Revisiting End-to-End Trace Comparison with Graph Kernels

Jonathan Mace *Rodrigo Fonseca*
Brown University

Abstract

End-to-end tracing has emerged recently as a valuable tool to improve the dependability of distributed systems by performing dynamic verification and diagnosing correctness and performance problems. End-to-end traces are commonly represented as richly annotated directed acyclic graphs, with events as nodes and their causal dependencies as edges. Being able to automatically compare these graphs at scale is a key primitive for tasks such as clustering, classification, and anomaly detection. In this paper we explore recent developments in the theory of graph kernels, and investigate the feasibility of using a family of kernels based on the Weisfeiler-Lehman graph isomorphism test [35] as an efficient and robust graph comparison primitive. We find that graph kernels provide a good formulation of the execution graph comparison problem, and present preliminary but encouraging results on their ability to distinguish high-level differences between execution graphs.

1 Introduction

End-to-end tracing has emerged in the past decade as a valuable tool to diagnose correctness and performance problems in distributed systems [14, 28, 30, 36, 37], model workloads, resource usage, and timings [4, 37, 24, 9, 36], and to detect anomalous requests at runtime [4, 10, 30, 26]. By recording causally-related events across the boundaries of components, layers, machines, and even administrative domains, end-to-end tracing avoids problems with more traditional component-centric logging [25]. First, it eliminates the process of correlating entries across multiple logs using ad-hoc identifiers and inference, which can be complex, error prone, or not even possible [4]; second, it may avoid depending on clocks, which can be unsynchronized among machines; third, when done with runtime support, allows for coherent sampling, or the capturing of only the events that are causally related to an initial event.

A common representation for end-to-end traces is a labelled, directed, acyclic graph, or an *execution graph*. These graphs are rich in information about the instrumented system: they record paths taken, timing between events, and resource usage. They are, however, hard to analyze manually, as an instrumented system can produce a large number of graphs, they can become large and complex, and hard to visualize effectively.

Because of this, many previous works sought to extract value from end-to-end traces by automatically analyzing series of execution through, *e.g.*, clustering, classification, or anomaly detection. In all of these tasks, graph comparison appears as a recurring primitive [4, 10, 24, 30]. There is however, room to improve. Sambasivan *et al.* [30] mention that off-the-shelf clustering algorithms produced clusters that were 'too coarse-grained and unpredictable', and that metrics better aligned with developer's expectations of similarity are needed. Some of the methods used, such as string-edit-distance of linearized versions of the graph [4, 30], or the probabilistic context free grammar from [10] lose parallelism and concurrency infomation, which may be detrimental in some use cases.

In this paper we explore recent developments in the theory of graph kernels for graph comparison, as a potential first step in addressing these limitations. In particular, we investigate the feasibility of using a family of scalable graph kernels based on the Weisfeiler-Lehman isomorphism test as a common and flexible framework for comparing execution graphs [35]. Our contributions are twofold: we (i) identify the underlying similarities in execution graph comparison methods from previous work; and (ii) present a preliminary evaluation of four graph kernels for execution graph comparison in the context of clustering and one application, *stratified sampling*.

The rest of this paper is organized as follows. In §2, we discuss causal tracing, graph comparison methods utilized in previous work, and candidate methods from graph theory. We provide a formulation for graph kernels in §3. We perform a number of experiments and provide evaluation in §4. In §5 we discuss the future directions of execution graph comparison and end-to-end tracing.

2 Background and Related Work

2.1 Causal Tracing

Traditional approaches to debugging and profiling end-to-end requests – *ad-hoc* analysis and per-machine log-

ging - scale poorly in large-scale distributed systems. Causal end-to-end tracing has emerged as a valuable tool for improving the dependability of these systems by recording, diagnosing and analyzing their execution across components. End-to-end tracing frameworks correlate events across multiple machines and record their causality according to Lamport's happens before relation [22]. A number of such frameworks exist both in academia [4, 9, 14, 28, 37] and in industry [36, 11, 3, 38, 12], and have been used for a variety of purposes, including diagnosing anomalous requests whose structure or timing deviate from the norm [4, 10, 30, 26]; diagnosing steady-state problems problems that manifest across many requests. [28, 37, 14, 36, 30]; identifying slow components and functions [9, 36, 24]; and modelling workloads and resource usage [4, 24, 37].

A common representation for requests recorded by such frameworks is an *execution graph*, a directed, acyclic graph that describes the path of a single request through components of a distributed system. Events that occur during the request's execution are represented as nodes in the graph, with the root node representing the start of the request. An edge from one node to a subsequent node only exists if the corresponding events satisfy Lamport's happened-before relationship [22]. Events may have further information, such as labels, wall-clock timing, and resource usage – which can then be attributed to nodes and edges in the execution graphs. An individual execution graph provides a description of a request as it traverses a distributed system – it captures the path of the request, performance costs incurred at the components visited, and timings between events. Collectively, execution graphs can represent aggregate system behavior - capturing the commonly-traversed paths, corner case executions, and distributions over paths and timings.

Execution graphs capture a lot of useful information, but they are difficult to manually analyze for three reasons: graphs can be large, growing to thousands of nodes; the nodes and edges of a graph can incorporate a large amount of information; and large graphs cannot easily be visualized or compared. Consequently, prior work in the area emphasizes automated approaches to comparing and reasoning about execution graphs.

2.2 Execution Comparison in Prior work

Despite the varied end goals of prior work, a common theme is the need to compare or cluster execution graphs. The specific graph comparison methods utilized vary by application, but there is broad similarity in the features selected for comparison. In this section we look into four examples from the literature.

Magpie [4] correlates events generated by the operating system, middleware, and applications, and infers causal relations between events to produce execution graphs. The authors use comparison based on a simple string-edit-distance metric on flattened execution graphs as a basis for execution clustering. Their approach discards some structural and temporal information, and whilst the technique produced reasonable results, the authors acknowledged the need for tree- and graph-edit distance algorithms.

Pinpoint [10] collects execution traces as a series of paths through the system. The authors diagnose anomalies by generating a probabilistic context-free grammar (PCFG) from the paths. They perform anomaly detection at runtime, whereby new paths are compared to the generated model to determine the probability with which it would be produced from the grammar. Anomaly detection worked well in experiments, but the authors note that there are a number of realistic scenarios where it would not work well: features must be represented in the training set for them to be considered at runtime; changes such as software upgrade require the model to be retrained; and the learned model represents a superset of observed paths.

Spectroscope [30] collects execution traces represented as process invocation trees, and diagnoses performance changes by comparing sets of before- and after-traces. Spectroscope assumes that a similar workload was run before and after the performance change, and that the performance change manifests as a change in distribution over the request structures and/or request timings. To diagnose a change, spectroscope compares the distributions of service completion times for graphs that are topologically identical, and compares structural differences between executions using string-edit-distance.

Mann *et al.* [24] collect execution traces from datacenter services, and model the latency of a service given the child services invoked. The execution graphs recorded do not fully capture the causal dependencies internal to a service, so one component of the work is to deduce those causal dependencies from a collection of training examples. The training examples are then clustered if they have identical execution graphs. At runtime, a cluster is selected by comparing its service timings with the cluster centroids, and selecting the nearest-neighbour. A prediction for the execution's overall runtime is then given based on the other executions in the selected cluster.

There are a few common limitations in these works. Mann *et al.* and Spectroscope do clustering, but only for graphs that are isomorphic. For graphs with different topology, Spectroscope and Magpie use string-edit-distance on canonicalized and linearized graphs as a metric, for proper clustering in the latter, and for finding similar clusters in the former. Both this linearization, and the PCFG from Pinpoint lose information about the concurrency structure of the graph, which can be important

for many applications. We argue that these limitations stem from the lack of graph comparison methods that are both *effective* and *efficient*. While it is true that more general graph kernels have been traditionally quite expensive (§2.3), new developments in efficient graph kernels warrant an investigation on their suitability for this setting.

2.3 Graph Comparison Methods

Much prior work on graph comparison exists in the fields of graph theory and machine learning [1]. Application domains include bioinformatics [32], data mining [40], social network analysis [21], and more [5, 6, 20]. Of particular interest to us are techniques for graphs with labelled nodes and edges. The size of execution graphs can extend to thousands of nodes, but this remains small when compared to some applications, which have graphs in the hundreds of millions of nodes [21, 40]. This accords us flexibility for considering techniques that may not be tractable for some application domains.

Candidate techniques include frequent subgraph mining [18], maximal common subgraphs [8], graph edit distance [29], graph isomorphism [13], and graph kernels [39]. Among these, graph kernels are attractive because they can tolerate approximate matches and can incorporate node and edge labels, timings, and other features in a uniform manner. Also, by formulating graph comparison in terms of kernel functions, subsequent applications can utilize a wealth of off-the-shelf kernel methods from the machine learning domain [15]. Recent developments, which we describe in §3.1, have produced graph kernels that are efficient to compute while producing results that are close to previous state-of-the-art, but expensive, graph kernels.

3 Graph Kernels for Execution Graphs

A graph $G = (V, E, \ell)$ consists of a set of vertices $V = \{v_0, v_1, ..., v_n\}$, a set of directed edges $E \subset V \times V$, and a labelling function $\ell : V \rightarrow \Sigma$ that assigns labels from an alphabet Σ to verticies.

When G represents an execution graph, V is constructed from the events of the execution, E from the causal edges, and ℓ is constructed from the user-specified labels that the execution graph records for each event. That is, for any nodes $v_0 \in G_0$ and $v_1 \in G_1$, $\ell(v_0) = \ell(v_1)$ implies that v_0 and v_1 are different occurrences of the same event. Commonly this means that the events were generated by the same line of code, though specifying event labels is a design decision at the time of instrumentation.

3.1 Graph Kernels

Informally, a graph kernel is a function that takes two graphs, G_0 and G_1 as input, and produces a numeric value as output quantifying their similarity. It must satisfy two mathematical requirements; namely, it must be symmetric and positive semi-definite. For a rigorous mathematical definition, see [31].

A variety of popular graph kernels exist in the literature, broadly falling into three categories: kernels based on walks and paths [7, 19, 27], kernels based on subgraphs [16, 34], and kernels based on subtree patterns [23, 27]. The more recent Weisfeiler-Lehman kernel and framework [33, 35] present generalizations that encompass a number of previously described kernels. Here, we give descriptions for the *node-count kernel*, *edge-distance kernel*, as well as a description of the *Weisfeiler-Lehman framework*. For an overview of other kernels, we refer the reader to [39].

Node-count kernel The node-count kernel simply counts mutual occurrences of labelled nodes in the graphs

$$K_{NC}(G_0, G_1) = \sum_{u \in V_0} \sum_{v \in V_1} f(u, v)$$

$$\text{where } f(u, v) = \begin{cases} 1 & \text{if } \ell_0(u) = \ell_1(v), \\ 0 & \text{otherwise.} \end{cases}$$

Edge-distance kernel The edge-distance kernel counts mutual occurrences of edges in the graphs. Here, we incorporate edge weights ω into the kernel score.

$$K_{ED}(G_0,G_1) = \sum_{(a,b) \in E_0} \sum_{(c,d) \in E_1} f(a,b,c,d)$$

$$\text{where } f(a,b,c,d) = \begin{cases} \dfrac{\min(\omega_{ab},\omega_{cd})^2}{\max(\omega_{ab},\omega_{cd})} & \text{if } \ell_0(a) = \ell_1(c) \\ & \text{and } \ell_0(b) = \ell_1(d), \\ 0 & \text{otherwise.} \end{cases}$$

Weisfeiler-Lehman framework The Weisfeiler-Lehman framework provides a computationally efficient method for computing graph kernels over successively-expanding subgraphs of the input graphs. The framework is paramaterized by a depth d, and a different user-specified graph kernel K. The framework describes a method for expanding input graphs A and B into sequences of relabelled graphs $\{A_1, ..., A_d\}$ and $\{B_1, ..., B_d\}$. Subsequently, the graph kernel K can be applied to each pair of graphs in the sequences:

$$K_{WL} = \sum_{i=1}^{d} \alpha_i K(A_i, B_i)$$

We refer the reader to [35] for a more rigorous definition. When the Weisfeiler-Lehman framework is paramaterized with $K = K_{NC}$, this is called the Weisfeiler-Lehman kernel. The Weisfeiler-Lehman framework is based on the Weisfeiler-Lehman isomorphism test; as such, for increasing values of d, the Weisfeiler-Lehman framework itself tends towards testing for isomorphism. In our evaluation, we include the Weisfeiler-Lehman framework paramaterized with the node-count and edge-distance kernels, and a depth of 3. The Weisfeiler-Lehman kernel is an appealing choice because it presents a computationally efficient method for operating on subgraphs. We select a depth of 3 so as to incorporate sufficient detail for reasoning about branch points in executions.

3.2 Applicability to Prior Work

For each of the examples in §2.2, graph kernels are a viable alternative for comparing the execution graphs used. For both Magpie and Spectroscope, graph kernels such as the edge-distance kernel could directly replace the string-edit-distance metric they use, and it would an interesting experimental comparison to see how they perform. In the case of Mann *et al.*, which compares timings only between graphs with identical topology, kernels could be a viable alternative approach that relax the requirement to select a single topology to compare an execution to.

Pinpoint's context-free grammar encodes probabilistic expansion rules representing the probability that any given component will call a particular set of other components. We present an alternative conceptualization of this context-free grammar - that the probability of some new execution is simply the summation of graph kernel scores between the training set examples and the new execution. Depending on Pinpoint's specific implementation details, the graph kernel used could be a simple node-count, edge-count, or a path-based kernel.

4 Evaluation

In this section we evaluate the three kernels described in §3.1. We present two experiments whereby executions can be separated into broad high-level categories. We show that the graph kernels can effectively distinguish executions from each of the clusters. We also explore an example application of graph kernels, stratified sampling.

4.1 Experimental Setup

We instrumented Hadoop map-reduce v2.04 and YARN [2] with the X-Trace framework [14] and deployed it on a small, 10-node cluster. Using this instrumentation, we ran jobs from the HiBench benchmark [17] and recorded execution graphs of the jobs. We implemented the three kernels outlined in §3.1, and used them to compare recorded execution graphs. For comparison, we also implemented distance metrics based on total runtime and on the string-edit-distance of canonicalized linearizations of the graphs.

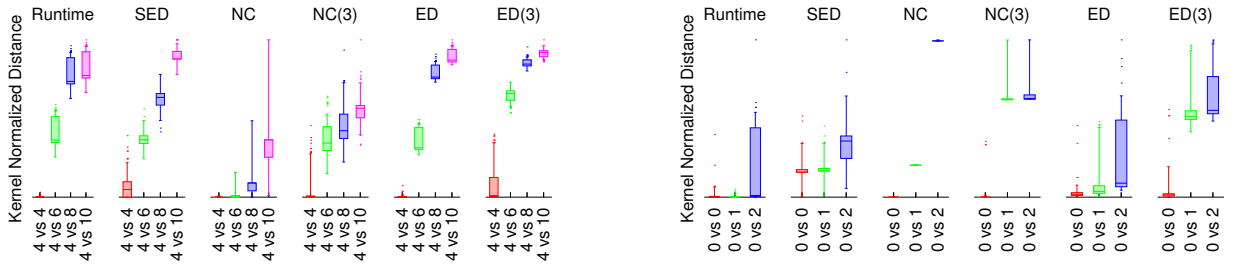
Varying Execution Topology We generated a 2.5GB input file split across 10 HDFS data nodes with a block size of 64MB. We sequentially ran a number of word-count map-reduce jobs, varying the number of nodes available for executing map-reduce tasks. For topologies of 4, 6, 8 and 10 machines, we ran 200 executions each, collected the resulting execution traces, and manually clustered them according to the number of nodes that were available in the cluster.

Varying Speculative Executions We generated a 2.5GB input file split across 10 HDFS data nodes with a block size of 64MB. We sequentially ran a number of wordcount map-reduce jobs across 10 machines, varying the Hadoop parameter that specifies the probability of speculative executions being initiated. After collecting the execution traces, we manually clustered them according to the number of speculative executions that Hadoop initiated. Our resulting data set includes executions with 0, 1 and 2 speculative executions.

4.2 Kernel Validation

Figure 1 shows the ability for the kernels to separate clusters of executions in the two experiments. For a given column, say, 4 vs 8, for each of the 200 executions in the '4 machine' group, we find the minimum distance to all traces in the '8-machine' group. The boxplot shows the distribution of this value over all executions in the source group. All values are normalized, since it is meaningless to compare actual scores across metrics. A metric is successful if there is little to no overlap between the first boxplot, of the distances within the group, and the minimum distances to the other groups.

In all cases, executions from within the same cluster have a low kernel distance. However, the amount of separation varies significantly. The runtime comparison does a good job in the varying execution topology experiment, but not in the speculative execution one. This makes sense, since speculative execution will not dramatically affect the runtime, whereas a topology with fewer nodes will be able to run fewer tasks in parallel and thus take longer. Similarly, we note that string-edit-distance behaves poorly in the speculative executions experiment. This occurs because there are numerous minor fluctuations in the graph topology between executions, dwarfing the small contribution of the inserted specula-



(a) Varying Execution Topology

(b) Varying Speculative Executions

Figure 1: Boxplots showing the separation between clusters for two experiments. Each plot compares clusters using runtime, string-edit-distance (SED), node-count kernel (NC), edge-distance kernel (ED), and the Weisfeiler-Lehman kernel paramaterized with depth 3 and node-count (NC(3)) or edge-distance (ED(3)) kernels.

tive execution nodes.

Both node-count kernels do well in the speculative execution experiment, where the number of events changes, but not so well in the varying topology, as the number of events is more uniform: with more machines, they are just spread across them.

For both experiments, the Weisfeiler-Lehman edge-distance kernel produces good results, as it takes into account both timings and topology. The higher variance seen in some comparisons can be attributed to subtle changes in execution graph topology that are exacerbated by the Weisfeiler-Lehman framework. One example of such a change is Hadoop's policy of batching map output data during the shuffle phase, which can be variable between similar executions. Thus, the variance that we see is not undesirable because it correctly reflects differences present in execution graphs within each cluster.

4.3 Stratified Sampling

In this section, we describe an example application using graph kernels: *stratified sampling*. By recording only a small fraction of requests, the resource overhead of capturing end-to-end traces can be as little as 1%. Dapper, for example, samples requests uniformly at random [36]. One consequence, however, is that a framework may fail to capture infrequent but interesting requests.

The goal of stratified sampling is to bias the sampling scheme such that dissimilar requests have a higher sampling probability. Graph kernels provide a measure of similarity that we can use to bias the decision on whether to persist an execution. Whilst we acknowledge that directly applying graph kernels at runtime would be an inefficient solution, graph kernels provide a means to assess potential features that could feasibly be used by tracing frameworks in efficient sampling implementations. Here, we evaluate a simple stratified sampling scheme based on graph kernel similarity. This approach assumes that traces would stay in memory for a period, and the decision in question is whether or not to persist the trace.

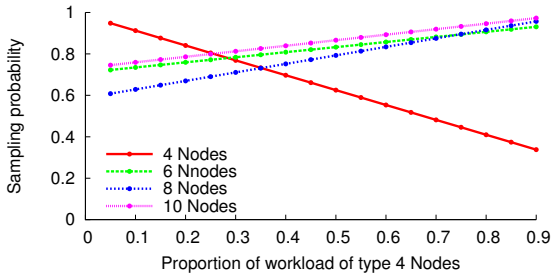


Figure 2: Sampling probabilities for requests from the 4-, 6-, 8- and 10-node clusters as the workload proportion of the 4-cluster varies

Using data from the *varying topology* experiment, we simulated workloads with varying proportions of requests. Given a new request r, we calculate its sampling probability based on the observed workload as $p(r) = 1 - \text{mean}\{K(r,w) | w \in \text{Workload}\}$.

Figure 2 shows the mean sampling probabilities for executions from each cluster as the workload varies. We vary the contribution of the 4-node cluster to the workload, from 5% to 95%. The remainder of the workload is distributed evenly over the other clusters. The figure shows how the sampling probabilities for requests from each cluster change. We observe that when the 4-node cluster accounts for only a small proportion of the workload, the sampling probability for 4-node requests is highest. As we increase the workload contribution of the 4-nodes cluster, the sampling probability for 4-node requests decreases. Simultaneously, the sampling probabilities for requests from the 6-, 8- and 10-node clusters increase as they consequently contributed a smaller proportion of the workload.

5 Conclusion and Future Work

In this work, we present initial results on the use of efficient graph kernels for comparing execution graphs from end-to-end tracing. While our experiments are preliminary – performed at very small scale, in controlled experiments, with data processed offline and centralized – the resuts encourage further investigation. Beyond validating the use of graph kernels for execution graph comparison, we intend to investigate their robustness to instrumentation loss, and also their performance on different instrumentations of the same system, to ultimately derive the most effective instrumentation for the system. If found to be suitable, graph kernels then provide a sound basis for the application of established machine learning techniques for anomaly detection, clustering, and classification, facilitating new avenues of research for the distributed systems community, in both end-to-end tracing, and the further applications that it enables.

References

[1] C. C. Aggarwal and H. Wang. *Managing and mining graph data*, volume 40. Springer, 2010.

[2] Apache. Hadoop 2.0.x. http://hadoop.apache.org/, 2013. Accessed May 2013.

[3] Appneta traceview. `http://appneta.com`. July, 2013.

[4] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, volume 9, 2003.

[5] V. Barra and S. Biasotti. 3d shape retrieval using kernels on extended reeb graphs. *Pattern Recognition*, 2013.

[6] D. D. Bonchev and D. H. Rouvray. *Chemical graph theory: introduction and fundamentals*. Taylor & Francis, 1991.

[7] K. M. Borgwardt and H.-P. Kriegel. Shortest-path kernels on graphs. In *ICDM*, pages 8–pp. IEEE, 2005.

[8] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 1997.

[9] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. *SIGOPS OSR*, 2007.

[10] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[11] Cloudera htrace. `http://www.github.com/cloudera/htrace`.

[12] Compuware dynatrace purepath. `http://www.compuware.com`. Accessed July, 2013.

[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI*, 26(10):1367–1372, 2004.

[14] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.

[15] T. Hofmann, B. Schölkopf, and A. J. Smola. Kernel methods in machine learning. *The annals of statistics*, 2008.

[16] T. Horváth, T. Gärtner, and S. Wrobel. Cyclic pattern kernels for predictive graph mining. In *SIGKDD*. ACM, 2004.

[17] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite. In *ICDEW*, pages 41–51, 2010.

[18] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 2013.

[19] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *ICML*, volume 3, 2003.

[20] H. Kubinyi. Drug research: myths, hype and reality. *Nature Reviews Drug Discovery*, 2(8):665–668, 2003.

[21] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *SIGKDD*, 2006.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[23] P. Mahé and J.-P. Vert. Graph kernels based on tree patterns for molecules. *Machine learning*, 75(1):3–35, 2009.

[24] G. Mann, M. Sandler, D. Krushevskaja, S. Guha, and E. Even-Dar. Modeling the parallel execution of black-box services. *HotCloud*, 2011.

[25] A. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, Feb. 2012.

[26] K. Ostrowski, G. Mann, and M. Sandler. Diagnosing latency in multi-tier black-box services. In *LADIS*, 2011.

[27] J. Ramon and T. Gärtner. Expressivity versus efficiency of graph kernels. In *MGTS*, pages 65–74, 2003.

[28] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, volume 3, page 9, 2006.

[29] K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. In *IAPR*, 2007.

[30] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, pages 43–56. USENIX Association, 2011.

[31] B. Schölkopf and A. J. Smola. *Learning with kernels*. The MIT Press, 2002.

[32] R. Sharan and T. Ideker. Modeling cellular machinery through biological network comparison. *Nature biotechnology*, 24(4):427–433, 2006.

[33] N. Shervashidze and K. M. Borgwardt. Fast subtree kernels on graphs. In *NIPS*, 2009.

[34] N. Shervashidze, T. Petri, K. Mehlhorn, K. M. Borgwardt, and S. Viswanathan. Efficient graphlet kernels for large graph comparison. In *AISTATS*, pages 488–495, 2009.

[35] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. *JMLR*, 2011.

[36] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. *Google research*, 2010.

[37] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. Ganger. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS PER*. ACM, 2006.

[38] Twitter Zipkin. `https://github.com/twitter/zipkin`. Accessed July, 2013.

[39] S. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *JMLR*, 99:1201–1242, 2010.

[40] T. Washio and H. Motoda. State of the art of graph-based data mining. *ACM SIGKDD Explorations Newsletter*, 5(1), 2003.