Towards interactivity with DViz

Stewart Grant*
University of British Columbia

ABSTRACT

Understanding and debugging distributed systems is a difficult task. Without proper tools developers are forced to inspect logs from diverse machines. Tracing tools are used track distributed executions based on control flow, and are typically accompanied by a visualization front end for ease of use, and comprehension. Here we propose a visualization for state based tracing. Our prior work used t-SNE to visually cluster an execution. This approach took minutes on traces of over 100 trace points, and over 300 variables, a significant barrier to interactivity. In addition we used inferred data invariants as a characterization of a systems properties. Lists of such invariants totalled in the hundreds, and were incomprehensible to users. Here we propose algorithmic, and architectural techniques for improving the performance of t-SNE for the sake of interactivity, and a novel technique for automatically extracting interesting data invariants to reduce their cardinally, and increase comprehensibility.

Index Terms: K.6.1 [Distributed Systems]: Trace Visualization—Debugging Visualization;

1 INTRODUCTION AND MOTIVATION

The complexities of distributed systems have long plagued their developers. Writing code which executes on various machines is intrinsically more complicated due to networking eccentricities, such as failures, partitions, and message delays. Debugging and checking the correctness of such systems is laborious and technical, as developers must inspect large logs for small discrepancies in expected values, and timestamps. At scale auxiliary tools are necessary for interpreting logs and making them understandable. Typically tracing tools are used to reconstruct the communication of nodes throughout a system, order their events, and present developers with a comprehensive view of an execution. Tracing tools alone still produce large amounts of data, albeit their structure is more understandable than raw logs. Typically tracing tools are equipped with a visual front end, allowing users to quickly observe the behaviour of executions, and under scrutiny identify irregular or bugging behaviour. No one tracing technique is sufficient for debugging distributed systems. While most tracing tools are concerned with debugging they typically fall into 3 sub categories of performance tuning, distributed control flow, and model checking. Each of these tracing objectives have different flavors of visualization which pair with them. We overview these visualization techniques in Section 2

We propose a tracing tool which captures state similar to a model checking trace tools, with the exception that rather than logging control flow, our tracing tool only logs a distributed programs state. Such tracing is unconventional and does not fit nicely into the aforementioned tracing categories. As such our unique requirements demand innovative visualization solutions. Entirely state based program analysis has ties in the world of trajectory programming [18–20]. In trajectory analysis simple ML techniques such as weather man, and mean prediction, as well as linear, and logistic regression are used to automatically predict and parrallelize computation based solely on state analysis.

Our proposed visualization for traces of program state applies a similar ML technique, t-SNE clustering, a dimensionality reduction algorithm [11]. We leverage t-SNE to clusters points in a programs execution based on the similarities of a programs state. T-SNE requires a distancing function to cluster state, our distance function is as follows. The distance between two trace points p and p', whose state is composed of an identical set of variables with potentially different values. For each matching pair of variables XOR them together. Each 1 bit in the resultant XOR is a difference of 1 bit between the variables. We calculate the difference between two variables as the number of one bits in the XOR. The distance between trace points p and p' is the euclidean norm of all variable distances.

We found the results of our initial technique promising, and the high level behaviour of distributed programs we traced. However, our visualization suffers due to a high level of computational complexity in running t-SNE. A single step of the iterative t-SNE algorithm is $O(n)$, and the algorithm is typically executed for greater that 20 iterations before a reasonable clustering is obtained. Typical runtimes for traces consisting of 80 - 100 trace points, containing 80 - 100 variables each resulted in runtimes in the tens of minutes. This significant barrier to interactivity lead us to alter the architecture of our tool, and implement parallel t-SNE to achieve interactive sub 10s computation times.

Our prototype visualization allowed users to inspect trace points, and view individual variable values, without the ability to compare points. All variables were weighted equally when computing weights. Our interactivity goals are two fold first developers should be able to query two points, and inspect variables which caused them to be distant from one another. Second we acknowledge that all variables are not of equal importance in a program. Some variables values drastically alter the behaviour of a program while other do not. To this end we extended our interface to support re-clustering with increased weights for important user specified variables.

An additional feature of our tracing tool is the analysis of distributed data invariants. We infer invariants using Dinv, a distributed front end for Daikon [6]. These invariants are inferred over entire traces of an execution, and the number of invariants can be large and incomprehensible (300-400). In addition to our improvements of t-SNE we refined our invariant analysis by first, logically detecting t-SNE clusters using k-Means clustering, deriving per cluster invariants, and refining those further to unique invariants for each cluster. This processing step greatly reduces spurious, and uninteresting invariants, and profiles cluster behaviour more precisely.

The rest of the paper is laid out as follows. Section 2 covers related work. Section 3 overviews the scope of our contributions, and Section 4 covers our implementation. Section 5 details our performance results and data refinement accuracy. Finally Section 6 and Section 7 denote potential future work, and the conclusion of the paper.

2 RELATED WORK

Performance tracing tools such as [1, 9, 13, 16, 21], aim to decrease latency of distributed operations by maximizing concurrency, and minimizing time spent blocking. Performance visualizations often aligned parallel processes approximately, and color encode large

*e-mail:sgrant09@cs.ubc.ca

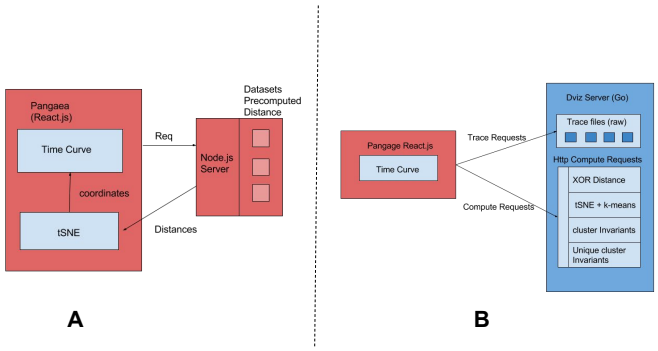

Figure 1: Alterations in Dviz architecture. A) Original architecture with single-threaded JavaScript computation. B) JavaScript front end with highly paralleled Go server managing compute requests.

latency's to help developers identify sub optimal segments of their executions.

Control flow tracing tools such as [2,5,15] help developers debug fine grain messaging protocols. Visualizations of such traces must often encode all messages, as developers must reason about all potential message orderings. A common encoding for distributed control flow is a message passing graph, in which time is encoded as a vertical line for each traced host, and messages are directed arrows connecting host lines.

Model checking trace tools like [3,4,17] help developers verify the correctness of their system by generating an abstract model of an execution. Such systems typically capture control flow, and state transitions of a trace, and compose a finite state machine (FSM) of a systems execution. Visualizations of such traces often encode the trace as a control flow points (such as the entrance of a function) connected with directed edges to other control flow points. These visuals can grow to an incomprehensible scale as the size of many models is exponential.

3 Scope

Our initial distributed trace time curve visualization suffered a number of shortcomings. Mainly the time to compute the time curve using t-SNE was too slow, on the order of 8-10 minutes for the traces we wished to analyze. Such a compute time is problematic for us because we wished to achieve interactive recomputation of the time curve by biasing clusters towards variables that users marked as important. The leading cause of our visualizations compute time was the single threaded JavaScript architecture we used to run t-SNE. Another shortcoming of our visualization was the lack of labels on clusters, and the size of invariants generated for a single trace. T-SNE generates visual clustering, but not logical ones, so no further computation can be done to the clusters. Daikon, our tool for inferring distributed invariants, outputs all of its template invariants which are not violated during an execution. This number can be large, and many invariants are spurious. In the following section we describe our architectural shift from JavaScript to Go, the implementation of an efficient parallel t-SNE, our back end support for skewing t-SNE clustering towards important variables. Further, we describe our algorithm for computing visual t-SNE cluster into logical ones, our approach for calculating cluster invariants, and their subset of unique cluster invariants.

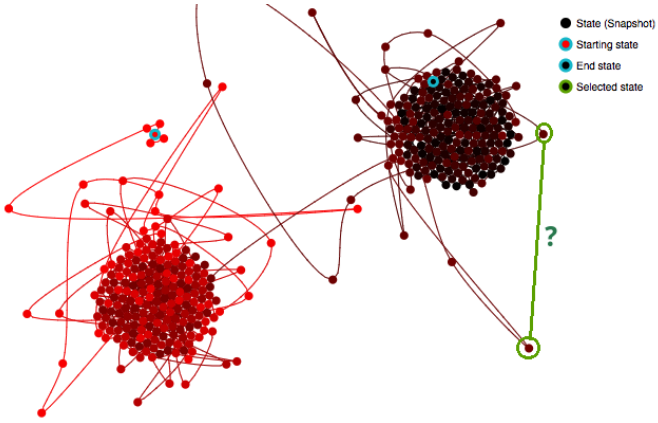

Figure 2: Time curve generated by our prototype running puts and gets to a key value store. Highlighted in green are two points which are both executing puts. However, their state distance is large, with no justification.

3.1 JavaScript to Go

JavaScript has many convenient frameworks for building client server applications which are compatible with nearly all browsers. This fact lead us to develop our time curve prototype in a mixture of React [8], and node.js [12]. Calculating XOR distance on our target traces was sufficiently slow with this framework that we cached results and served precomputed distance from our node.js server. Figure 1 outlines our original architecture. T-SNE coordinates were calculated client side, at latencies of 30s. To achieve our goal of interactivity we designed a new architecture with a thin client, which issues computation requests to an optimized server written in Go. Our choice of Go was due to its concurrency language primitives and increased performance. In section 4 we discuss the details of our implementation.

3.2 Parallel t-SNE

t-SNE is a ML algorithm for dimensionality reduction. We leverage t-SNE as the state space of a trace point can be modelled as a high dimensional vector where each variable is a dimension with a magnitude equal to it's binary encoding. At it's core each iteration of t-SNE has 4 steps. First a cost gradient is calculated, using a distance function, second the gradient is normalized, third the gradient moves all projected points a distance, last the momentum of each point is updated for the next step, and the points are re-projected. Each stage of this algorithm has a data dependency on its prior step, therefore there is no trivial method for parallelizing t-SNE. Our solution requires that separate threads are delegated points for which they must compute values, and a master thread which coordinates barriers to protected against inconsistent memory accesses.

XOR Distance: A single point in a distributed trace can consist of hundreds of variables. Computing the distance between any two points requires that we calculate XOR on each variable pair, and then calculate the euclidean norm of each. Our typical traces consist of 100-300 variables per trace, and 100 trace points. As t-SNE is $O(n^2)$ per iteration, requiring approximately 20 iterations for reasonable results we end up with a typical number of XOR computations in the order of ((300*300)/2)*100 * 20 100M XOR computations. We alleviate much of this complexity by precomputing XOR distance for each pairs of points and caching them. By doing so we only incur the full cost of running XOR distance for a single iteration of t-SNE.

3.3 variable weighting and distance reporting

Figure 2 is an example visualization generated by our original tool. At a glance the image is composed of 2 clusters, with no semantic meaning behind them. In fact this is a distributed key value store responding to a 50% put, and 50% get workload. Under the assumption that users would generate their own traces from test cases, we assume that they will have some understanding of the high level functional behaviour of their system. To help developers reason precisely about clusters we aimed to answer the question *Why is point A distant from point B?*.

3.4 Cluster Detection, and Invariants

The results of running t-SNE on high dimensional data are visual clusters of points. While these points are useful for discerning patterns in a trace, but as they are visual, they are not available for further analysis without an additional processing. We propose the use of k-means clustering on t-SNE output, to logically cluster visual clusters. Alternative density clustering techniques such as DBSCAN [7] have greater precision at detecting clusters automatically, our choice of k-means is to allow users to select the number of clusters they observe.

Logical clusters detected by k-means, can be further processed in isolation from other clusters. Our prototype tool output distributed data invariants which held over an entire execution. These invariants are course grain, and do not expose invariant behaviour of individual clusters. Daikon outputs all template invariants which are not violated by a trace, and which are supported by entries in a trace up to a minimal confidence measure. Applying daikon to individual clusters has the downside that the number of detected invariants grows, as there is less evidence to invalidate spurious invariants. We propose that invariants unique to clusters identify their most interesting behaviour. To detect unique invariants we compared each cluster with all others using Daikon's *Invariant Checker* tool. Unique invariants for a cluster, are invariants which are violated by all other clusters.

4 IMPLEMENTATION

4.1 Go Server

Section 3.1 describes the shortcomings of JavaScript as a fast computational engine, here we describe our migration to a server written in Go, along with it's advantages and disadvantages. Our initial Node.js server, output precomputed XOR distances, and states to a JavaScript client which computed t-SNE and plotted a time curve. They communicated requests via HTTP, in JSON format. Our Go server and JavaScript client communicate using the same protocol, with the exception that XOR distances are not precomputed. The client first requests raw, trace data, followed by a request to process the trace, and compute t-SNE clustering, and or invariant detection. Our decision to server trace data from the server is for the sake of extensibility. In the future trace data could be streamed to the server directly from an instrumented program, an operation which should bypass client functionality.

4.2 Parallel t-SNE

T-SNE was implemented in two stages using an architecture similar to in memory Map-Reduce. We use a master worker system wherein a master schedules treads equal to the number of available cores on a machine, and allocates them a range of work. In the case of XOR their work is a range of points to compute distances on. XOR distance computation is embarrassingly parallel so the master thread simply waits for all threads to complete before continuing to t-SNE iterations. The four data dependent computations of t-SNE are described in Section 3.2, in order to respect data dependencies we implemented parallel t-SNE as a 4 stage synchronous pipeline. Worker threads are allocated points identically to XOR, and are additionally provisioned with communication channels. Workers listen for commands on channels, and execute a single state in the 4 state pipeline upon request. When complete they single the leader. Our implementation is built off of [14] and required an additional 80 lines of Go to parrallelize.

4.3 Variable weighting

Reporting variable weights to our client is simplistic from an implementation perspective, but with a few subtle pitfalls when implemented practically. Variable differences are calculated during XOR computation, collecting variable difference for each variable required only a few lines of code change. On our modest size trace of 300 variable per trace point over 100 points, the size of the difference matrix was 75GB, requiring over 8min of transfer time on a local machine! Luckily the matrix is sparse, reporting only variable differences reduces the matrix to 2GB. Further much of data in the matrix is redundant (i.e. variable names) post tar.gz compression our matrices averaged 35MB.

4.4 Cluster Detection

To detect t-SNE clusters for further processing we leveraged an off the shelf k-means clustering library goxmeans [10]. Goxmeans generates multiple cluster models per data set. In all cases we select the clustering model with the highest bayesian information criterion, which matches the number of user selected clusters. In our current implementation, cluster# selection has not been integrated into our front end visualization, and specified via command line.

4.5 Daikon and Cluster Invariants

Logical clusters are fed as input into a Go frontend which executes Daikon as a separate process. Cluster invariants are inferred by partitioning original traces into sub traces which map to clusters, and executing daikon on them. Unique cluster invariants are computed by checking each clusters invariants against all others. Unique invariants, are invariants which are not present in any other cluster. Daikon does not have a Go API and thus the files in the Linux operating system are used to coordinate between the two processes.

5 RESULTS

To evaluate both the increase in performance gains from our architecture transfer we ran a performance evaluation. In addition to comparing our Go server to a JavaScript baseline we also compared the performance gains from running single v.s. multi threaded t-SNE. We close our evaluation with an accuracy measure of k-means clustering as a technique for logically grouping t-SNE clusters.

Experimental setup. We ran all our experiments on an Intel machine with a 4 core i5 CPU and 8GB of memory. The machine was running Ubuntu 14.04. and all applications were compiled using Go 1.6 for Linux/amd64. Experiments were run using a trace file containing 362 trace entries, with 80 variables per trace entry with a size of 1.5MB.

5.1 XOR distance and t-SNE

Calculating XOR distances is an embarrassingly parallel task. Figure **??** shows our scaling results taken from incrementing the number of threads used to compute the distance. Although the algorithm is easily parallelizable, a master thread must still aggregate the results generated by the working threads, and wait for the final stragglers (threads which are scheduled less frequently or run slower) to complete their execution. The average time achieved from running XOR distance on 4 cores was 1.75s.

Prior to refactoring running t-SNE for 20 iterations, each of which made a call to a single threaded XOR function required 507s or 8min on average. Post go migration, and XOR caching single threaded latencies for 20 iterations dropped to an average of 1.1s. To test the scalability of our mulit threaded t-SNE we incrementally added threads to additional cores on our test rig. The average time obtained

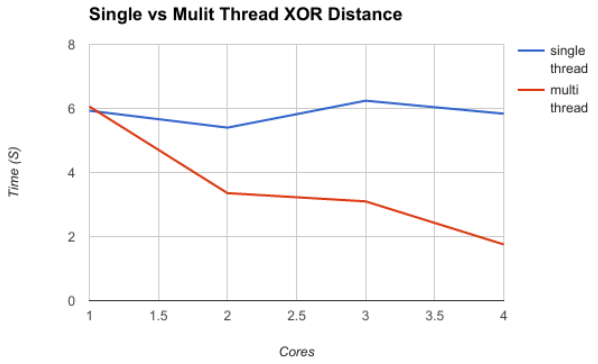

Figure 3: Single v.s. Multi threaded execution of XOR distance. Multi threaded gains slightly more than a 3x speedup running on 4 cores.

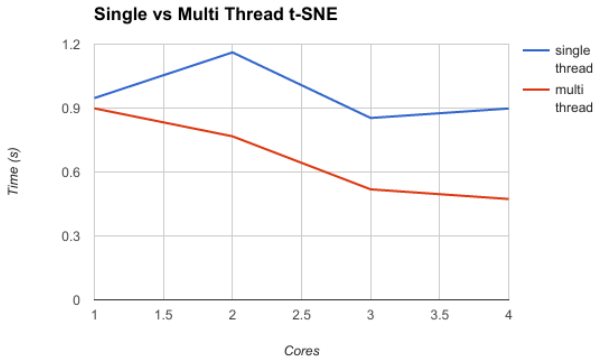

Figure 4: Single v.s. Multi threaded execution of t-SNE. Multi threaded gains a 2x speedup running on 4 cores.

from executing 20 iterations of multi threaded t-SNE on 4 cores was 0.47s.

5.2 Clustering

We evaluated our technique of generating logical clusters with k-means from t-SNE visual clusters by measuring it's accuracy on our sample trace. The sample trace was taken from a replicated key value store performing 3 large scale function, initialization where values are read from disk, followed by 50 puts, and 50 gets. Figure 5A show a simplified gnu plot rendering of the servers execution. Figure 5B Shows the results of running k-means on the visual clusters. Initialization is colored yellow, putting is black, and getting is coloured red. A few points which are visually clustered with both initialization and putting are logically clustered with getting. Using manual inspection we consider 6 of the trace points out of 362 to be classified correctly at an accuracy of 98.3%. We admit that non perfect classification will lead to incorrectness when inferring cluster unique invariants. We propose that in the future users be able to manually classify points, which are visible outliers.

6 Discussion and Future Work

limitations Our approach to visualizing distributed state is limited by computational power. Execution times of t-SNE grow polynomial with the length of a trace, therefore traces with lengths of tens of thousands would execute slowly regardless of parallelism. One heavy handed approach to gain scalability would be to spread out

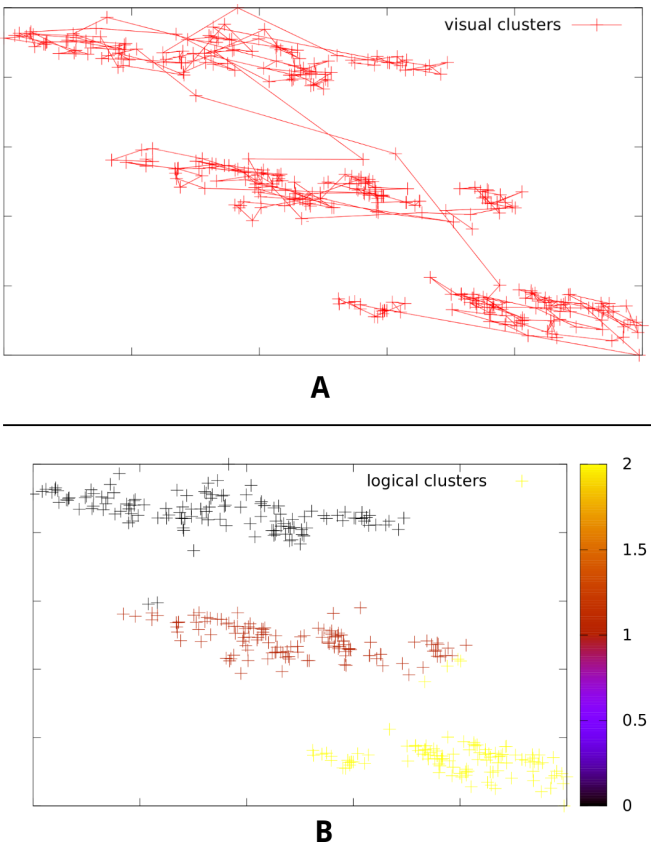

Figure 5: Clusters generated form a t-SNE clustered program trace. A) Show visual clusters only connected temporally. B) shows logical clustering achieved by running k-Means clustering, set for 3 clusters. Clustering accuracy 98.3%.

t-SNE computation on a cluster. However, the size of the trace would have to be in the tens of thousands to overcome the cost of synchronizing state.

Logged state has a limited view of a programs behaviour. All state collected with our tracing technique must be at the application layer and not hidden within binaries. The more state which is logged, the better our clustering technique responds. Therefore, we are limited to applications where the majority of functionality, and interesting behaviour is resident in users application code. One potential solution to this problem would be to analyze a programs stack and heap at the OS layer. While this would provide a more holistic view of a programs state during execution it sacrifices important information such as variable names, log lines, not to mention a state space explosion.

Allowing users to weight variables manually introduces the possibility of bias and error into our state model. Users with little experience of the programs they trace may be biased towards variables which are inconsequential, and could lead to their misunderstanding of a programs execution. This restricts our users to those which have a comprehensive knowledge of their software, and are in search of anomalies and bugs.

future work Clusters generated by t-SNE form a de facto state machine when connected temporally. Future work could extend our visualization to abstract clusters away, and present a state machine, where transitions between clusters were labeled with variables which caused them. The state of individual clusters could be simply encoded with their unique invariants. Our current visualization is

limited to a single execution. This approach may be useful for traces known to contain bugs, and abnormalities, but is arguably poor for checking subtle differences between multiple executions. Future work could cluster 2 or more executions together and compare their clustering transitions for similarities and differences.

7 Conclusion

Here we presented a our work parallelizing t-SNE for the sake of interactive clustering, and a proposed architecture for JavaScript applications requiring fast responses, and large scale computation. We improved the interactivity of our clustering visualization by allowing users to re weight variables based on importance, and by justifying point distances by reporting per variable distances. addition we presented our technique for reducing large scale invariant data in our traces by applying k-means to t-SNE output, and logically analyzing the unique invariant behaviour of clusters.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] J. Abrahamson, I. Beschastnikh, Y. Brun, and M. D. Ernst. Shedding light on distributed system executions. In *ICSE 2014, Proceedings of the 36th International Conference on Software Engineering*, pp. 598–599. Hyderabad, India, June 2014.

[3] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *In Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, pp. 85–90. USENIX Association, 2003.

[4] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 468–479. ACM, New York, NY, USA, 2014. doi: 10.1145/2568225.2568246

[5] L. K. Dillon, W. Visser, and L. Williams, eds. *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. ACM, 2016.

[6] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering pointer-based program invariants, 1999.

[7] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pp. 226–231. AAAI Press, 1996.

[8] Facebook. React: A javascript library for building user interfaces. `https://facebook.github.io/react/`, 2017.

[9] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA, 2007.

[10] B. Hancock. goxmeans: An implementation of the x-means algorithm in go. `https://github.com/sacado/tsne4go`, 2015.

[11] G. Hinton and Y. Bengio. Visualizing data using t-sne. In *Cost-sensitive Machine Learning for Information Retrieval 33*.

[12] Joyent. Node.js: Javascript runtime build on chrome's v8 javascript engine. `https://nodejs.org`, 2017.

[13] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996.

[14] sacado. tsne4go, an implementation of the tsne algorithm in go. `https://github.com/sacado/tsne4go`, 2015.

[15] L. M. Schnorr, G. Huard, and P. O. A. Navaux. Towards visualization scalability through time intervals and hierarchical organization of monitoring data. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 428–435, May 2009. doi: 10.1109/CCGRID.2009.19

[16] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[17] R. J. Walker, G. C. Murphy, B. Freeman-benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models, 1998.

[18] A. Waterland, E. Angelino, R. P. Adams, J. Appavoo, and M. Seltzer. Asc: Automatically scalable computation. *SIGARCH Comput. Archit. News*, 42(1):575–590, Feb. 2014. doi: 10.1145/2654822.2541985

[19] A. Waterland, E. Angelino, E. D. Cubuk, E. Kaxiras, R. P. Adams, J. Appavoo, and M. Seltzer. Computational caches. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pp. 8:1–8:7. ACM, New York, NY, USA, 2013. doi: 10.1145/2485732.2485749

[20] A. Waterland, J. Appavoo, and M. Seltzer. Parallelization by simulated tunneling. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*. USENIX, Berkeley, CA, 2012.

[21] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with jumpshot. *Int. J. High Perform. Comput. Appl.*, 13(3):277–288, Aug. 1999. doi: 10.1177/109434209901300310