered HB instrumentation, durations

Manual HB instrumentation, durations

Fig. 6. Gadget2 Phase Heartbeats.

tree algorithm and represents fluid behavior by means of smoothed particle hydrodynamics (SPH) [21], and is used in significant cosmological research. Gadget2 combines N-body simulation with hydrodynamic forces for large-scale cosmological simulations. As with many scientific simulations, it is timestep-based, recomputing particle densities, accelerations, and positions over a timestep-driven loop with four main function calls in it. Gadget2 is interesting in that these different parts of the computation loop occur quickly, and thus our one-second interval-based analysis does not to a good job in detecting different phases in the computation.

Table VI shows our phase analysis of Gadget2, with framework producing three phases with three different instrumentation sites (one site shared). For our manual instrumentation we selected the four main timestep functions. Figure 6 shows the heartbeats over time. Because the main timestep loop always calls each of the four main functions once per loop, our manual heartbeat sites result in a plot where all four lines essentially overlap each other, modulo intervals where a single count is missed (the interval ends in the middle of the main loop). All three discovered heartbeat sites are called indirectly from *compute_accelerations*, which is responsible for about 75% of the execution time. Since the discovery analysis is working from one-second interval profile data, which may sometimes partition the main computation loop disadvantageously, the phase detection does see enough variation to decide there are three phases (k-means clusters), and tries to select heartbeat sites representative of these.

Gadget2 is an example of an application that, when we look at the code, clearly has four main computation steps, each of which should be tracked with a heartbeat to enable an understanding of that portion of execution performance, yet none are long-running phases that can be detected with our phase analysis. This points to a need for an alternative analysis scheme for applications with fast phases.

VII. RELATED WORK

Beyond the large body of work in phase detection based on measuring hardware usage, for simulation improvement, architecture improvements, and scheduling, and cited in Section II, other work also related to our work is described below.

Nickolayev et al. [22] is a classic work in which statistical online clustering was performed in real-time over a stream of metrics (events) being collected from an application. This idea focused on real-time analysis and selected application-specific metrics, and showed that the statistical clustering could track the known phases of the application. Our focus is on a generic method that can be applied to all applications. Other related program analysis work can be found in [23], [24], [25], [26], [27], [28], which due to space are undescribed here.

Other forms of profile data, e.g., the Context Execution Tree of [29], might be useful for our work. Work in online performance monitoring and analysis, e.g., [30] can also be relevant, since such work is processing incremental performance data. Others have looked at using trace data to understand software phases in systems [31], [1], [32], [33], [34], [35], including sampling of traces, e.g., [36].

VIII. CONCLUSION

As shown across the applications described above, our attempt at automatically detecting software-based phases of applications has promise, but also has significant room to improve. Using other data sources, and even using the profile data differently, may improve its performance, and many other ideas for potential future work are too numerous to catalog here. Even this preliminary work, though, has some important takeaways. One is that real applications do have phases of different behavior, although they can look different (e.g., overlapping or sequenced). Two is that heartbeat data, even just visually, can capture how those phases are behaving. Three, future analyses developed for heartbeat data can provide portable, consistent, and quantitative evaluation of scientific application performance.

References

[1] S. P. Reiss, "Dynamic detection and visualization of software phases," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 16, may 2005. [Online]. Available: https://doi.org/10.1145/1082983.1083254

[2] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2. ACM, 2003, pp. 336–349.

[3] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*. IEEE, 2003, pp. 220–231.

[4] D. M. Pase, "Dynamic probe class library (dpcl): Tutorial and reference guide," *Version 0.1. Draft document, IBM Corporation*, vol. 552, 1998.

[5] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W.-m. W. Hwu, "Vacuum packing: Extracting hardware-detected program phases for post-link optimization," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 2002, pp. 233–244.

[6] B. Davies, J. Bouguet, M. Polito, and M. Annavaram, "ipart: An automated phase detection and recognition tool," Technical report, Intel Research Tech Report IR-TR-2004-1 (http://research. intel. com/ir/tools/presentations/files/IR-TR-2004-1-iPART. pdf), Tech. Rep., 2003.

[7] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic behavior and Simulation Points in Applications," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. IEEE, 2001, pp. 3–14.

[8] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[9] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809065

[10] H. Hoffmann, J. Eastep, M. Santambrogio, J. Miller, and A. Agarwal, "Application heartbeats for software performance and health," *ACM SIGPLAN Notices*, vol. 45, pp. 347–348, 09 2009.

[11] M. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. Malony, T. Sterling, and R. Fowler, "An autonomic performance environment for exascale," *Supercomput. Front. Innov.: Int. J.*, vol. 2, no. 3, pp. 49–66, Jul. 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026759.3026764

[12] E. S. Buneci and D. A. Reed, "Analysis of application heartbeats: Learning structural and temporal features in time series data for identification of performance problems," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2008, pp. 1–12.

[13] A. Agelastos *et al.*, "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *SC'14: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 154–165. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.18

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[15] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, p. 89100, jun 2007. [Online]. Available: https://doi.org/10.1145/1273442.1250746

[16] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 120–126.

[17] D. Bader, J. Berry, S. Kahan, R. Murphy, E. Riedy, and J. Willcock, "Graph 500 Benchmark 1 (Search)," 2010, www.graph500.org.

[18] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[19] C. T. Vaughan and R. F. Barrett, "Enabling Tractable Exploration of the Performance of Adaptive Mesh Refinement," in *2015 IEEE International Conference on Cluster Computing (CLUSTER)*, 2015, pp. 746–752.

[20] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, Mar. 1995. [Online]. Available: http://dx.doi.org/10.1006/jcph.1995.1039

[21] V. Springel, "The Cosmological Simulation Code Gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, no. 4, pp. 1105–1134, 2005.

[22] O. Y. Nickolayev, P. C. Roth, and D. A. Reed, "Real-time statistical clustering for event trace reduction," *Int. J. High Perform. Comput. Appl.*, vol. 11, no. 2, p. 144159, jun 1997. [Online]. Available: https://doi.org/10.1177/109434209701100207

[23] J. Zhang and R. J. Figueiredo, "Application classification through monitoring and learning of resource consumption patterns," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. USA: IEEE Computer Society, 2006, p. 144.

[24] T. Sondag and H. Rajan, "Phase-based tuning for better utilization of performance-asymmetric multicore processors," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. USA: IEEE Computer Society, 2011, p. 1120.

[25] J. Wang, "Constraint-based event trace reduction," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 11061108. [Online]. Available: https://doi.org/10.1145/2950290.2983964

[26] S. Mühlbauer, S. Apel, and N. Siegmund, "Accurate modeling of performance histories for evolving software systems," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 640652. [Online]. Available: https://doi.org/10.1109/ASE.2019.00065

[27] D. Licata, C. Harris, and S. Krishnamurthi, "The feature signatures of evolving programs," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, 2003, pp. 281–285.

[28] M. Chabbi, W. Lavrijsen, W. de Jong, K. Sen, J. Mellor-Crummey, and C. Iancu, "Barrier elision for production parallel programs," in *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, vol. 2015, 02 2015.

[29] T. Kumar, J. Sreeram, R. Cledat, and S. Pande, "A profile-driven statistical analysis framework for the design optimization of soft real-time applications," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 529532. [Online]. Available: https://doi.org/10.1145/1287624.1287702

[30] J. Kroß, F. Willnecker, T. Zwickl, and H. Krcmar, "Pet: Continuous performance evaluation tool," in *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, ser. QUDOS 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 4243. [Online]. Available: https://doi.org/10.1145/2945408.2945418

[31] Y. Watanabe, T. Ishio, and K. Inoue, "Feature-level phase detection for execution trace using object cache," in *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, ser. WODA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 814. [Online]. Available: https://doi.org/10.1145/1401827.1401830

[32] T. Mizouchi, K. Shimari, T. Ishio, and K. Inoue, "Padla: A dynamic log level adapter using online phase detection," in *Proceedings of the 27th International Conference on Program Comprehension*, ser. ICPC '19. IEEE Press, 2019, p. 135138. [Online]. Available: https://doi.org/10.1109/ICPC.2019.00029

[33] Y. Feng, K. Dreef, J. A. Jones, and A. van Deursen, "Hierarchical abstraction of execution traces for program comprehension," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: Association for

Computing Machinery, 2018, p. 8696. [Online]. Available: https://doi.org/10.1145/3196321.3196343

[34] L. Alawneh, A. Hamou-Lhadj, and J. Hassine, "Segmenting large traces of inter-process communication with a focus on high performance computing systems," *Journal of Systems and Software*, vol. 120, pp. 1–16, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216300954

[35] T. Ishio, Y. Watanabe, and K. Inoue, "Amida: A sequence diagram extraction toolkit supporting automatic phase detection," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 969970. [Online]. Available: https://doi.org/10.1145/1370175.1370212

[36] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, L. Alawneh, and A. Shafiee, "Stratified sampling of execution traces: Execution phases serving as strata," *Science of Computer Programming*, vol. 78, no. 8, pp. 1099–1118, 2013, special section on software evolution, adaptability, and maintenance & Special section on the Brazilian Symposium on Programming Languages. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642312002080

[37] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[38] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *Proceedings of the 4th international conference on Computing frontiers*. ACM, 2007, pp. 143–152.

[39] M. Tanash, N. Ghazanfari, O. Aaziz, and J. Cook, "Automatically instrumenting scientific applications to produce heartbeat events," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 1678–1686.