Automating Vertical Profiling

Matthias Hauswirth
University of Colorado at Boulder
Matthias.Hauswirth@colorado.edu

Amer Diwan[‡]
University of Colorado at Boulder
diwan@cs.colorado.edu

Peter F. Sweeney[§]
IBM Thomas J. Watson Research Center
pfs@us.ibm.com

Michael C. Mozer
University of Colorado at Boulder
mozer@cs.colorado.edu

ABSTRACT

Last year at OOPSLA we presented a methodology, *vertical profiling*, for understanding the performance of object-oriented programs. The key insight behind this methodology is that modern programs run on top of many layers (virtual machine, middleware, etc) and thus we need to collect and combine information from all layers in order to understand system performance. Although our methodology was able to explain previously unexplained performance phenomena, it was extremely labor intensive. In this paper we describe and evaluate techniques for automating two significant activities of vertical profiling: trace alignment and correlation. Trace alignment aligns traces obtained from separate runs so that one can reason across the traces. We are not aware of any prior approach that effectively and automatically aligns traces. Correlation sifts through hundreds of metrics to find ones that have a bearing on a performance anomaly of interest. In prior work we found that statistical correlation was only sometimes effective. We have identified highly-effective approaches for both activities.

For aligning traces we explore dynamic time warping, and for correlation we explore eight correlators: Pearson's coefficient, Spearman's coefficient, Manhattan distance, Euclidean distance, dynamic time warping, manual linear pattern, best splits, and same splits. Although we explore these activities in the context of vertical profiling, both activities are widely applicable in the performance analysis area.

[‡]This work is supported by NSF ITR grant CCR-0085792, NSF Career CCR-0133457, and an IBM faculty partnership award. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

[§]This work is supported by Defense Advanced Research Project Agency Contract NBCH30390004.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Computer Systems Organization**]: Performance of Systems—*measurement techniques, performance attributes*

General Terms

measurement, performance, experimentation

Keywords

vertical profiling, whole-system analysis, perturbation, hardware performance monitors, software performance monitors

1. INTRODUCTION

Although object-oriented technologies have many software engineering benefits, they have a significant cost in performance. Nonetheless, object-oriented technologies are now widely accepted, partly because rapid increases in processor performance – as predicted by Moore's law – have offset their performance costs. However, for various reasons including power dissipation and wire delays, many computer architects believe that Moore's law, which predicts that computing power will double every 18 months, will cease to hold sometime in the near future. Thus, to get the needed performance for new object-oriented technologies, such as web services, we need to look at alternatives to rapid increases in hardware performance. One possibility is to use performance analysis and tuning techniques to better exploit the hardware. This approach has great potential because most applications realize only a fraction of the full potential from modern hardware.

Last year at OOPSLA, we presented a methodology, *vertical profiling*, to analyzing the performance of object-oriented applications [10]. This analysis is a prerequisite for performance tuning. Vertical profiling works by collecting performance and behavioral data from all components of the system (e.g., application, Java virtual machine, and hardware) and then uses statistical and visualization techniques to understand how the components interact in order to effect overall system performance. We applied the vertical profiling methodology to five case studies. In each case study we used the methodology to explain *performance anomalies* in one or more Java application. We define a *performance anomaly* as any aspect of an application's performance that is unusual, unexpected, or undesirable. We had been unable to explain these anomalies using conventional profiling tools and techniques[1]. Although our evaluation showed that vertical profiling was effective, it was extremely labor intensive. Each case study took us weeks of investigation to complete. With the increasing complexity of both software and hardware, the difficulty of understanding the performance of applications will only grow over time. To enable performance analysis and tuning to scale to newer systems we must automate much of vertical profiling. This paper shows how to automate significant activities of vertical profiling.

There are four main activities in vertical profiling:

- *Identification of anomalies.* In this activity the performance analyst picks an anomaly to explore in a target metric. For example, the performance analyst may wish to explore why the instructions-per-cycle (target metric) is so low during some periods in program execution.

- *Trace alignment.* The performance metrics are spread out over multiple trace files, with each file having only some metrics. Although all trace files come from executions using the same application inputs, each execution collects a different set of metrics. That along with nondeterminism in the underlying system means that the same event in different trace files may not line up perfectly (i.e., may happen at different times relative to the program start time). Before we can meaningfully analyze the relationship between different metrics, we must align the traces in which they occur.

- *Correlation.* For each of our modestly-sized case studies, we had over 300 performance metrics from which we needed to identify ones that had a bearing on the performance anomaly of interest. This task is complex because the relevant metrics may have vastly differing frequencies than the target metric, may be involved in nonlinear relationships with the target metric, or there may be significant noise in the data. In our past work we used statistical correlation techniques to automate this step; however, those techniques did not always work due to the complexities described above.

- *Determination of causality*: In this activity the performance analyst applies domain knowledge to the correlated metrics to determine the chain of causality that explains the performance anomaly.

Of the above activities, the second and third are the most labor intensive, requiring us to manually align and correlate hundreds of metrics per benchmark spread out over tens of traces. The fourth activity, determination of causality, is also a hard problem but its difficulty has more to do with the needed domain knowledge rather than intense manual labor. For the first activity, providing a GUI for browsing and selecting metrics was enough for our needs: performance analysts could efficiently and quickly spot anomalies that they wanted to explore. Thus, in this paper we focus on automating the "trace alignment" and "correlation" activities.

We describe and evaluate one technique from the speech recognition literature for automatic trace alignment: *dynamic time warping* (DTW). To evaluate automated trace alignment, we use a graphical user interface to visually check if the traces are aligned correctly. We find that for our needs dynamic time warping works nearly perfectly. However, we do identify two scenarios (which we exercised using synthetic traces) where DTW fails; these insights are invaluable to performance analysts who wish to use this trace alignment technique.

[1]Our earlier attempt to explain these anomalies using only hardware performance monitor information failed [21].

We describe and evaluate eight correlators. Four of the techniques are from the statistics literature: Pearson correlation coefficient, Spearman correlation coefficient, Euclidean distance, and Manhattan distance. Three of the techniques are new uses or variations of techniques in prior work from the speech recognition and data mining literatures: dynamic time warping, same splits, and best splits. Finally, we describe a novel technique: manual linear pattern. We evaluate these techniques based on how effective they are at identifying the metrics that the performance analyst should look at. Of the techniques described in prior work, no obvious technique is the winner: some work better for correlating continuous signals although others are better for discontinuous signals. Our novel technique, manual linear pattern, which is based on Pearson's correlation coefficient, performs the best or very close to the best consistently.

The remainder of the paper is organized as follows. Section 3 presents how we use dynamic time warping to perform trace alignment. Section 4 presents our evaluation of trace alignment. Section 5 presents the correlators. Section 6 presents our evaluation of the correlators. Section 7 reviews related work. Section 8 presents our conclusions.

2. BACKGROUND

In our previous work [10], we analyzed five performance anomalies in a set of nine benchmarks. The benchmarks consisted of the seven SPECjvm98 applications, SPECjbb2000, and hsql. In this paper, we use four of those performance anomalies as a basis for evaluating our correlators. The four performance anomalies are:

- **Gradual increase**
  - *Performance anomaly*: Over time, the instructions per cycles (IPC) gradually increases
  - *Benchmarks*: mpegaudio, mtrt, jbb, and hsql
  - *Explanation*: Over time more and more of the code gets optimized. Optimized methods use registers instead of an evaluation stack and thus suffer fewer misspeculations (i.e., flushes in the load/-store unit) resulting in better performance.
- **Sudden increase**
  - *Performance anomaly*: Sudden increase in IPC
  - *Benchmark*: compress
  - *Explanation*: Similar to gradual increase except that compress spends most of its time in two methods (compress and decompress) and there is a big jump in performance once these methods are optimized.
- **Periodic**
  - *Performance anomaly*: repeated periodic sine pattern in IPC

  - *Benchmark*: db
  - *Explanation*: db executes a number of shell sorts over its database. At the beginning of the sort, the working set fits in the L2 cache and the algorithm touches each entry more and more times (thus resulting in better temporal locality) as the sort progresses. When the working set ceases to fit in the L2 cache the performance degrades.

- **Pre-GC dip**
  - *Performance anomaly*: IPC suddenly dips before a garbage collection (GC)
  - *Benchmarks*: jbb, hsql
  - *Explanation*: The GC expands and shrinks the heap as needed (using mmap and munmap)[2] . An application uses the existing heap before growing into the expansion area. The first accesses to the expansion area cause a series of page misses that degrades performance. Because the expansion area is used just before the program runs out of memory and triggers a GC, the dips happen right before a GC.

In our previous work, while determining the causes for performance anomalies, we were surprised to find that we needed to look at a diverse range of metrics, some of which we would never have thought to look at. For example, in the pre-GC dip performance anomaly, a key metric was exceptions disabled, an operating system metric that provided insight into page fault behavior; in the gradual and sudden increase performance anomalies, a key metric was the number of flushes in the load store unit, a functional unit of the microarchitecture; and in the periodic performance anomaly, a key metric was the set size of the shell sort, an application metric. This diversity of the key metrics underscores the importance of exploring every metric in every component when trying to understand a performance anomaly. Furthermore, our previous experience demonstrates that while visual correlation identifies the key metrics it is extremely labor intensive since we need to examine hundreds of metrics. Thus, it is worthwhile to automate correlation.

3. TRACE ALIGNMENT

This section discusses trace alignment, an essential step in automating vertical profiling. We first describe the nature of the problem that trace alignment solves, then describe our approach for aligning traces, and finally discuss how we use trace alignment to reason about metrics collected in different traces.

3.1 Problem Description

For each of our case studies we collected hundreds of metrics using hardware and software performance monitors. Many application runs had to be made to collect these metrics. Consequently, the metrics are spread out over many trace files[3]. There are two reasons why we cannot collect all the metrics in a single run: (i) The software metrics require us to instrument code at all levels of the system. We found that for any single metric the instrumentation did not significantly perturb program execution. However, if we collect all the metrics in a single run it substantially perturbs the run. Thus, to minimize perturbation, we spread out the collection responsibility over a number of runs, with each run collecting only a small subset of the software metrics. (ii) While modern microprocessors can measure over a hundred metrics using their hardware performance monitors, in any given run they can collect only a handful of the possible metrics. This is because the number of hardware performance registers is smaller than the number of metrics that one can collect using these registers.[4] For example, the PowerPC POWER4 microprocessor can use its hardware performance monitors to measure over a hundred metrics but has only eight hardware performance registers. Thus, to collect all the hardware metrics, we need tens of runs.

Having the metrics spread out over multiple trace files is problematic for any performance analysis system because it must first align the traces before it can reason about them. For example, one trace may contain the L1CacheMisses metric and another may contain the StallCycles metric. To determine if fluctuations in StallCycles are caused by fluctuations in L1CacheMisses, we must align the traces to see if the two fluctuations in the two metrics line up appropriately.

Unfortunately, trace alignment is a non-trivial problem for two reasons. First, many aspects of modern systems are nondeterministic (e.g. scheduling in the operating system as affected by external interrupts). Thus, even two runs collecting the same metrics and using the same inputs generate unaligned traces. Second, each run measures different metrics which means that each run is perturbed slightly differently than other runs. Figure 1 illustrates the need for trace alignment. Each signal is from a different trace of an execution of the *db* benchmark with the same set of inputs. The signal shows the IPC (instructions-per-cycle) metric. (The highlighted regions will be discussed in Section 3.3.) Each trace collects a different set of metrics, along with the IPC. The blank intervals in the signals represent garbage collections (we filtered out the garbage collector events from these signals). The traces clearly need alignment; for example garbage collections occur at different times (relative to the start of execution) for the different runs.

To better understand the alignment issues, Table 1 identifies four situations that an alignment technique must deal with. The "Conceptual example" column shows pairs of traces that suffer from the situation in Column "Situation". The "Real Example" column gives an example (excluding the fourth, which is discussed below) of the situation from our case studies.

The four situations in Table 1 are as follows. First, one of the runs may appear to be slower than the other run (*scaled*). Second, one of the runs may be *shifted* with respect to the other run. Third, one of the runs may appear *warped* with respect to the other. "Scaled" is really a special case of "warped" where the warping factor is constant throughout the run. Fourth, the events (e.g., garbage collection and compilation) in one run may be *reordered* with respect to the other run.

The situations in Table 1 are caused by factors such as variations in

[2]We never saw the GC shrink the heap in any of our benchmark executions.

[3]In our experiments, we needed to have 50 or so trace files per application.

[4]To the best of our knowledge, this is true for every modern microprocessor.

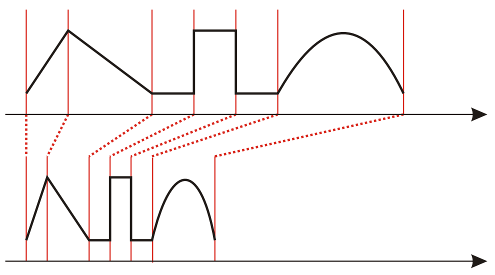
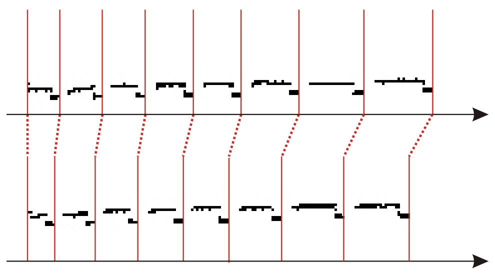
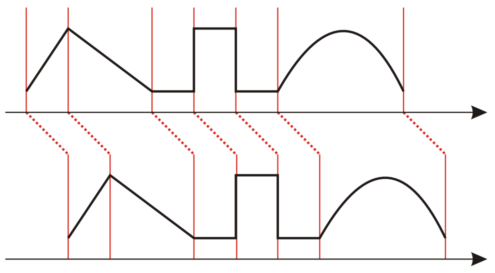
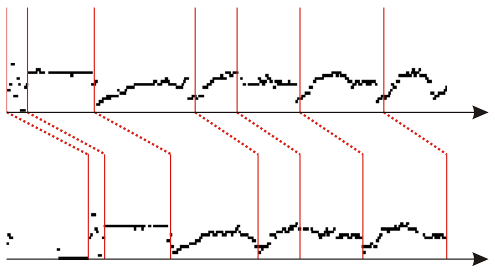
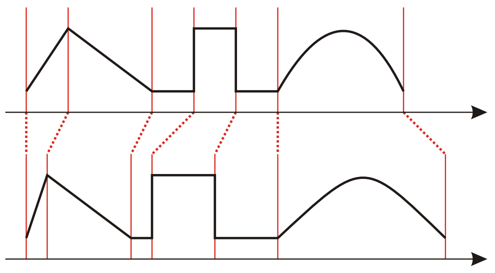
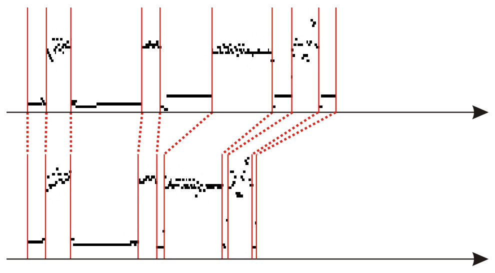
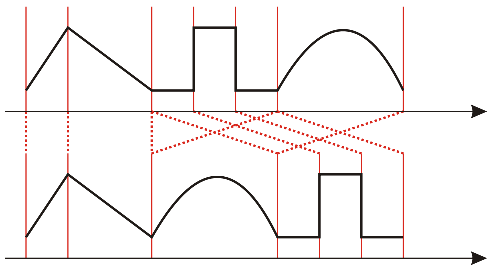
Situation	Conceptual Example	Real Example
Scaled		
Shifted		
Warped		
Reordered		*unlikely for a deterministic thread*

Table 1: Situations when Aligning Traces

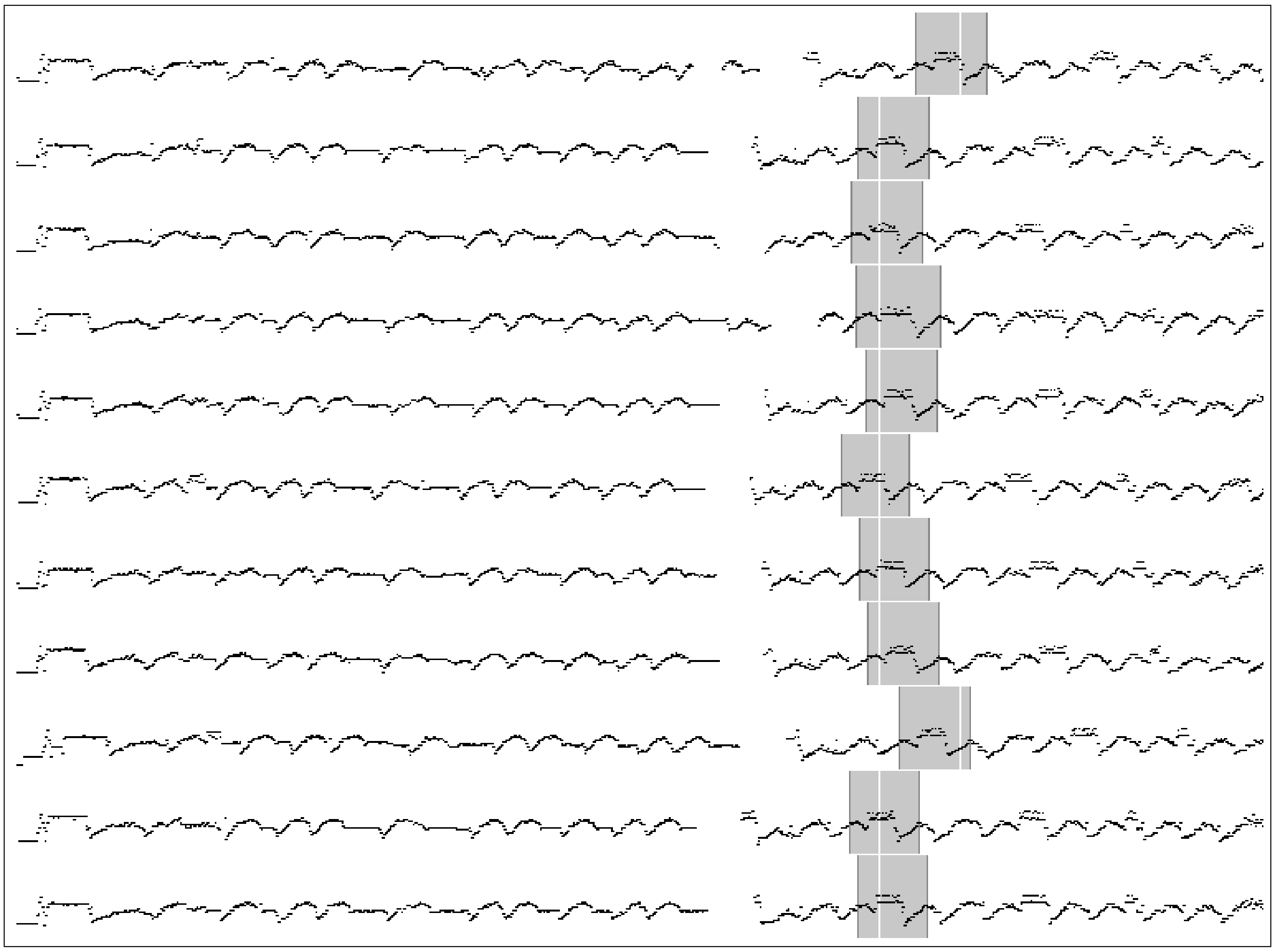


Figure 1: Alignment of Selection Across the Prefixes of Different Traces

- operating system activity;
- workload on computer (e.g., other running processes);
- instrumentation, because the two traces are instrumented to collect different sets of metrics;
- virtual machine behavior (e.g., the order, time, and optimization level that different methods get optimized);
- hardware performance (e.g., machine enters a different thermal mode when it becomes hot).

In Table 1, we do not have a real example for the fourth reordered situation, because we did not encounter it in practice. For events to be reordered, there has to be nondeterminism. However, we are primarily interested in only the parts of a metric that apply to a particular Java thread, and for the most part, these threads are deterministic.

3.2 Our Approach

We use a technique from the speech recognition literature, dynamic time warping (DTW) [4], to align the traces. Given two sequences, X and Y, of lengths $|X|$ and $|Y|$,

$$X = (x_1, x_2, \ldots, x_i, \ldots, x_{|X|})$$
$$Y = (y_1, y_2, \ldots, y_j, \ldots, y_{|Y|})$$

dynamic time warping constructs a warp path W (w_1, w_2, $\cdots$, $w_{|W|}$) such that each w_k is a pair (i, j), where i is an index into X and j is an index into Y. The warp path satisfies the following constraints:

1. For every element x_i of X, there is at least one $w_{i,*}$ and for every element y_j of Y, there is at least one $w_{*,j}$, i.e., no element of X or Y is omitted
2. $w_1 = (1, 1)$ and $w_{|W|} = (|X|, |Y|)$, i.e., the end points of the two sequences are aligned;
3. $w_k \lhd w_{k+1}$, where $(i_1, j_1) \lhd (i_2, j_2)$ if ($i_1 = i_2$ or $i_1 + 1 = i_2$) and ($j_1 = j_2$ or $j_1 + 1 = j_2$) and $(i_1, j_1) \neq (i_2, j_2)$, i.e., the warp path respects the order of both sequences.

DTW uses a dynamic programming algorithm to construct the warp path. The algorithm minimizes the *DTWError*:

$$\text{DTWError} = \sum_{k=1}^{|W|} |x_i - y_j| \text{ where } w_k = (i,j) \qquad (1)$$

Figure 2 shows an example of a warp path. The samples of the X sequence are the columns of the matrix and the samples of the Y sequence are the rows. The shaded squares represent the warp path. More specifically, if cell (i, j) is shaded it means that there is a k for which $w_k = (i, j)$. Figure 3 shows the alignment implied by the warp path in Figure 2. If (i, j) is on the warp path, then x_i is aligned to y_j. The lines that go between the two sequences in Figure 3 represent the alignment. If the two sequences are identical (and thus do not need any alignment), all these lines would be vertical. Note that there are situations where a single element of X aligns with multiple elements of Y and also where a single element of Y aligns with multiple elements of X. In Figure 2, such many-to-one situations appear as vertical or horizontal sequence of gray squares. For example in Figure 3, the four circled X events are mapped to one circled Y event and this is represented in warped path as a horizontal sequence of four gray squares.

In addition to the warp path, the DTW also produces another useful piece of information: the *DTWError* (Equation 1). This metric effectively gives the degree of similarity between the two traces being aligned and thus can also be used as a correlator (Section 5). As a matter of fact, prior work has used DTW primarily to compute the *DTWError*; using DTW to align traces is a new use of DTW introduced in this paper.

Finally, the DTW algorithm addresses the first three issues in Table 1; however, it cannot handle "reordered" because of the constraint that $w_k \lhd w_{k+1}$. Fortunately, we have not encountered this situation in practice.

3.3 Discussion

Trace alignment enables us to reason across metrics in different traces. If we want to determine the correlation of two metrics in the same trace, we can apply the correlation mechanism (Section 5) right away. However, if we want to determine the correlation of two metrics in different traces, we first align the two traces and then we use the correlation mechanism. To align two traces, we require that the two traces have at least one metric in common (*common metric*). We use the common metric to effect the alignment and then use the correlation mechanism on the aligned traces. In our experiments, all traces have IPC (instructions-per-cycle) as the common metric.

Figure 1 shows an example of DTW in practice. Each signal represents the IPC (instructions-per-cycle) metric collected in a separate traces of the *db* benchmark. Each trace collects a different set of metrics. The shaded regions represent how an interval in the first metric is aligned with intervals in the other metrics. We clearly see that the traces need alignment; for example the shaded regions occur at different times (relative to the start of execution) for the different runs.

4. EVALUATION OF TRACE ALIGNMENT

We now describe the methodology for evaluating DTW for trace alignment (Section 4.1) and evaluate DTW based on our needs (Section 4.2).

4.1 Methodology

To begin using our system, the performance analyst selects an area in a metric (corresponding to an anomaly) in a trace that she wants to explore further. We use trace alignment to select the corresponding intervals in all the other traces. We require that the analyst-selected metric occurs in all the traces; thus we use DTW to align different subsequences of the same metric.

We evaluate trace alignment in two ways: *global* and *local* correctness. "Global correctness" determines the extent to which DTW selects the same event in all the traces. "Local correctness" determines the extent to which DTW selects the same boundaries that a human would have picked. For example, if the trace analyst selects the second period (see "Periodic pattern" in Section 2) in the IPC sequence, it would be globally correct to select the second period in the IPC sequence in all the traces. Moreover, it would be locally correct if the boundaries (i.e., start and end points) of that period, as computed by DTW for all traces, match the boundaries we would have selected manually.

4.2 Evaluation

In our experience DTW was almost always globally and locally correct. The only exception we encountered was in the *hsql* benchmark when we were investigating the pre-GC dip (Section 2) (Section 4.2.2 describes this problem). Because DTW worked so well for our needs, we did not investigate alternate approaches for aligning traces. That said, using mostly synthetic traces, we were able to get DTW to perform poorly; the remainder of this section discusses these scenarios.

Both the scenarios, described in the following subsections, occur because DTW can map a single point in one trace to many points in the other traces. A consequence of this is that a single point (which may be a measurement artifact) can degrade the alignment if inappropriately situated. While these scenarios were not common in our experiments (we encountered only one instance of one of the scenarios) we still include them since they provide valuable guidance to others wishing to use DTW for trace alignment.

4.2.1 Local Correctness

Figure 4 illustrates a situation that causes DTW to be locally incorrect. The figure shows two sequences to be aligned. Broadly speaking, both sequences alternate between "high" and "low" values, staying at each value for several samples. However, both sequences have "noisy" samples, at height *h2* for the upper sequence and at height *h3* for the lower sequence. Intuitively, we would expect each sample in the upper trace to line up with the sample immediately below it on the lower trace. However, instead, DTW uses a many-to-one mapping since that is what minimizes the DTWError (Section 3.2, Equation 1). This results in a local inaccuracy since even though DTW selects the boundaries incorrectly the broad alignment is correct.

More generally, the above problem arises when trying to align two sequences that are mostly constant. If the values were constant in the two sequences, there would be no local errors. However, small deviations from the constant values lead to alignment errors.

While we can easily generate synthetic traces that exhibit this problem, we have not yet encountered it in practice. The reason for this is that so far we have focused on anom-

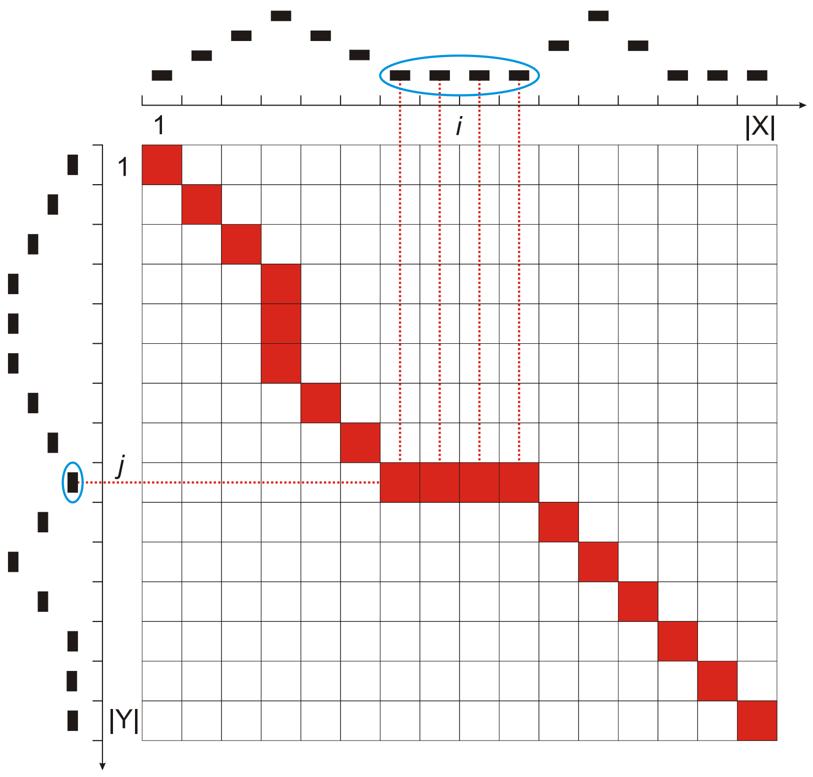


Figure 2: Dynamic Time Warping Warp Path

alies where behavior varies rather than remains constant.

One way to alleviate local incorrectness while trying to align mostly constant sequences is to augment DTWError so that it penalizes many-to-one and one-to-many associations. We will investigate this in future work.

4.2.2 *Global Correctness*

Figure 5 illustrates a situation that causes DTW to be globally incorrect. To improve readability of the figure, we show only some lines between samples. As with the previous example, both sequences alternate between low (*h1*) and high (*h3*) values. However, the upper sequence has a single noisy point at *h2*. Since *h2* is close to *h3*, DTW, aligns it with the *h3* points of the second period in the lower sequence. This is not what we would manually do: we would consider points at *h2* as if they were at *h1* (effectively disregarding deviations).

The consequence of this error is a global incorrectness: DTW fails to associate the second period of the upper sequence with the second period of the lower sequence.

We encountered the above problem once when analyzing the *hsql* benchmark. Fortunately, however, because the different pre-GC dips in *hsql* are nearly identical, it did not affect the results of subsequent analyses (particularly correlation).

The combination of constraints that DTW enforces (Section 3.2) collectively ensure that global (and even local) misalignments are uncommon. For example, consider two sequences that align well (i.e., DTW introduces no global or local alignment errors). Now let's suppose we add noise at one position in the first sequence. The only way that this noise can cause a global misalignment is if there is a later point in both sequences that allows the sequences to synchronize again (recall that DTW always synchronizes the end points of sequences and does not reorder points). If the sequences are repetetive, i.e., the same pattern repeats itself many times (e.g., hsql) then the repetetion of the sequences may provide the synchronization point. If the sequences are not repetetive, then a single noisy subsequence will most likely not affect global alignment: we will need appropriate "noise" later on in the single to enable the synchronization.

5. CORRELATORS

In our previous work [10], we used statistical correlation, specifically Pearson's coefficient, to identify metrics that were likely to be relevant to the performance anomaly of interest. Unfortunately, we found Pearson's coefficient lacking: for some of the case studies, it did not identify the key metrics as being correlated with the target metric. Consequently, we resorted to visual correlation. However, visual correlation is extremely labor intensive and error prone: the performance analyst must look at visualizations of hundreds of metrics to determine which ones look correlated. Thus, in this section we explore automated approaches to correlation. In Section 6, we evaluate how well these approaches work.

The correlators described here determine the strength of the relationship between two sequences of values. In the context of vertical profiling, these two sequences may be two metrics (which are sequences of values) or they may be subsequences of the metrics. The outcome of correlating two sequences is a number, the *correlation score*, which determines the extent to which the two metrics are related.

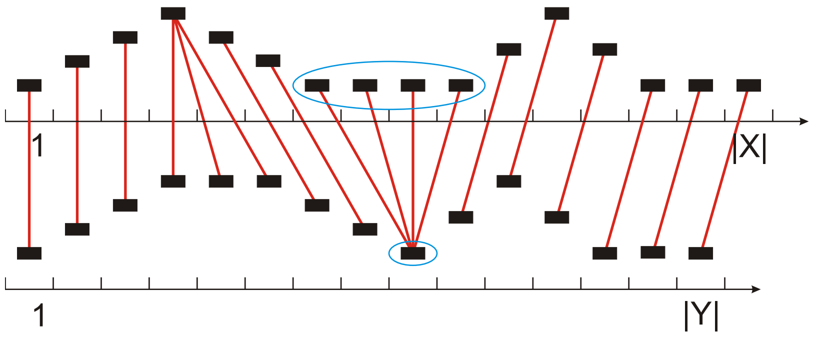


Figure 3: Time Warping Between Two Sequences X and Y

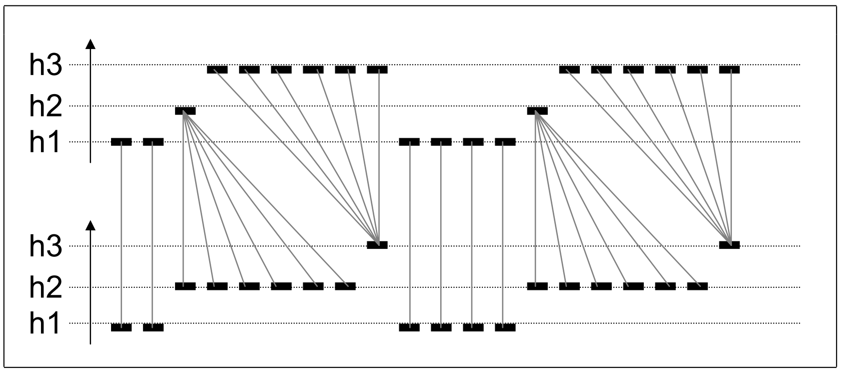


Figure 4: Local Incorrectness

When presenting these results to the user, we sort and rank the metrics in descending order using the correlation score; the highest ranked metric will have the highest correlation score and thus is most likely (according to the correlator) to have a bearing on the anomaly of interest.

For example, suppose we wish to explore an anomaly in the IPC (instructions-per-cycle) metric. A correlator may report that L1CacheMisses has a correlation score of 0.9 with IPC and the NumberOfSystemCalls has a correlation score of 0.75. Thus L1CacheMisses will be ranked higher than NumberOfSystemCalls, indicating that the performance analyst should look at L1CacheMisses first when investigating the anomaly.

The correlators consider both covariance and contravariance. A sequence *covaries* with another sequence if both vary in the same way (e.g., both increase and decrease at the same time). A sequence *contravaries* with another sequence if they vary inversely (e.g., one decreases when the other increases). Some of our correlators naturally consider both covariance and contravariance, while for the others, we correlate both a sequence and its negation to capture both covariance and contravariance.

Table 2 classifies our correlators along three dimensions. The *mechanism* dimension gives the underlying mechanism that the correlator uses. The *sensitivity* dimension indicates whether or not the correlator is time or order-sensitive. The *time-sensitive* correlators require not only the sequence of values but also the values' start and end times. The *order-sensitive* and *order-insensitive* correlators only require the sequence of values. The scores of order-sensitive correlators depend on the ordering of the values, i.e., they would not produce the same result if the order in the two sequences were changed. The scores of order-insensitive correlators do not depend on the order of values; it treats the sequences as multisets. Finally, the *computation* dimension specifies whether the correlator is fully automatic or if it requires some user input.

We now describe the different correlators in more detail.

5.1 Statistical Correlation

The correlators in this category come from the statistics literature and are widely used.

5.1.1 *Pearson's Product Moment Correlation Coefficient*

The Pearson's product moment coefficient, r, gives a measure of covariance or contravariance between two sequences. If the sequences are covariant, r lies between 0 and 1. If they are contravariant r lies between 0 and –1. Given two sequences X and Y of length N, Pearson's correlation coefficient is defined as:

$$r = \frac{\sum_i X_i Y_i - \frac{\sum_i X_i \sum_i Y_i}{N}}{\sqrt{\left(\sum_i X_i^2 - \frac{(\sum_i X_i)^2}{N}\right)\left(\sum_i Y_i^2 - \frac{(\sum_i Y_i)^2}{N}\right)}} \qquad (2)$$

To obtain the correlation score we use the absolute value of Pearson's coefficient, $|r|$, because both contravariance and covariance are equally indicative of a relationship between the two sequences.

The strength of Pearson's coefficient is that it is known to work well for clean, continuous signals. However, it does not perform well for noisy signals. In particular, outliers (called

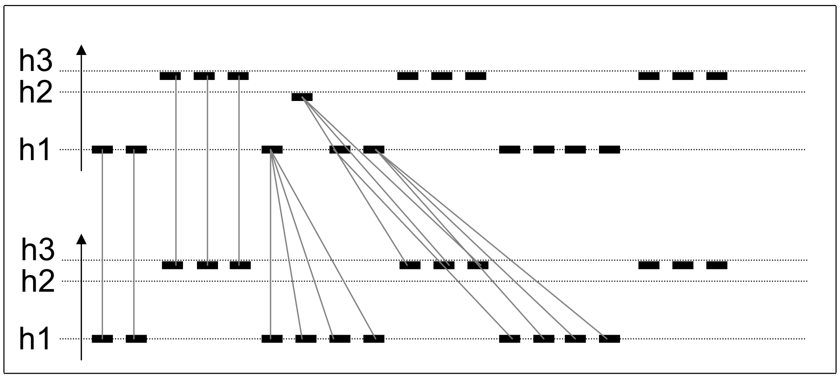


Figure 5: Global Incorrectness

Correlators	Mechanism	Sensitivity	Computation
Pearson correlation coefficient	statistical correlation	order-insensitive	automatic
Spearman correlation coefficient	statistical correlation	order-insensitive	automatic
Euclidean distance	distance	order-insensitive	automatic
Manhattan distance	distance	order-insensitive	automatic
Dynamic time warping distance	distance	order-sensitive	automatic
Same splits	segmentation	time-sensitive	semi-automatic
Best splits	segmentation	time-sensitive	semi-automatic
Manual linear pattern	segmentation	time-sensitive	semi-automatic

Table 2: Classification of Correlators

leverage points in the literature) can significantly perturb the correlation.

5.1.2 Spearman's Rank Correlation Coefficient

As noted above, Pearson's coefficient is overly sensitive to outliers. Spearman's rank coefficient attempts to improve on Pearson's coefficient in this respect by using the ranks of the values rather than the values themselves. The first step in computing Spearman's coefficient is to convert each sequence of values into a sequence of ranks. For example the sequence ⟨10,15,9⟩ becomes the rank sequence ⟨2,3,1⟩. Equation 3 shows the formula for computing Spearman's coefficient (r_s) from the rank sequences, $\ddot{X}$ and $\ddot{Y}$, derived from the value sequences X and Y, respectively.

$$r_s = 1 - \frac{6\sum_i \left(\ddot{X}_i - \ddot{Y}_i\right)^2}{N\left(N^2-1\right)} \tag{3}$$

As with the Pearson correlator, we use the absolute value of the Spearman's coefficient to get the correlation score.

A strength of Spearman's coefficient is that it can correlate sequences that covary or contravary but are not necessarily linearly related. This is because Spearman's coefficient does not directly use the values of the two sequences but instead uses their relative rank. For example, consider two sequences ⟨1,2,3,4,5,6⟩ and ⟨10^1,10^2,10^3,10^4,10^5,10^6⟩. These two sequences covary because when the first goes up the second also goes up. However, the variations in the two sequences are not linearly related. Thus Pearson's coefficient is 0.7 while Spearman's coefficient is 1.0 which more closely matches our intuition.

Another advantage of Spearman's coefficient over Pearson's coefficient is that, as mentioned above, using the ranks instead of the actual values enables Spearman's coefficient to gracefully handle some kinds of noise in the data. However, Spearman's coefficient also has its weaknesses. For example, consider the two sequences: ⟨1000, 1002, 1003⟩ and ⟨1, 100, 1000⟩. Because Spearman considers only the rank and not the absolute values, the two sequences end up looking the same (i.e., ⟨1,2,3⟩) even though the first may be a constant sequence with some noise while the second is an exponentially increasing sequence. Thus, a signal with constant stretches that are slightly noisy can significantly perturb the correlation.

5.2 Distance

The correlators in this section consider the distance between corresponding values of the sequences to determine covariance or contravariance. The first two correlators, which are special cases of the Minkowski distance, are well known and are order-insensitive. The third correlator, based on dynamic time warping, comes from speech recognition, and is order-sensitive.

Because the values of different metrics can have vastly different domains (metrics like IPC can have values around 1, while metrics like cache misses per cache reference can have values around 0.02), it would not be meaningful to compute the distance between the raw values of different metrics. Thus, before using any of the distance correlators, we first normalize all the values in a sequence such that each normalized sequence has a mean of 0 and its standard deviation is 1. We denote a normalized version of sequence X as $\tilde{X}$. Note that the other correlators do not need any such normalization.

A distance measure readily detects covariance between metrics, not contravariance. Consequently, we compute both the distance between the two sequences, Distance$(\tilde{X}, \tilde{Y})$, and the distance between the negation of one sequence and the other sequence, Distance$(-\tilde{X}, \tilde{Y})$. We use the inverse of the *smaller* of these two measures as the correlation score. We use the inverse because a smaller distance means higher correlation. If the smaller of the two distances is zero, the score is MAXINT.

5.2.1 Minkowski Distance

The Minkowski distance metric [3] evaluates differences between corresponding pairs of points in the two sequences $\tilde{X}$ and $\tilde{Y}$:

$$d = \left(\sum_i \left| \tilde{X}_i - \tilde{Y}_i \right|^M \right)^{\frac{1}{M}}, \qquad (4)$$

where the exponent M is a free parameter that specifies how much emphasis is to be placed on large differences, and the metric is often referred to as the L_M norm. A large M amplifies large differences: the L_∞ norm is a max operation that finds the largest difference among all pairs of the points, whereas the L_1 norm simply sums the absolute value of differences, treating small and large differences evenly.

We use two common special cases of the Minkowski metric, the L_1 or *Manhattan* distance, d_m:

$$d_m = \sum_i \left| \tilde{X}_i - \tilde{Y}_i \right| \qquad (5)$$

and the L_2 or *Euclidean* distance, d_e:

$$d_e = \sqrt{\sum_i \left(\tilde{X}_i - \tilde{Y}_i \right)^2}. \qquad (6)$$

The emphasis on larger differences makes the Euclidean distance more stable than the Manhattan distance in the presence of small amounts of noise. However, this greater emphasis also makes the Euclidean distance more susceptible than the Manhattan distance to perturbations from outliers.

5.2.2 Dynamic Time Warping (DTW)

The dynamic time warping (DTW) uses the *DTWError* (Equation 1 in Section 3.2) as the the distance between two sequences. Recall that the correlation score is the inverse of the distance.

DTW distinguishes itself from the other correlators in that it does not assume that

- An event in one metric *immediately* causes a corresponding event in the target metric. For example, compared to other automatic correlators DTW would consider the following two sequences to be highly correlated: ⟨0,100,0,0⟩ and ⟨0,0,100,0⟩.
- An event in one metric is of the same duration as the corresponding event in the target metric. For example, compared to the other automatic correlators, DTW would consider the following two sequences to be highly correlated: ⟨0,100,0,0⟩ and ⟨0,100,100,0⟩.

5.3 Segmentation Based

With the exception of Spearman's coefficient, all of the correlators presented so far can be perturbed significantly by noise in the data. Spearman's coefficient handles noise in the data by abstracting away from the raw data by using ranks instead. Another way of handling noise is to fit a polynomial to the data and use that polynomial as an approximation of the data. Noise is suppressed by using points on the polynomial curve rather than the raw data points. It is important to use a low-order polynomial otherwise we will end up with a polynomial that fits not only trends in the sequence, but also noise. In our work, we have focused on first-order polynomials, i.e., lines.

Given a sequence of points $y_t = \langle y_1 \ldots y_T \rangle$, where the subscript is a time index, linear regression can be used to determine the best-fit straight line, $\hat{y} = at + b$. This line, referred to as the regression line, has a slope a and a y-intercept b. Given a and b, $\hat{y}_t$ is the approximation to y_t. Linear regression finds the values of a and b that minimize $\sum_t (y_t - \hat{y}_t)^2$.

Given the complexity of our data, a single line will generally fail to accurately characterize the metric. Thus, rather than using linear regression directly as a correlator, we use it as a component of *piecewise linear segmentation*. A piecewise linear segmentation of a sequence is defined as a sequence of line segments, each segment defined by a 4-tuple $\langle s_j, e_j, a_j, b_j \rangle$, where s_j and e_j are the start and end time while a_j is the slope and b_j is the intercept of the jth line segment. Piecewise linear segmentation is a well-known technique and there are various approaches (e.g., [12]) to segmenting a signal.

We compute the piecewise linear segmentation of the signal with a *greedy iterative top-down* approach. At each iteration, piecewise linear segmentation picks a segment to split and the point within the segment to use for the split. While doing this, it attempts to minimize the overall error of the new segmentation. Each split reduces the overall error. Once the algorithm has picked a split point, it does not reconsider that decision (i.e., it is greedy). Piecewise linear segmentation uses least squares linear regression, described above, to compute the line segments at each iteration.

We now describe the two correlators that exploit this segmentation. These correlators require the user to pick the number of segments. In our system, users can visualize the segmentation for each number of segments and pick the one that looks best to them.

5.3.1 Same Splits

The *same splits* correlator segments one sequence using the greedy top-down approach described above (sequence T in Figure 6). It then segments and fits (using linear regression) the second sequence forcing the second sequence to have the same segment boundaries as the first sequence (sequence M in Figure 6). Since we are using the segment boundaries of T for M, we may or may not get a good fit. We use the inverse of the error of the fit for M as the correlation score. The error of the fit is the sum of the square of the difference between the points on the line segments and the corresponding points on the original sequence.

5.3.2 Best Splits

The *best splits* correlator segments each sequence individually using the same *number* of segments but does not enforce the same segment boundaries. For example in Figure 7 we have used the same number of segments (2) for both T and M, but the segment boundaries for T and M are different. We use the inverse of the cumulative difference between the corresponding boundaries of the two sequences (d_i) as the correlation score.

Both the same splits and best splits correlators reward

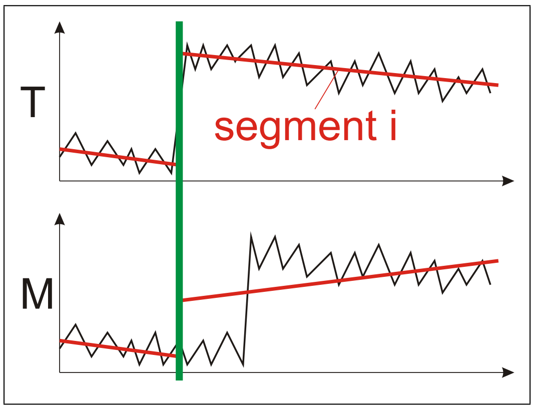


Figure 6: Illustration of Same Splits

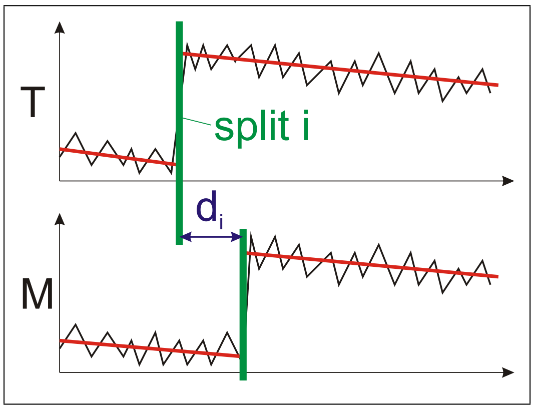


Figure 7: Illustration of Best Splits

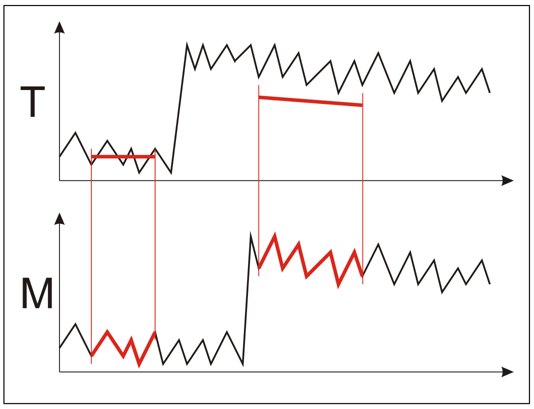


Figure 8: Illustration of Manual Linear Pattern (MLP)

sequences that have the same natural segment boundaries.

5.3.3 *Manual Linear Pattern (MLP)*

Often times a performance analyst can immediately pick out a pattern of interest and abstract the pattern away from the noise in the data. The manual linear pattern (MLP) correlator exploits the performance analyst's intuition. The performance analyst manually segments one of the sequences (sequence T in Figure 8). From this point onwards, this correlator behaves like Pearson's r, but instead of using the original T sequence, it uses the manual segmentation. Its correlation score is the absolute value of Pearson's r.

5.4 Discussion

The correlators enable the performance analyst to prioritize her efforts so that she looks at the most highly correlated (and thus most likely to be relevant) metrics first. Correlation, however, does not imply causality: just because a metric correlates well with the target metric it does not mean that the metric causes the anomaly in the target metric. To determine causality, the performance analyst needs to bring domain knowledge to bear on the problem.

6. EVALUATION OF CORRELATORS

Section 6.1 presents the experimental methodology for evaluating correlators. Section 6.2 evaluates the correlators. Finally, Section 6.3 discusses our findings and suggested future work.

6.1 Methodology

A given performance anomaly may occur many times in a metric. For example, the periodic pattern repeats throughout the execution of the db benchmark. In our prior work [10], we used visual and statistical correlation over the entire metric. However, this approach has three problems:

1. If the metric is long, it may take a long time to compute the correlation score;

2. Some occurrences of the anomaly may be noisy while others not. If we correlate over the entire metric, the noise may degrade the effectiveness of the correlators.

3. A given metric may suffer from multiple performance anomalies. Thus, some occurrences of an anomaly may

be superimposed with other anomalies. This, once again, can degrade the effectiveness of the correlators.

Thus, in this work we correlate not just on the entire metric but also on subsequences of a metric. We use trace alignment to align the user-selected pattern with corresponding patterns in other traces. During correlation, we ignore all parts of the trace that fall outside the pattern. Enabling the performance analyst to pick subsequences allows the analyst to focus on relatively noise-free and unperturbed instances of an anomaly.

Table 3 illustrates the seven anomalies that we use in our evaluation. We chose these anomalies from the performance anomalies described in Section 2. Column "Complete IPC metric with anomaly shaded" shows the entire sequence for the IPC. The shaded area in the sequence identifies the anomaly that we are interested in. Column "Name" names the anomaly. For example the *H PGD* anomaly, contains an instance of the pre-GC dip phenomenon in hsql. In the gradual increase anomaly in jbb, *J GI 2*, the anomaly contains two subsequences on which to perform correlation. Column "Category" categorizes the anomalies. *Continuous patterns* contain subsequences that change slowly over time. *Discontinuous patterns* contain subsequences that change sharply. Finally, *distorted patterns* contain subsequences that appear to contain significant noise (this is a judgment call on our part). The distorted category may include anomalies from the continuous or discontinuous categories. Because we do not have enough case studies (each case study takes weeks to months to do) we decided to not split out the "distorted anomalies" class into "distorted continuous anomalies" and "distorted discontinuous anomalies".

Because our goal is to remove the correlation responsibility from a human analyst, we evaluate our correlators with respect to how they perform compared to a human expert. Thus, to perform this evaluation, one of the authors manually labeled each metric with *yes*, *maybe*, or *no*, depending on whether or not the pattern in the metric visually correlates with the pattern in the target metric. We provided a qualitative correlation score (yes, no, or maybe) rather than a fine grained correlation score (e.g., L1CacheMiss has correlation score of 0.95 and, L2CacheMiss has a correlation score of 0.8, etc.) because it is very hard to objectively score several hundred metrics manually. We use the qualitative score to rank the metrics such that all "yes" metrics are ranked higher than all "maybe" metrics, which in turn are ranked higher than all "no" metrics. We compute Spearman's coefficient on this ranking and with each correlator's ranking. Because the human did not fully rank the metrics, but just labeled them with one of three labels, Spearman's coefficient can never reach its extreme values $+1$ or -1. Thus, we normalized the output of Spearman's so that if the automatic correlation put all the "yes" before all the "maybe" and all the "maybe" before all the "no" metrics, then we had a correlation value of 1. If a correlator puts a "maybe" before a "yes", then its normalized Spearman coefficient is lower than one. If a correlator puts a "no" before a "yes" then its Spearman coefficient is lower than the correlator that puts a "maybe" before a "yes".

For the three correlators that require additional user input we did the following. For the "Best Splits" and "Same Splits" correlators, we picked the number of segments based on what suited the target metric the best. This was easy and quick with our GUI. For the MLP correlator we used the first natural-looking manual segmentation that we came up with (i.e., we did not try different segmentations to find one that gave the best results).

6.2 Evaluation

We now evaluate the performance of the eight correlators presented in Table 2 on the seven patterns presented in Table 3. For each correlator, Figures 9 shows its accuracy *as compared to visual correlation* presented in Section 6.1. We group the patterns in the figure according to their "category" (see Table 3). To improve readability, we start the y-axis at 0.5 instead of 0.

6.2.1 Overall Results

From Figures 9 we see that the MLP is either the best or one of the best performing correlators for all the patterns. Perhaps most important, there is no pattern where MLP performs poorly. More specifically, the lowest MLP bar is at 0.83, while all other correlators have a bar at or below 0.60.

6.2.2 Evaluation by Correlator

For four of the seven patterns (*J GI 1*, *D SLP*, *C SI*, and *D FLP*), the two statistical correlators (Pearson's coefficient and Spearman's coefficient) perform similarly. Of the remaining three patterns, Spearman does significantly worse in two (*J GI 2* and *H PGD*) and Pearson does significantly worse in one (*D D*). Both correlators perform poorly for the *D D* pattern. The fact that both correlators are below 0.60 for some anomaly confirms our previous experience that statistical correlation does not work as well as visual correlation.

Even though MLP uses Pearson's coefficient, it usually significantly outperforms Pearson's coefficient. The key advantage of MLP over Pearson's coefficient is that MLP gets assistance from the human in filtering out the noise in the data (because the human performs the piecewise-linear segmentation). Thus we conclude that Pearson's coefficient is significantly affected by noise in our data. Because Spearman's coefficient does not outperform MLP, we conclude that Spearman's noise filtering is not sufficient for our needs.

Figure 9 illustrates that there is no clear winner between the distance-based correlators: Euclidean, Manhattan and DTW. All three distance-based correlators are always close to each other and are usually comparable in performance to one of the statistical correlators. The main exception to this is the *D D* pattern where the distance-based correlators are far worse than the statistical correlators. The Euclidean distance performs worse than the Manhattan distance for *D D* because the significant outliers in *D D* affect Euclidean distance (with $M = 2$) more than Manhattan distance (with $M = 1$). Also, while DTW has been used almost exclusively for correlation in the past, it performs relatively poorly for our needs.

Of the automatic segmentation-based correlators (same splits and best splits), same splits beats best splits in six out of the seven patterns, and for the pattern that best splits wins (*C SI*) it is by a small margin. In contrast, in two of the patterns (*J GI 1* and *J GI 2*) best splits is over 30% worse than same splits. On visually inspecting the segmentations, we found that same splits actually does a good job of the segmentation. However, it still performs much worse than MLP. Therefore we conclude that using the linear regression

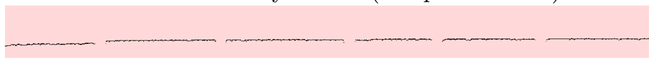
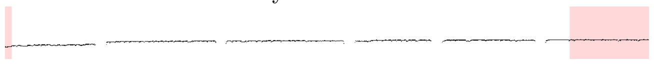
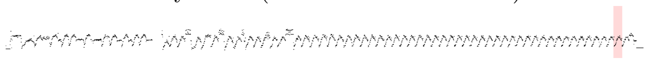
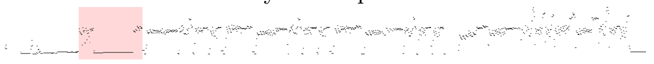
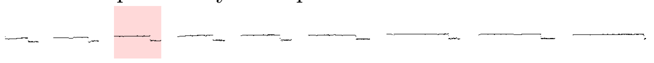
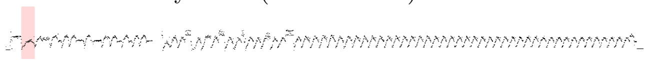
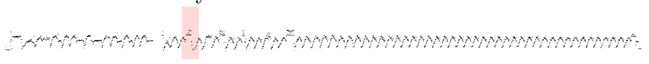
Complete IPC metric with anomaly shaded	Name	Category
Gradual increase anomaly in Jbb (complete metric) 	J GI 1	continuous
Gradual increase anomaly in Jbb 	J GI 2	continuous
Periodic anomaly in db (second to last instance) 	D SLP	continuous
Sudden increase anomaly in compress 	C SI	discontinuous
Pre-GC dip anomaly in hsql 	H PGD	discontinuous
Periodic anomaly in db (first instance) 	D FLP	distorted
Distortion anomaly in db 	D D	distorted

Table 3: Performance Anomalies Used in our Study

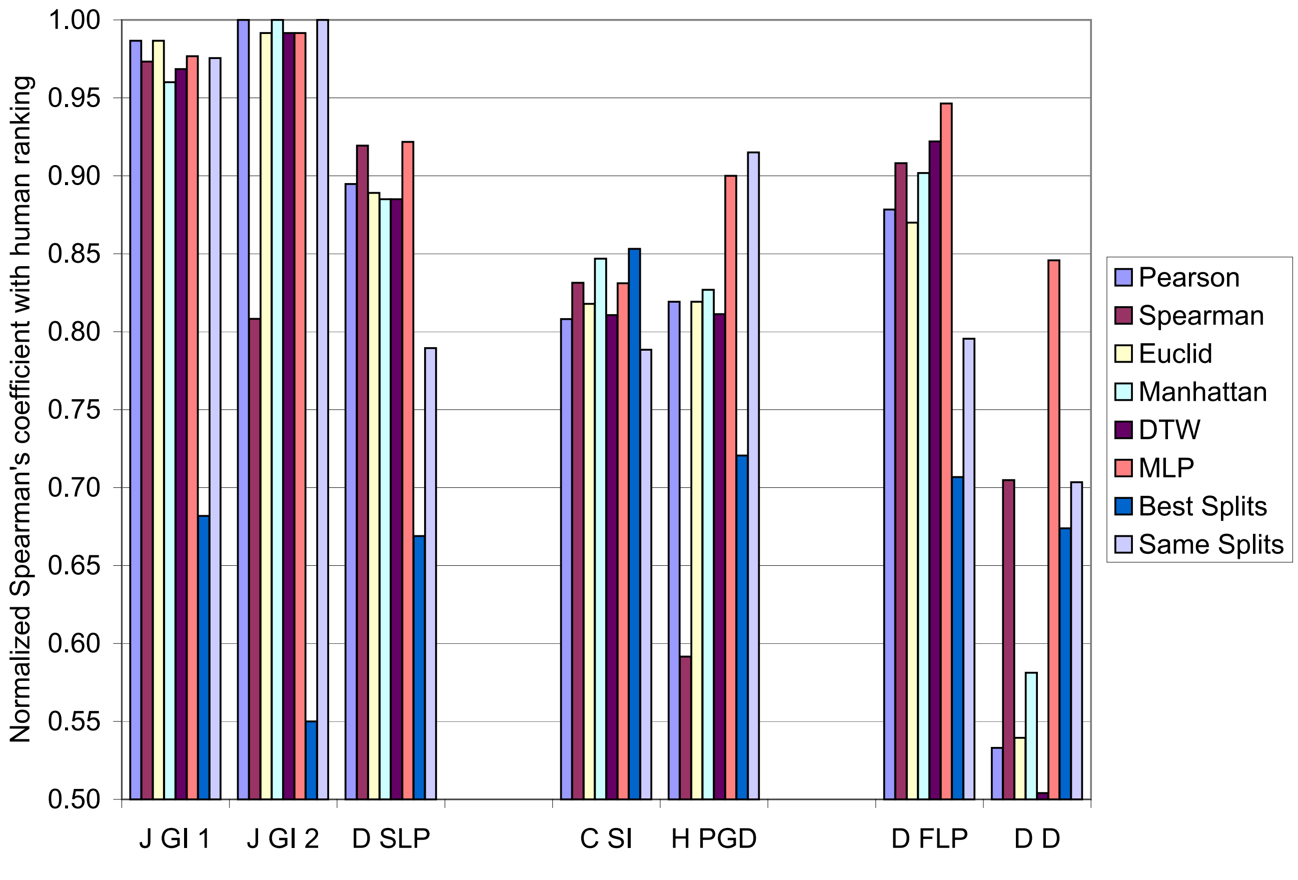


Figure 9: Correlator Accuracy: Compared to Visual Correlation

error to get the correlation score is inferior to using Pearson's coefficient (which is what MLP uses). We will explore the reason behind this in future work.

6.2.3 Evaluation by Category

For the continuous patterns, which make up the first group of bars, Figure 9 illustrates that five of the correlators achieve at least 0.88 accuracy (Pearson coefficient, Euclid, Manhattan, DTW, and MLP). For *J GI 2*, six of the eight correlators get close to 1.0 (i.e., fully consistent with the human correlation). Best splits does significantly worse than the other correlators on all three patterns: never better than 70%. The reason for best split's poor performance is that in a continuous pattern it is harder to pick segment boundaries (because there are no abrupt changes that are obvious for boundaries). Thus two patterns may end up with very different "best" boundaries even though visually they are of a nearly identical shape. Same split does better than best split for the continuous metrics because it uses the same boundaries for all the metrics.

For the discontinuous patterns, Figure 9 illustrates that the overall correlator accuracy is worse than for the continuous patterns, with the Pearson's coefficient being closer 0.8 than to 0.9. That said, best splits performs the best for *C SI* but poorly for *H PGD*. On the other hand, same splits performs the best for *H PGD* and the worst for *C SI*; there is no clear winner between these two.

For the distorted patterns, we see that the manual linear pattern (MLP) performs best by far, because this correlator uses the performance analyst's manually drawn line, which does not contain any noise. No other correlator does well for both of these patterns, although some do well with one. As expected, Spearman's coefficient does better than Pearson's coefficient for both patterns because using the rank of the values helps to eliminate noise. Furthermore, because MLP uses Pearson's coefficient after manual piecewise-linear segmentation and MLP does significantly better than Pearson's coefficient, we conclude that Pearson is affected to noise in our data.

6.2.4 Subsequences Versus Complete Metrics

In prior work we had correlated on the entire metric. One of the contributions of this work is that we propose correlating on subsequences of the metric. Section 6.1 discusses why we believe that correlating on subsequences is a better idea than correlating on the entire sequence. In this section we discuss whether or not we realized any of the benefits.

From Figure 9 we see that the bars for *J GI 1* and *J GI 2* differ slightly for the better performing correlators. The difference between *J GI 1* and *J GI 2* is that the former correlates on the entire metric, while the latter correlates on two subsequences of the metric. Therefore, at least for these two anomalies, selecting a set of subsequences in a metric yields better results than selecting the complete metric. In addition, the poor performance of most correlators on the *D D* bars emphasizes that noise in the data can significantly degrade their performance. Because our methodology does not require correlation on the entire trace, it enables users to correlate on cleaner subsequences of the trace and thus obtain better results.

6.3 Discussion

Our evaluation validates our previous observations that statistical correlation may be inaccurate with respect to visual correlation. Our new correlator, MLP, is better or almost as good as statistical correlation across the board. Our evaluation also allowed us to better understand the strengths and weaknesses of the different mechanisms.

Nevertheless, even our best correlator, MLP, is not perfect. In the worst case, its Spearman's coefficient with human correlation is as low as 0.83. While 0.83 sounds high, it may be that some of the metrics that MLP fails to find as correlated are the key causal metrics for the performance anomaly of interest. In future work we will investigate how to combine multiple correlators in an attempt to achieve better accuracy.

7. RELATED WORK

This section surveys work related to vertical profiling. This includes work on performance studies of Java workloads, performance visualization and analysis tools, and time series data mining.

7.1 Performance Studies of Java Workloads

Java middleware and server applications are an important class of emerging workloads. Existing research uses simulation and/or hardware performance counters to characterize these workloads. Cain et al. [5] evaluate the performance of a Java implementation of the TPC-W benchmark and compare the results to SPECweb99 and specjbb. Shuf et al. [20] analyze the memory performance of SPECjvm98 and pBOB on an IBM PowerPC processor using simulation and hardware performance counters. Luo and John [14] evaluate specjbb and VolanoMark on a Pentium III processor using the Intel hardware performance counters. Seshadri, John, and Mericas [19] use hardware performance counters to characterize the performance of specjbb and VolanoMark running on two PowerPC architectures. Karlsson et al. [11] characterize the memory performance of Java server applications using real hardware and a simulator. They measure the performance of specjbb and ECPerf on a 16-processor Sun Enterprise 6000 server. Other studies focus on behavior impacting specific subsystems, like Dieckmann et al. [7], who investigate memory performance metrics of interest for garbage collection designers. These studies generally focus on the overall characteristics of the workloads. We are interested in the causes of temporal performance phenomena, we present tools and techniques for correlating performance information, gathered in multiple runs of a benchmark, across different levels of a system.

7.2 Performance Visualization and Analysis Tools

A large body of work exists on performance visualization. Kimelman et al. [13] introduce PV, a performance visualizer focused on presenting temporal information from various levels of the system. PV shows only a subsection of the whole trace, but it allows scrolling through the whole trace, thereby continually updating the subsection currently visualized. Mellor-Crummey et al. [15] present HPCView, a performance visualization tool together with a toolkit to gather hardware performance counter traces. They use sampling to attribute performance events to instructions, and then hierarchically aggregate the counts, following the loop nesting structure of the program. Their focus is on attributing performance counts to source code areas. Miller et al. [16]

present Paradyn, a performance measurement infrastructure for parallel and distributed programs. Paradyn uses dynamic instrumentation to count events or to time fragments of code. It can add or remove instrumentations on request, reducing the profiling overhead. Metrics in Paradyn correspond to everything that can be counted or timed through instrumentations. The original Paradyn does not support multithreading, but Xu et al. [23] introduce extensions to Paradyn to support the instrumentation of multithreaded applications. Zaki et al. [25] introduce an infrastructure to gather traces of message-passing programs running on parallel distributed systems. They describe Jumpshot, a trace visualization tool, which is capable of displaying traces of programs running on a large number of processors for a long time. They visualize different (possibly nested) program states, and communication activity between processes running on different nodes. The newer version by Wu et al. [22] is also capable of correctly tracing multithreaded programs. Pablo, introduced by Reed et al. [18], is another performance analysis infrastructure focusing on parallel distributed systems. It supports interactive source code instrumentation, provides data reduction through adaptively switching to aggregation when tracing becomes too expensive, and introduces the idea of clustering for trace data reduction. DeRose et al. [6] describe SvPablo (Source View Pablo), loosely based on the Pablo infrastructure, which supports both interactive and automatic software instrumentation and hardware performance counters, to gather aggregate performance data. They visualize this data for C and Fortran programs by attributing the metric values to specific source code lines.

To the best of our knowledge none of the above tools provides a mechanism to integrate information from multiple runs of a benchmark, or support to for the identification of causal relationships between performance metrics.

Recent work uses statistical techniques to analyze performance counter data. Eeckhout et al. [8] analyze the hardware performance of Java programs. They use principal component analysis (PCA) to reduce the dimensionality of the data from 34 performance counters to 4 principal components. Then they use hierarchical clustering to group workloads with similar behaviors. They gather only aggregate performance counts, and they divide all performance counter values by the number of clock cycles. Ahn and Vetter [2] hand-instrument several code regions in a set of applications. They gather data from 23 performance counters for three benchmarks on two different parallel machines with 16 and 68 nodes. Then they analyze that data using different clustering algorithms and factor analysis, focusing on parallelism and load balancing.

Even though the correlators in our vertical profiling approach are reducing the number of metrics that need to be studied from a large number down to a few metrics, we do not want to reduce the dimensionality of the captured data per se (i.e. by reducing 300 metrics down to 5 principal components). The correlators are intended to find the small number of metrics that are potential causes of the pattern in the target metric. PCA would reduce the number of metrics, but it would do so by introducing new, derived metrics (the principal components), which would be linear combinations of many metrics.

7.3 Time Series Data Mining

The field of time series data mining provides many techniques useful for the analysis of performance data traces, since all temporal performance data can be considered a time series.

Our approach for trace alignment is based on the dynamic time warping (DTW) technique [17], first introduced into the data mining community in 1994 [4]. The data mining community primarily uses DTW for the comparison of two time series, or two subintervals in a time series. This interpretation of the DTW distance as a similarity measure has been used for many data mining applications, such time series database queries [24], but also for the alignment of gene expression time series [1]. To the best of our knowledge we are the first to use DTW for aligning performance traces, and thus to enable the automatic performance analysis across multiple complementary traces produced by nondeterministic (real) systems.

Our segmentation based correlators were inspired by prior work on piecewise linear segmentation [12, 9]. Unlike [9], our goal is not to turn sequences of values into discrete events; but the overall idea of identifying change points in time series is the same. We use these change points for the purpose of determining the similarity between two time series (i.e. with the same splits and best splits correlators).

8. CONCLUSIONS

While most modern processors are capable of executing multiple instructions per cycle, most applications do not even come close to utilizing this potential. For example, the PowerPC POWER4 processor is capable of executing five instructions per cycle; however, in our experience with a number of standard Java applications, the average number of instructions completed per cycle is under 1.1 for all the benchmarks. In other words, performance tuning techniques have the potential for dramatically impacting performance. However, before we can tune the performance of an application we must first understand its behavior. Unfortunately, even with sophisticated visualization and statistical tools understanding the performance of modern applications is very labor intensive [10]. This paper describes and evaluates techniques for automating two labor intensive activities of performance analysis: aligning of traces and correlation.

We evaluate one technique for aligning traces: dynamic time warping (DTW) from the speech recognition literature. We show that for our experiments, DTW performs extremely well. To increase the generality of our results, we use not only traces from real benchmarks but also synthetic traces to evaluate DTW. We evaluate eight techniques for correlation. These techniques include techniques from prior work (e.g., statistical correlation) and one new technique: Manual Linear Pattern (MLP). We show that the techniques from prior work are inconsistent: they work well for some situations and not so well for others. Our MLP correlator is the best or close to the best correlator in all our experiments.

9. REFERENCES

[1] John Aach and George Church. Aligning gene expression time series with time warping algorithms. *Bioinformatics*, Vol. 17:495–508, June 2001.

[2] Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–16. IEEE Computer Society Press, 2002.

[3] Bruce G. Batchelor. *Pattern Recognition: Ideas in Practice.* Plenum Press, 1978.

[4] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Working Notes of the Knowledge Discovery in Databases Workshop*, pages 359–370, July 1994.

[5] Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko H. Lipasti. An architectural evaluation of Java TPC-W. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 229–240, Nuevo Leone, Mexico, January 2001.

[6] Luiz DeRose and Daniel A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *Proceedings of the International Conference on Parallel Processing*, Fukushima, Japan, September 1999.

[7] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Springer Verlag, June 1999.

[8] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 169–186, 2003.

[9] Valery Guralnik and Jaideep Srivastava. Event detection from time series data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 33–42, 1999.

[10] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* ACM Press, October 2004.

[11] Martin Karlsson, Kevin E. Moore, Erik Hagersten, and David A. Wood. Memory system behavior of Java-based middleware. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, pages 217–228, Anaheim, California, February 2003.

[12] Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An online algorithm for segmenting time series. In *Proceedings of the International Conference on Data Mining*, pages 289–296, 2001.

[13] Doug Kimelman, Bryan Rosenburg, and Tova Roth. Strata-various: Multi-layer visualization of dynamics in software system behavior. In *Proceedings of the conference on Visualization (VIS'94)*, pages 172–178. IEEE Computer Society Press, October 1994.

[14] Yue Luo and Lizy Kurian John. Workload characterization of multithreaded Java servers. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 128–136, Tucson, Arizona, November 2001.

[15] John Mellor-Crummey, Robert Fowler, and Gabriel Marin. HPCView: A tool for top-down analysis of node performance. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, New Mexico, October 2001.

[16] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[17] Chotirat Ann Ratanamahatana and Eamonn Keogh. Everything you know about dynamic time warping is wrong. In *Third Workshop on Mining Temporal and Sequential Data*, August 2004.

[18] Daniel A. Reed, Ruth. A. Aydt, Roger J. Noe, Philip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, October 1993.

[19] Pattabi Seshadri, Lizy John, and Alex Mericas. Workload characterization of Java server applications on two PowerPC processors. In *Proceedings of the Third Annual Austin Center for Advanced Studies Conference*, Austin, Texas, February 2002.

[20] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 194–205. ACM Press, 2001.

[21] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04)*, May 2004.

[22] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.

[23] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *Principles Practice of Parallel Programming*, pages 49–59, 1999.

[24] Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. Efficient retrieval of similar time sequences under time warping, February 1998.

[25] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.