Compact Application Signatures for Parallel and Distributed Scientific Codes *†

Charng-da Lu Daniel A. Reed
{clu2,reed}@cs.uiuc.edu

Department of Computer Science
University of Illinois
Urbana, Illinois 61801 USA

Abstract

Understanding the dynamic behavior of parallel programs is key to developing efficient system software and runtime environments; this is even more true on emerging computational Grids where resource availability and performance can change in unpredictable ways. Event tracing provides details on behavioral dynamics, albeit often at great cost. We describe an intermediate approach, based on curve fitting, that retains many of the advantages of event tracing but with lower overhead. These compact "application signatures" summarize the time-varying resource needs of scientific codes from historical trace data. We also developed a comparison scheme that measures similarity between two signatures, both across executions and across execution environments.

1 Introduction

Developing applications that achieve high performance on parallel systems often requires multiple iterations of performance analysis and refinement. Moreover, for complex, data dependent applications, performance measurements from one execution may not be representative of future executions, nor may they highlight critical bottlenecks. This is even more true for emerging applications that execute atop computational Grids [1], where resource availability and performance can change in unexpected ways [2, 11].

Event tracing is the standard approach for obtaining detailed data on application execution dynamics. Although event traces provide the requisite detail to understand execution dynamics, they are often voluminous, especially when analyzing highly parallel codes. Conversely, statistical profiles are compact but lack detail.

In this paper, we describe an intermediate approach, based on curve fitting, that retains many of the advantages of event tracing but with lower overhead. These compact "application signatures" summarize time varying application behavior while still retaining the compactness of statistical summaries.

Moreover, signatures provide a natural, temporal complement to standard comparison of execution profiles. Such profile comparisons highlight *where* in the code the distribution of execution time may have shifted. Similarly, signature comparisons identify *when* two executions differ. To support signature comparison, we describe a scheme to measure the similarity between two signatures, both across executions and across execution environments.

The remainder of this paper is organized as follows. In §2, we describe the requirements for application signatures and their relation to program models and performance verification. In §3-§4, we describe algorithms for signature generation and comparison. This is followed by experimental results and discussions in §5. In turn, §6 reviews related work and §7 summarizes our results and outlines possible directions for future work.

2 Application Signatures

Event tracing is the standard approach for obtaining detailed data on the execution dynamics of sequential and parallel applications. For a particular event type, each (time, metric value) pair defines a two-

*This work was supported in part by the National Science Foundation under grants NSF EIA-99-72884 and ASC 97-20202; by the Department of Energy under contracts DOE W-7405-ENG-36, LLNL B341494, DOE SciDAC DEFC02-01ER41205, and LLNL B505214; and by the NSF Alliance PACI Cooperative Agreement.

†0-7695-1524-X/02 $17.00 ©2002 IEEE

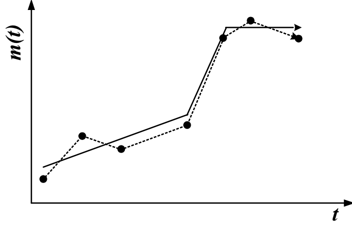


Figure 1: Metric Trajectory and Polyline Fit

dimensional metric point. All the metric points then collectively define a function $m(t)$ where time is the independent variable, and the metric value is the dependent variable.

Conceptually, this $m(t)$ defines a performance metric trajectory that traces a curve in the plane. As an example, Figure 1 illustrates a performance metric trajectory of seven metric points and a polyline fit to the data. More generally, instrumentation that captures N metrics defines a trajectory in an $(N+1)$-dimensional metric space.

Application signatures are a compact representation of these trajectories. Below, we outline the desired features of these signatures and their use in performance verification via contracts.

2.1 Signature Requirements

Our concept of signatures is a compression of trace data that captures performance metric dynamics while minimizing loss of detail. Ideally, signatures should have the following properties:

- *Conciseness.* Signatures should be physically compact, enabling description of long-running codes.
- *Execution context invariance.* Signatures should capture an application's intrinsic behavior in a way that is not dependent on the extrinsic execution environment.
- *Easy acquisition.* Signatures should be captured in real-time with minimal computation overhead.
- *Recognition.* Signatures from two executions should be comparable, and the comparison should reflect intuitive notions of similarity.

To meet these requirements, we propose a signature mechanism based on polylines and markers. A polyline (a series of piecewise linear segments) provides a compact representation of performance metric trajectories whose compression ratio and information loss can be adjusted by changing the number segments.

A set of trajectory markers embedded in the signatures define standard reference points across executions on differing platforms. By synchronizing comparison at the marker reference points, one can compare behaviors at known points. Moreover, marker-based signatures enable performance validation via the notion of performance contracts.

2.2 Performance Contracts

The motivation for performance signatures is quantitative comparison and validation of application execution dynamics across platforms, input data sets, and configurations. In this model, each application has a performance contract [11] that may specify performance expectations, scheduling constraints, and other requirements.

We envision application signatures being used as shown in Figure 2. The application signature module accepts real-time performance data and a performance signature from previous executions, compares historical and real-time signatures, and then provides feedback to the adaptive control module.

The *model signature* (or *model*), which is a part of the performance contract and is created during a previous run of the same application, is loaded prior to application runtime. During execution, the application signature module receives performance data from the performance monitor and generates the *observation signature* (or *observation*) in real-time. The comparison is invoked periodically to calculate the degree of similarity between the *model* and the *observation* and uses this feedback for dynamic tuning and reconfiguration of resources.

3 Signature Creation

With the context of §2, an application signature consists of two components: a polyline fitting the trajectory and a set of markers for signature recognition. Below, we describe both polyline fitting and marker generation.

3.1 Polyline Fitting

Many curves can be used to fit a sequence of points. For simplicity, we have chosen to use polylines. A wide variety of other alternatives exist (e.g., higher-degree polynomials, splines, time series, or feature/line identification from images), with a broad range of implementation complexity and overhead.

Pragmatically, it is impractical to fit a single curve to any data set of unknown volume and distribution

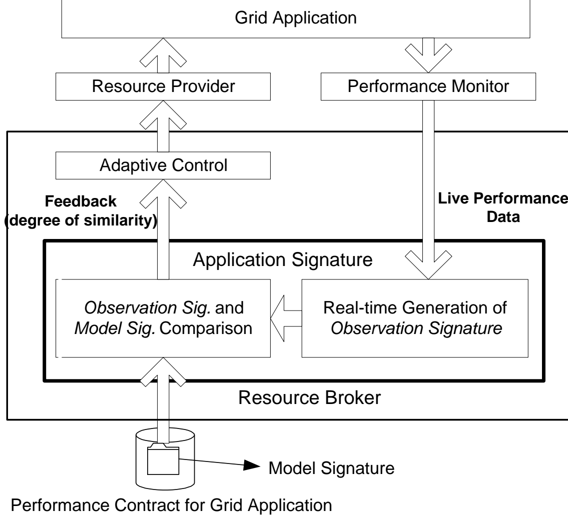


Figure 2: Performance Contract Framework

and expect a good fit. Conversely, polylines are flexible enough to handle arbitrary data sets. Equally importantly, there are efficient algorithms for computing similarity between two polylines.

Our polyline generation algorithm is based on least squares linear fitting [10], which minimizes

$$\begin{aligned} \delta(a,b) &= \sum_{i=1}^{N} \frac{(q(t_i)-m_i)^2}{N} \\ &= \frac{1}{N}\Big(a^2N + 2ab\sum t_i - 2a\sum m_i - \\ &\quad 2b\sum t_i m_i + b^2\sum t_i^2 + \sum m_i^2\Big) \end{aligned} \quad (1)$$

where $q(t) = a + bt$ is the linear fit and

$$(t_i, m_i) \quad i = 1, \ldots, N$$

are the (time, metric value) pairs. Computing the partial derivatives of $\delta(a,b)$ with respect to a and b and setting them equal to zero, we have

$$a = \begin{vmatrix} \sum m_i & \sum t_i \\ \sum t_i m_i & \sum t_i^2 \end{vmatrix} \begin{vmatrix} N & \sum t_i \\ \sum t_i & \sum t_i^2 \end{vmatrix}^{-1} \quad (2)$$

$$b = \begin{vmatrix} N & \sum m_i \\ \sum t_i & \sum t_i m_i \end{vmatrix} \begin{vmatrix} N & \sum t_i \\ \sum t_i & \sum t_i^2 \end{vmatrix}^{-1} \quad (3)$$

With this context, our polyline algorithm accepts a sequence of metric points (t_i, m_i) and generates a series of line segments in real-time. Figure 3 summarizes this algorithm.

Initially, one line segment is used to fit the data. The algorithm then attempts to stretch the line segment by fitting as many metric points as possible until the measure of error exceeds a user-specified

Zero N, Σt, Σm, Σt^2, Σm^2, and Σtm
while more metric point (t_i, m_i) **do**
  $N \leftarrow N + 1$
  Update Σt, Σm, Σt^2, Σm^2, and Σtm.
  if $N > 1$ **then**
    Calculate a and b using (1) and (2)
    Calculate $\delta(a,b)$ using (3)
    Calculate measure of error using (4)
    if measure of error > threshold ϵ **then**
      Output a', b', and t_i
      Initialize N, Σt, Σm, Σt^2, Σm^2, and Σtm
    end if
    $a' \leftarrow a$
    $b' \leftarrow b$
  end if
end while

Figure 3: Polyline Fitting Algorithm

threshold ϵ. We have chosen the Coefficient of Variation (CV), the ratio of the standard deviation to the mean, as the error metric. Here, it is the quantity

$$\frac{N\sqrt{\delta(a,b)}}{\sum_{i=1}^{N} m_i} \quad (4)$$

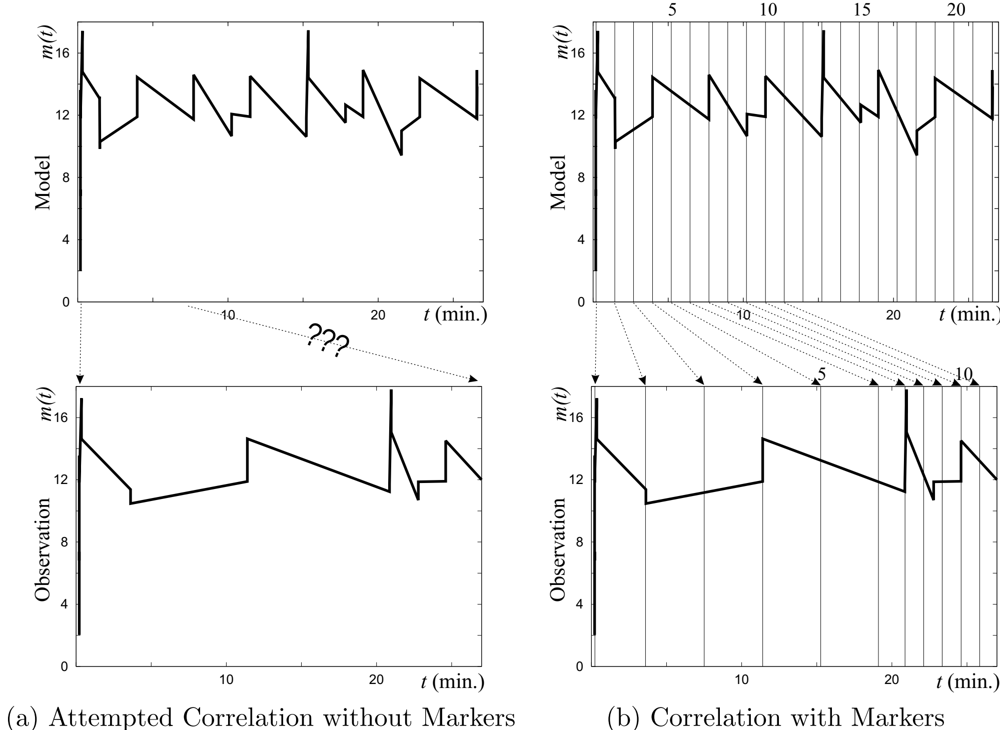
The output of this algorithm is a series of triples (a_i, b_i, t_i), each of which denotes a unique line segment $q_i(t) = a_i + b_i t$ where $t \in [t_{i-1}, t_i)$.

3.2 Markers

Multiple executions of the same code, whether on similar or different platforms, will result in different execution times and trajectories through the performance metric space. These trajectories can be either dilated or compressed, affecting the polylines fitted to them. Figure 4(a) illustrates the non-linear scaling effects possible due to execution on different platforms.

Despite non-linear compression or dialation, it is reasonable to view two polylines as similar even if one is a scaling of the other. Therefore, removing the effects of scaling is critical to accurate comparison. Equally importantly, characterizing the scaling provides insight into the effects of execution platforms on application behavior.

To quantify scaling effects, we track the application's intrinsic progress by inserting special instrumentation code at selected points (e.g., selected function or loop entries.) When the instrumentation is invoked, a special event, called a "marker," is generated. Each marker consists of the time relative to the application start time and a unique index number. Figure 4(b) illustrates the benefit of adding markers:



(a) Attempted Correlation without Markers (b) Correlation with Markers

Figure 4: Marker Use

the vertical lines denote occurrences of marker events and numbers atop are corresponding indices.

4 Signature Comparison

Given application signatures, our goal is creation of a simple mechanism for comparing signatures across executions, either on the same or different platforms. Below, we describe a template metric for real-time signature comparison.

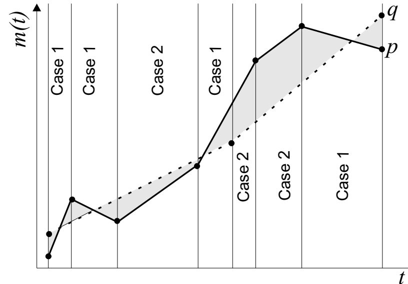


Figure 5: Polyline Similarity Comparison

4.1 Template Metrics

Mathematically, a similarity measure on the set G of geometric shapes is a distance function defined as $d : G \times G \to \mathbb{R}$. A shorter distance implies higher similarity. In this paper, we adopt the template metric [8], which is a special case ($n = 1$) of the L_n metric defined as

$$L_n(p, q) = \Big(\int |p(t) - q(t)|^n dt \Big)^{1/n}$$

for curve similarity comparison. As an example, Figure 5 illustrates two polylines p and q. The gray area denotes $L_1(p, q)$.

Direct computation of $L_1(p, q)$ is difficult. However, if one draws vertical lines through all endpoints of the line segments of p and q, the gray area partitioned by every pair of consecutive vertical lines can be calculated separately. Moreover, there are only two cases to consider.

In case one, two line segments intersect, and in case two, one segment lies atop the other. Both cases can be calculated efficiently from basic geometry.

Finally, we define the degree of similarity (DS) of q with respect to p as

$$\mathrm{DS}(p, q) = \max\Big(1 - \frac{L_1(p, q)}{\int p(t)dt}, 0\Big)$$

The DS metric ranges between 0 and 1, with larger values of DS implying higher similarity.

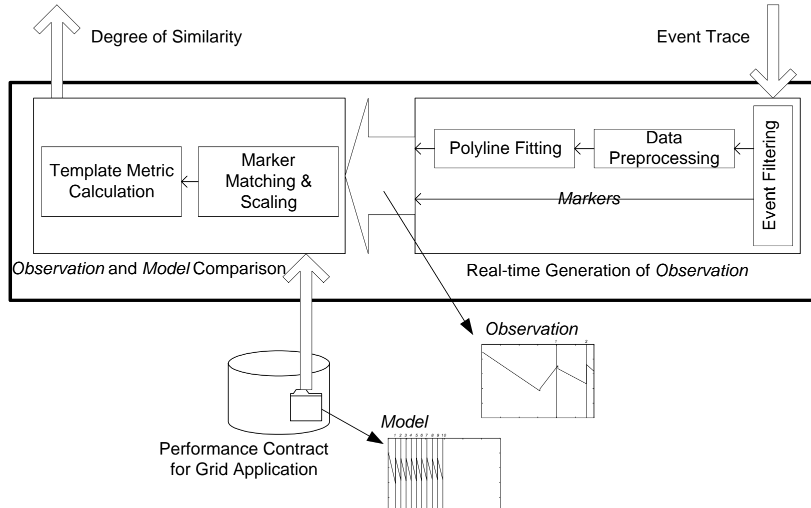


Figure 6: Application Signature Detail

4.2 Comparison Algorithm

To compute signature similarity, our comparison algorithm reads a baseline signature (i.e., the *model*) prior to application execution. During application execution, marker events trigger similarity evaluation.

Without loss of generality, assume marker events are indexed $1, 2, \ldots$. Also, define the portion of a polyline partitioned by consecutive markers as a *slice*. When a marker of index i is observed, the algorithm extracts from the model the slice partitioned by markers i and $i-1$. It then scales and aligns the observation slice with respect to the extracted model slice and computes the template metric. Figure 6 illustrates the complete process.

5 Experimental Assessment

To evaluate signature creation and comparison algorithms, we captured execution behavior on a diverse set of hardware and software configurations, including multiple Linux clusters, a loose collection of Sun workstations, and an IBM SP2; see Table 1.

We chose three different scientific codes, CSAR Rocket, Gadget, and Cactus, for signature evaluation. They are widely used for large-scale scientific calculations on distributed memory parallel systems and each has markedly different execution behavior.

After capturing traces from each code on each system configuration, we smoothed the trace data via moving window averages. We then normalized the smoothed data before input to the polyline fitting algorithm. This smoothing and normalization simplified comparison across platforms [7].

5.1 CSAR Rocket

The CSAR Rocket Code [12] is an integrated, whole-system solid propellant rocket simulation. It focuses on propellant combustion, turbulent flows, rocket case and nozzle structures under both normal and abnormal operations, component aging and other failure models. We chose to put marker generation code at entry of the outermost loop in source file `gen1.f90`.

Figure 7 shows the signatures for I/O write sizes from CSAR Rocket of node 0. With an error threshold ϵ value of 0.05, each signature is 0.8% of the size of the raw event trace.

Moreover, the signatures are very similar visually, save for non-linear time dilation due to hardware performance differences. Table 2 shows that using the generated markers to compare signatures across execution contexts yields quantitative similarity as well.

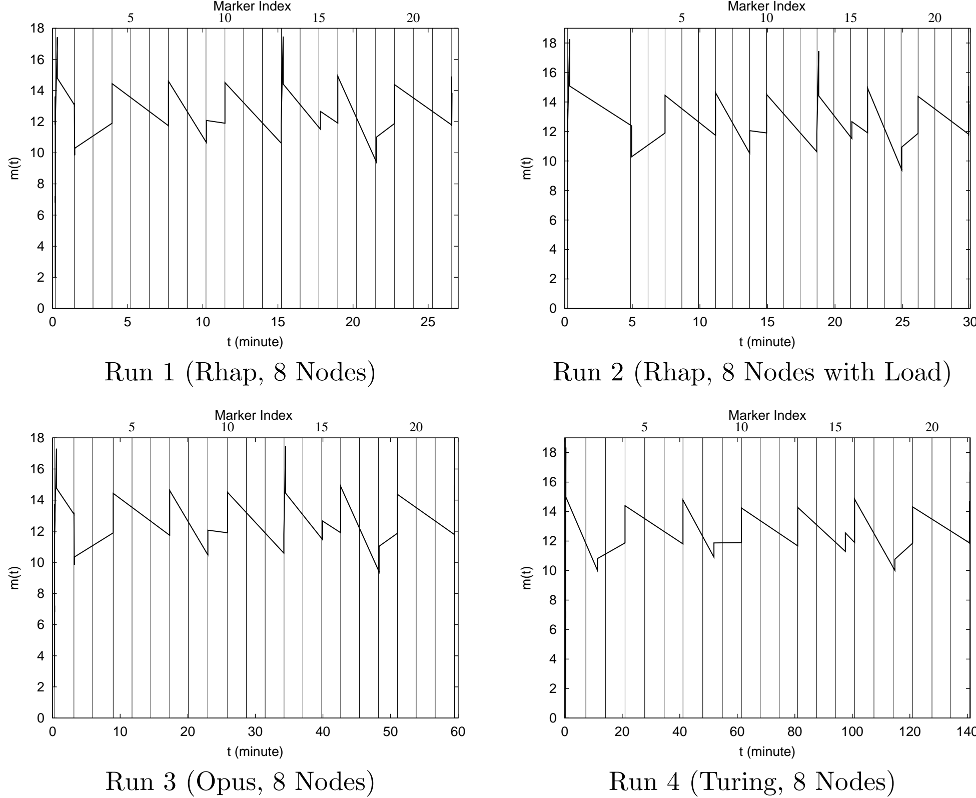
The ability to compare non-linear scalings of event traces via compact signatures enables assessment of the interplay of computation, communication, and I/O performance on execution dynamics. For example, by bounding the range of acceptable scalings across signatures, one can identify when execution behavior lies outside acceptable ranges – this is precisely the motivation for performance contracts.

5.2 Gadget

Gadget [13] is a cosmological simulation of interacting galaxies that evolves self-gravitating collisionless fluids and gas. We used a standard 277,000 particle

Name	Configuration	Processor	Network
Rhap	Linux Cluster	933 MHz Pentium III	Fast Ethernet
Opus	Linux Cluster	450 MHz Pentium II	Myrinet
Turing	Linux Cluster	550,1000 MHz Pentium III	Myrinet
Sitar	Sun Workstations	450 MHz UltraSPARC II	Ethernet
IBM	IBM RS/6000 SP2	375 MHz POWER3	IBM Switch

Table 1: Experimental Configurations



Run 1 (Rhap, 8 Nodes)
Run 2 (Rhap, 8 Nodes with Load)
Run 3 (Opus, 8 Nodes)
Run 4 (Turing, 8 Nodes)

Figure 7: CSAR Rocket Signatures (I/O Write)

	Model			
Run	1	2	3	4
1		0.99	0.99	0.99
2	0.99		0.99	0.98
3	0.99	0.99		0.98
4	0.98	0.98	0.98	

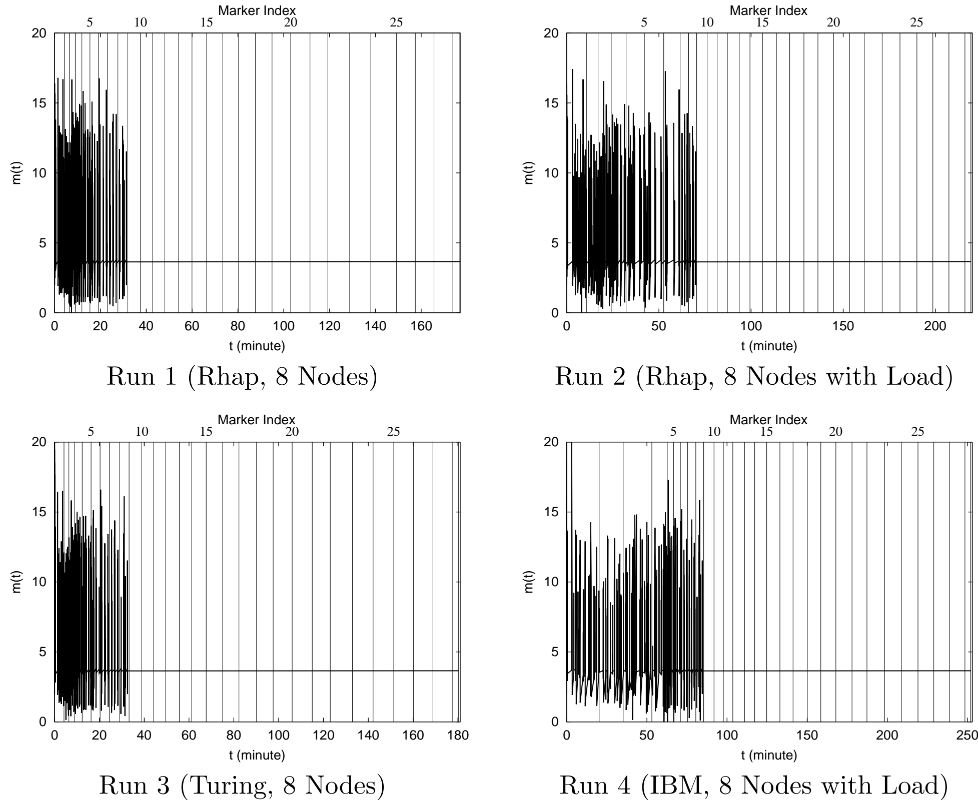
Table 2: CSAR Rocket Signature Similarity

data file as input to the code. We inserted marker generation code inside the main loop of `run()` function, emitting a marker every 1000 iterations.

Gadget's I/O activities only occur near the beginning, around one-third, and end of its three hour execution. However, its message passing events occur roughly every four milliseconds, producing a 10-40 MB trace on each node. To assess the efficacy of trace compression via signatures for traces with frequent events, we examined the trace of message sizes collected on node zero.

Figure 8 shows Gadget's signatures for various execution contexts. Gadget's signatures for message size show great variation during the initial portion of execution, though most of the metric values later fluctuate around a stationary mean.

A typical execution of Gadget produces roughly three million message passing events. With an ϵ value of 0.25, the signature is 0.02% of the size of the raw event trace. Moreover, as Table 3 shows, the com-



Run 1 (Rhap, 8 Nodes) Run 2 (Rhap, 8 Nodes with Load)

Run 3 (Turing, 8 Nodes) Run 4 (IBM, 8 Nodes with Load)

Figure 8: Gadget Signatures (Message Size)

	Model			
Run	1	2	3	4
1		0.94	0.96	0.85
2	0.97		0.95	0.88
3	0.96	0.91		0.87
4	0.94	0.90	0.95	

Table 3: Gadget Signature Similarity

	Model					
Run	1	2	3	4	5	6
1		0.94	0.54	0.99	0.99	0.98
2	0.94		0.56	0.94	0.94	0.95
3	0.16	0.20		0.16	0.16	0.21
4	0.99	0.94	0.54		0.99	0.98
5	0.99	0.94	0.54	0.99		0.98
6	0.98	0.94	0.56	0.98	0.98	

Table 4: Cactus Wavetoy Signature Similarity

parison algorithm captures the detailed execution dynamics.

5.3 Cactus Wavetoy

Cactus [14] is a toolkit originally developed for solution of numerical relativity problems. We chose the Wavetoy test program from the Cactus software distribution. Wavetoy simulates the evolution of an 3-D scalar field by solving a hyperbolic partial differential equation using finite differences. We inserted marker generation code inside the `IOBasic_OutputInfoGH` function of the main loop.

Cactus Wavetoy always transmits messages of constant size (161,600 bytes); therefore, we focus on its I/O activities (of node zero.) Using $\epsilon = 0.1$, the signatures reduce 98% amount of trace data.

To see how well the signature captures the behavioral differences, we tuned an input parameter that intensifies large writes of Cactus Wavetoy in Run 3. And as Figure 9 and Table 4 show, Run 3's curve is smoothed by these large writes and bears little resemblance to others.

5.4 Parallelism

So far, our experiments have only characterized the behavior of a single node on a parallel system. Our

approach can be easily extended to all nodes by creating a signature for each. By comparing signatures across nodes, we found that behavior of the nodes can be usually divided into several equivalence classes, as in [7]. Hence, only one signature is required for each equivalence class, saving much storage space and computation overhead.

For example, the CSAR Rocket's node 0 is the only one that performs disk writes. Similarly, all nodes of the Cactus Wavetoy code have homogeneous disk write patterns (99% similar). The case for Gadget is more interesting; on the average, the node signatuers are only 76-79% similar to each other. This irregularity is due to Gadget's dynamic tree structures and algorithms.

5.5 Discussion

In our signature scheme, one must create corresponding signatures for different input parameter sets. This potentially results in a great number of { application, input, signature } tuples. Alternatively, if one can parametrize the signature curve as a function of inputs, one needs only one signature for each performance metric. This is the key problem in the system identification research [17]. In our case, a scientific code is viewed as a nonlinear black-box system with *dozens* of input parameters. Automatically identifying those parameters that have impact on performance of interest is difficult, let alone quantifying the effects. Although artificial neural networks or fuzzy logic may help solve the problem, they have limitations. For the former, the training process could be time-consuming, and for the latter, prior knowledge about the program is required to write fuzzy rules [18].

Another issue is the choice of performance metrics. If the metric is runtime environment dependent, then the signatures will not be similar [16]. Examples are time-dependent metrics like "I/O operations *per second*." Timing variability causes inconsistency in resultant curves. Therefore, the area under curves might be similar, but the shapes of curves will not necessarily be so.

6 Related Work

Our curve fitting signature design was inspired by phase behavioral analysis of parallel codes [3, 4, 5, 6]. Phase behavioral analysis uses geometric curves to characterize performance as a function of time. Unlike our online signature approach, these automatic phase characterization algorithms must scan the entire trace several times. Prophesy [15] is the most similar work. It also uses curve fitting to model performance, but the form of the curve must be known in advance (e.g., cubic polynomials), whereas our polyline approach can fit any data set of unknown distribution.

For curve comparison, the most studied and applied distance function is the Hausdorff metric [9]. However, it is very sensitive to noise; a single outlier can skew the metric [8]. Although variants such as the ranked Hausdorff metric can overcome this limitation, they are only applicable for finite point sets instead of curves. Conversely, our template metric is resistant to outliers and easy to calculate.

7 Conclusions and Futures

We proposed a curve fitting approach for computing compact *application signatures* that captures the salient execution dynamics of scientific codes from event trace data. We also introduced algorithms to compare observed performance across signatures.

Our experimental results also suggest that application signatures can characterize application performance across diverse execution contexts, making them suitable for use with contracts for validating execution performance.

Building on this work, the next challenge is integration of application signatures with the performance contract mechanism of Figure 2. This will enable real-time tracking of the deviation of observed performance from expected behavior.

References

[1] Foster, I. and Kesselman, C. Eds. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 1998

[2] Berman, F., et al. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Computing Applications*, Winter 2001, Vol. 15, No. 4, pp. 327–344.

[3] Miller, B. P., Clark, M., Hollingsworth, J., Kierstead, S., Lim, S.-S., and Torzewski, T. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Trans. Parallel and Distributed Systems* , Vol. 1, No. 2, 1990, pp. 206–217

[4] Carlson, B. M. and Wagner, T. D. An Algorithm for Off-Line Detection of Phases in Execution Profiles. *Computer Performance Eval-*

uation, Modeling Techniques and Tools, 7th International Conference, G. Haring and G. Kotsis, Eds. Lecture Notes in Computer Science No. 794, Springer-Verlag, 1994, pp. 253–265.

[5] Carlson, B. M., Wagner, T. D., Dowdy, L. W., and Worley, P. H. Speedup Properties of Phases in the Execution Profile of Distributed Parallel Programs. *Computer Performance Evaluation 92: Modelling Techniques and Tools* , R. Pooley and J. Hillston, Eds. Edinburgh Press, 1992, pp. 83–95.

[6] Worley, P. H. Modeling Histogram Data with Piecewise Polynomials. Technical Report ORNL/TM-11637, Oak Ridge National Laboratory, 1990.

[7] Reed, D. A., Nickolayev, O. Y., and Roth, P. C. Real-Time Statistical Clustering for Event Trace Reduction, *Journal of Supercomputing Applications and High-Performance Computing*, Summer 1997, Vol. 11, No. 2, pp. 144-159.

[8] Veltkamp, R. C. and Hagedoorn, M. Shape Similarity Measures, Properties, and Constructions. Lecture Notes in Computer Science No. 1929, Springer-Verlag, 2000

[9] Huttenlocher, D. P., Klanderman, G. A., and Rucklidge, W. J. Comparing Images Using the Hausdorff Distance. *IEEE Trans. Pattern Analysis and Machine Intelligence* , Vol. 15, No. 9, 1993, pp. 850–863

[10] Gerald, C. F. and Wheatley, P. O. Applied Numerical Analysis. Third Edition, Addison Wesley, 1984.

[11] Vraalsen, F., Aydt, R., Mendes, C., and Reed, D. A. Performance Contracts: Predicting and Monitoring Grid Application Behavior. *Proc. 2nd International Workshop on Grid Computing*, 2001

[12] Heath, M. T., Fiedler R. A., and Dick W. A. Simulating Solid Propellant Rockets at CSAR. AIAA 2000-3455, *36th AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, 2000

[13] Springel, V., Yoshida, N., and White, S. D. M. GADGET: A Code for Collisionless and Gasdynamical Cosmological Simulations. *New Astronomy*, Vol. 6, 2001, pp. 79–117

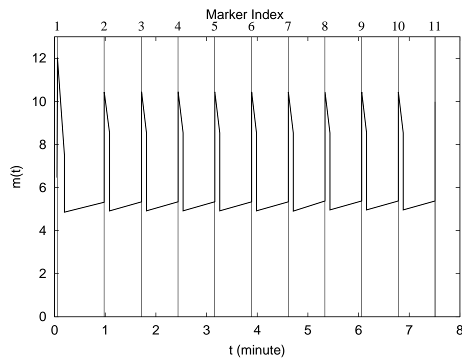
[14] Allen, G., et al. The Cactus Code: A Problem Solving Environment for the Grid. *Proc. High Performance Distributed Computing 2000* pp. 253–260

[15] Taylor, V., et al. Prophesy: Automating the Modeling Process. *3rd Annual International Workshop on Active Middleware Services*, in conjunction with HPDC-10, 2001

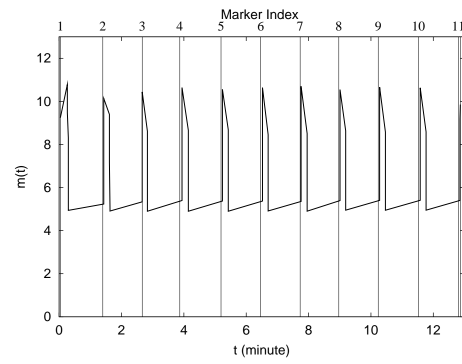
[16] Lu, C.-D. Application Signatures for Scientific Codes. Master Thesis, Univ. of Illinois at Urbana-Champaign, 2002.

[17] Ljung, L. System Identification - Theory For the User. Prentice Hall, 1987

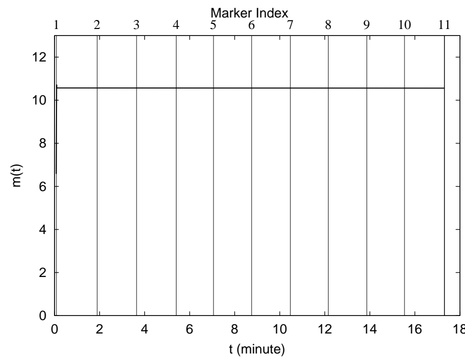
[18] Sjöberg, J., et al. Nonlinear Black-Box Modeling in System Identification: a Unified Overview. *Automatica*, Vol. 31, No. 12, pp. 1691–1724



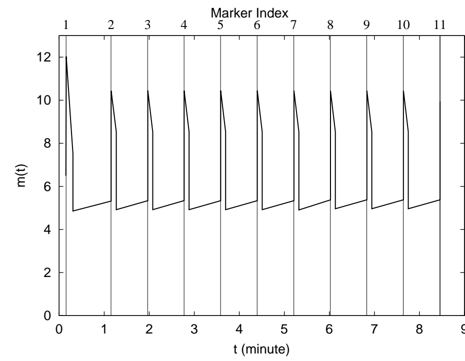
Run 1 (Turing, 4 Nodes)



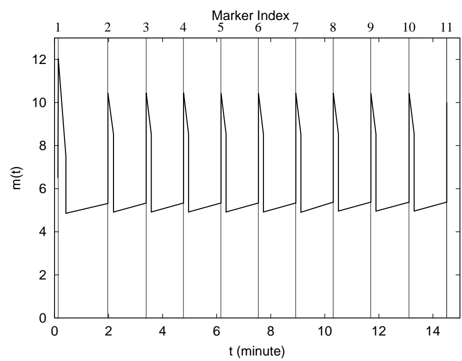
Run 2 (Turing, 16 Nodes)



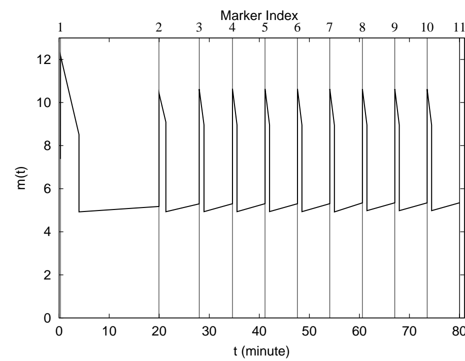
Run 3 (Turing, 4 Nodes with Changed Inputs)



Run 4 (Rhap, 16 Nodes)



Run 5 (Opus, 4 Nodes)



Run 6 (Sitar, 4 Nodes)

Figure 9: Cactus Wavetoy Signatures (I/O Write)