More intervention now!

Moises Goldszmidt and Rebecca Isaacs
Microsoft Research

ABSTRACT

Techniques for characterizing performance and diagnosing problems typically endeavor to minimize perturbation by measurements and data collection. We are making a call to do exactly the opposite. In order to characterize the behavior of a system and to perform root-cause analysis and answer what-if questions, we need to conduct active and systematic experiments on our systems, perhaps at the same time these systems are running. We argue that in distributed computing frameworks such as MapReduce, Dryad and Hadoop, the conditions are right for automatically conducting these experiments. At each stage there is a large number of nodes doing the same computation, hence providing a sound statistical population. Furthermore, we have the infrastructure in such systems to isolate and recreate the conditions of a run. In this paper we propose the missing piece: a blueprint of the causal interactions that can be used to plan these experiments and perform inferences about the results. Machine learning and statistical analysis give us the tools and algorithms for inducing such a causal blueprint from a combination of passive observations and active intervention.

1. INTRODUCTION

Performance debugging of distributed software systems on large clusters is notoriously challenging. Slowdowns are often caused by resource contention between unrelated applications, and the root cause of the problem can be transient. Unanticipated skew in the amount of computation by different instances of the same task, can be difficult to track down. The same factors mean that running time prediction is something of a black art and as a result the utility of capacity planners and performance-aware schedulers is limited.

We feel that the time is ripe to expand our arsenal of tools for performance analysis in the context of distributed computing frameworks like Dryad and MapReduce. We propose *active intervention*, where the test subject—a single task, such as a mapper, a reducer, or a Dryad vertex—is probed in the same environment as its live version in a controlled manner. Probing, by rerunning the task in-situ and fixing some variables while altering others, enables actionable root cause analysis of performance problems. The experiment is constructed around a *causal model*, which acts as a *blueprint* for the task by describing its resource demands, how they vary over time, the interactions between different instances of the task, and the dependencies between all of these.

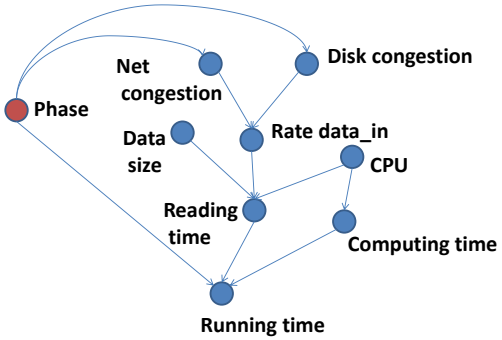

Figure 1: Part of the causal model for the running time of a task. This subgraph shows the factors that affect the time taken to read input. Other subgraphs, not shown, describe computing and writing output.

An example causal model for a Dryad vertex is shown in Figure 1. Intuitively, the graph represents that the running time of a task is partly determined by how quickly it can read its input data. The time this takes depends on disk bandwidth and contention. If the data resides on a remote machine, then network speed, latency and any congestion also have an impact. Some types of task compute over the input records while reading them, in which case the number and speed of the CPUs plays a part.

Using a causal model to describe task executions in this way is analogous to a blueprint for an electrical circuit, such as the one shown in Figure 2. The blueprint describes the causal relationships for the circuit, and hence plays two roles:

1. Given the state of one part of the circuit, it enables

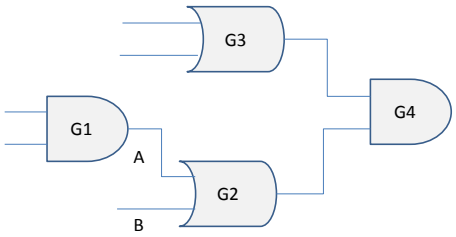

Figure 2: Blueprint of a digital circuit. The blueprint has two functions: a) to describe a set of causal assumptions and b) to provide an oracle for intervention.

one to deduce the state of some other part of the circuit (i.e. the *cause* of an observed output value, or the *effect* of an observed input value).

2. Following *intervention* to set any of the inputs, it enables one to infer the values of causally dependent inputs and outputs.

The first capability is used for *verifying* that the circuit is working properly, while the second is useful for *debugging* and finding the root cause of the failure (within the confines of the circuit), including the malfunctioning of a gate. So, for example, if we observe that inputs A and B to gate G2 are both 0, the blueprint enables us to infer, using algebra, that the output of G4 will also be 0. Moreover, we can also conclude that at least one of the inputs to G1 was also 0. Suppose now that we explicitly set the value of input A to 1 by physically injecting the appropriate voltage into the circuit. The blueprint tells us that the value at the output of gate G2 will now be 1, and we cannot make any inferences on the value of the inputs to gate G1 (since the value of A is no longer a consequence of these inputs).

Similarly, the causal model of a Dryad or MapReduce task allows us to systematically analyse and address performance problems. Given a model such as that in Figure 1, we can actively intervene by varying any of the accessible variables (from the congestion at the disk, to the CPU utilization and the rate of data-in) while re-running the task in a sandbox.

Active intervention contrasts with standard techniques for performance analysis, which typically monitor behavior *passively* to avoid perturbing the measurements and/or adversely affecting the performance of the system itself. Our approach makes use of passive measurements, in particular to construct the causal model, but augments current practices with a powerful, experiment-based approach. This not only helps to find the root-cause of a performance problem, but also identifies what adjustments will rectify the problem, with a guarantee that the changes will have the desired effect.

We are developing our ideas in the context of data-parallel distributed computing systems, e.g. Dryad or MapReduce. In such systems, applications are expressed as a dataflow graph that is scheduled at runtime onto the physical cluster. The input data is partitioned across machines to enable many identical *tasks* (e.g. Dryad vertex or MapReduce mapper or reducer) to execute in parallel. A large cluster will host many jobs at the same time, with consequent sharing of physical resources and potential for contention between unrelated programs[1].

The rest of this paper discusses Dryad vertices specifically, although the techniques are readily applicable to a MapReduce environment as well. At this exploratory stage we are particularly focused on DryadLINQ programs[17], in which the majority of vertices correspond to a well-defined set of operators, thereby simplifying the challenge of constructing causal models.

2. CAUSAL MODELS

Research into causality discovery and intervention has been revitalized recently, given the abundance of data in such fields as artificial intelligence, statistics, econometrics and epidemiology. In 1991 Pearl and Verma proved the limits for which causal relations can be inferred from passive observation alone[13]. To extend these results, formalisms were later proposed that develop a mathematical language to model causal relations, interventions, and the computational procedures to make inferences[11, 12]. Our modeling technique is inspired by this work.

The formal representation of the model is based on Bayesian networks, which are very efficient data structures for the representation of, and reasoning with, probabilistic and statistical information. In fact, there are many parallels between a circuit blueprint and a Bayesian network. A Bayesian network is a graph where the nodes represent the random variables and the relationship between a child and its parents is described by a probabilistic model. You can think of a Bayesian network as a blueprint where the inputs and outputs may be real continuous numbers, and the functional specifications of the gates are probabilistic[1].

Moreover, similar to a blueprint, a Bayesian network (and our causal models) are modular. As the blueprint represents a function by the composition of its gates, a Bayesian network represents a complete probability distribution given by the product of the relationships between the children and its parents in the graph. Thus, given a domain with random variables $X_1, \ldots, X_n$, we formally have that:

$$P(X_1, \ldots, X_n) = \prod_i P(X_i | Pa(X_i)) \qquad (1)$$

where $Pa(X_i)$ denotes the parents of X_i in the graph. These probabilistic relationships may be a linear regres-

[1]In a Bayesian network the connectivity in the graph has a formal semantics based on conditional independence[8].

sion between the inputs X_i and the output $y = \sum_i a_i X_i + \epsilon$, where ϵ is a zero mean Gaussian noise, or a mixture of Gaussian models, or any of the various non-parametric models. In our domain, linear models or generalized linear models will be sufficient to a first approximation.

An additional challenge for us is that unlike in a typical circuit blueprint, the functional specification of the causal model is not fully known. Fortunately, the tasks of a data-driven parallel program have inherent structure (as we will discuss in the following section), and as a result, we have a template for the "gates" and their composition as per Equation 1 in the causal model. The parameters of these gates can then be statistically fitted from traces of the task's resource demands, drawing on a well developed statistical theory of inducing Bayesian networks from data, some of which has been already applied to software systems[4].

Representing the execution behavior of a task in this way is analogous to the first role of the blueprint, described in the previous section. Since it is based on augmenting the representational capabilities provided by Bayesian networks and graphical models, our approach inherits the same capabilities. Prior art on representing and reasoning with uncertainty, and on automatically fitting and learning models from data, can be applied. This includes dynamic models and models with variables that are not directly observable (such as the *Phase* variable discussed in Section 4). Crucially, inference for the state of some variables given observations of other variables is performed by known algorithms and with guarantees of soundness.

The second role of the blueprint, enabling inference given active intervention, was formalized relatively recently by Judea Pearl[12]. The operations are similar to those we perform on a circuit blueprint when we apply voltage, and involve a transformation of the Bayesian network to one in which the algorithms for inference from observation will work. The approach provides a simple way to identify the controlling variables and model the immediate impact of intervention (or perturbation) on the variables of interest, in a single formalism.

3. APPLYING THE MODELS

There are many interesting characteristics of Bayesian networks, explained and exploited in numerous papers and real applications. The main one that concerns us here is that, much as in the electrical circuit blueprint, we can specify and learn from data each one of the relationships between inputs and outputs (parent and children in the graph) modularly and independently following the same composition as in Equation 1.

Thus, for example, we can specify that in the graph of Figure 1 "Reading time" (rt) is related to its inputs "Data size" (s), "Rate data_in" (r), "CPU" (c) as

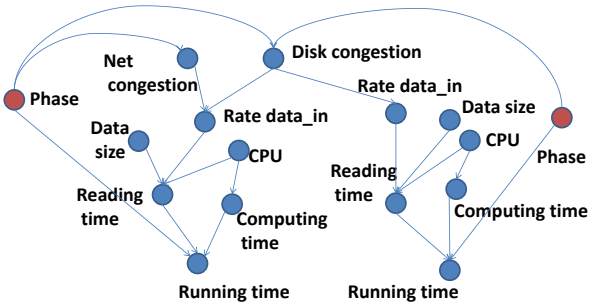

Figure 3: Adding another vertex with a local disk.

$$rt = s * r + s * c * A + \epsilon \quad (2)$$

Where A (one of the parameters) is the amount of compute demanded by the vertex while it is reading, and ϵ is a zero-mean Gaussian noise, whose variance σ is the other parameter. The read and compute rates are expressed per byte of input, and the result will be reading time in seconds. Note that the parameters A, ϵ and σ are fitted with standard linear regression techniques and the probabilistic relation between rt and its parent is a Gaussian with mean equal to $s * r + s * c * A$ and variance σ. More generally, such equations can be thought of as the expected "operational laws" for the performance of these systems.

Once we have defined the graphical structure and the operational laws for the model, an execution of the task in its parallel distributed environment, typically involving many replicas, will provide enough data to fit the parameters using standard regression. We anticipate that this can be done online with an explicit experiment: most of the platforms addressed in this paper already provide infrastructure to collect statistics from each node in the cluster, including CPU utilization, virtual memory, and bytes read and written (e.g. the Artemis monitoring system for Dryad[5]). Note that the uncertainty expressed in the parameter ϵ represents errors in measurement and errors due to modeling—we are not modeling all the possible inputs to each gate, and the time granularity may imply some degree of approximation.

The causal model can now drive active intervention as an experimental process in which values of causally relevant nodes are systematically varied to observe the effects on the output of interest. This is conceptually similar to physically injecting a voltage so as to fix an input in an electrical circuit. We envisage re-executing a vertex in a sandbox, but in-situ on the cluster[2], supported by the ability to artificially induce the desired values for

[2]Re-execution is straightforward using Dryad's built-in fault tolerance mechanisms.

the inputs. We will then be able to perform the same inferences as in a regular Bayesian network with the fixed variable set to the appropriate value, and use the results for "what-if" questions, root-cause diagnosis, and model refinement.

As an example of active intervention, consider the two nodes displayed in Figure 3. One has a local disk, which is shared with the other node that accesses the disk remotely. Assume that we want to find out the running time of the remote node if both are reading from the disk at the same time, under various modes of network congestion. The model would guide the set up of the simulation. The vertex would be sandboxed and its "Disk congestion" and other state variables set appropriately. We then re-execute the vertex, simulating various degrees of network congestion, and measure the dependent variables. This not only lets us estimate running time under different conditions, but also lets us gather the data necessary to refine the models for further online experiments. In this case, the resulting information might be effectively used by the scheduler or by the programmer to change allocation, data partition, or even the algorithm.

4. TASK PHASES

The phase of a vertex, which is an implicit variable in our causal models, arises from its pattern of reading input, computing over it, and writing output. As a vertex transitions from one phase to another, the bottleneck resource changes and hence the factor that determines the rate of progress. For instance, while in a computing phase a vertex might be CPU or memory-bound, but while writing its output the bottleneck becomes the rate at which the vertex can write to its local disk. By incorporating the phase, the causal model reflects the fact that contention on a resource will have different impact depending on which phase it occurs in.

Consider the `select` operator, which is the DryadLINQ equivalent of Map in MapReduce. `select` takes a stream of input records, applies the the same function to each record, and writes the output. The top graph of Figure 4 shows the cumulative data read and written against time (in seconds) for a `select` vertex processing 1GB of data. The middle plot shows the cumulative CPU seconds consumed by the process, and the bottom the average number of runnable threads, which gives an indication of the vertex's potential for concurrent execution. Vertical lines show the phase boundaries, which in this case are very clear transitions and provide evidence that these phases are readily identifiable.

The phase variable is not directly observable and so must be inferred from the rest of the variables in the model. We address this problem by noting that the dynamics of these phases are naturally expressed with a Hidden Markov Model (HMM). The phase is the only hidden state, while visible behavior such as the CPU utilization, memory consumption, and rate of the data flowing in/out provide the evidence for state transition. This is a relatively low dimensional problem[3] for which there are standard techniques such as Expectation Maximization [14].

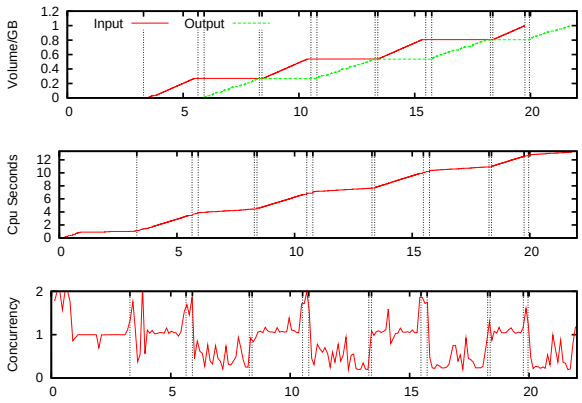

Figure 4: Cumulative resource usage vs time by a `select` vertex. The upper graph shows I/O GB, the middle CPU seconds and the bottom the degree of concurrency. The inferred phase boundaries are marked as dotted vertical lines and delimit distinct behavioral modes.

5. RELATED WORK

MapReduce and Dryad are popular platforms and there is a corresponding degree of interest in enhancing the performance analysis capabilities for them. The typical debugging approach uses a combination of online monitoring and after-the-fact log analysis[5], while schedulers have been proposed that attempt to balance fairness with locality[6, 18]. Efforts to multiplex tasks from different jobs on the same physical computer, such as Hadoop On Demand, highlight the need for performance-aware scheduling. Mantri[1] mitigates the effects of outliers in a MapReduce cluster by using passive observations to decide whether the underlying cause is likely to be the network, the machine or skewed data partitions. All these scenarios would benefit from the performance models that we can obtain using active intervention.

The mechanisms behind our approach have been used in other systems: the idea of controlled perturbation was tried in Triage[15], which is a system for diagnosing failures in server applications using sandboxed reexecution at the end-user's site. The 'Look Who's Talking' framework infers dependencies between VMs on a cluster using just the VMs' CPU usage[2]. It perturbs some randomly chosen VMs by capping their CPU utilization in order to check and refine the inferred clusters. The

[3]Relative to say, speech recognition, which is another problem solved by HMMs.

WebProphet system deliberately alters the timing of object load times in web pages to predict the impact of optimizations[9].

"Phases" of dominant resource activity are exploited in Ganesha[10], a fault-diagnosis tool for Hadoop that uses Expectation Maximisation to learn performance counter distributions and hence pinpoint the source node of a poorly performing MapReduce job.

There is a considerable amount of work on either building "observers" for a control loop or exciting the system with specific inputs to (automatically) build a model of the response with a control objective in mind[3]. As we explained earlier, our objective (performance analysis) and methods (representing a task's behavior with a causal model) are very different. Finally, our work shares some similarity with the work on the discovery of likely invariants described in [7]. The main differences are that we assume a lot more structure in our models, in particular a parameterized functional schema for the performance dependencies in the vertices, and we actively intervene on the system in order to gather specific data to properly fit the parameters in these functions.

6. CONCLUSION

Our belief that systematic active intervention for data-parallel systems is now feasible stems from three observations. Firstly, the mathematical approach required for causal modeling techniques has recently been developed[12]. Secondly, the applications of MapReduce and Dryad are highly structured, and typically run on largely homogeneous clusters. Thus the tasks in these jobs have a known set of external factors that affect the computation, and it is feasible to systematically vary them. Also, one run provides a statistically rich sample population as the same code is executed many times. Finally, the large-scale cluster environment makes it practical to probe a task on the same hardware as its original instantiation and hence recreate the original execution environment with a much higher degree of fidelity.

The approach we have described complements the prevalent passive monitoring methodology that has been developed in the past two decades, while enabling answers to what-if and related questions that require intervention in the form of active experimentation. We rely on passive monitoring to collect the data necessary for (a) fitting the parameters of the models and (b) identifying the cases that need further experimentation in order to find the root cause of the problems observed. The fitting of the parameters is done by using standard techniques from statistics such as regression and Expectation Maximization[14, 16]. To both direct the interventions and extract quantitative conclusions from the experiments, we rely on the algebra proposed in Pearl[12], which enables the rewriting of each intervention (the setting of a variable in the model) into a transformation on a Bayesian network. Once this is obtained we simply execute the well known algorithms on the Bayesian network for inference.

By taking into account the impact of different computational phases and different levels of congestion and resource utilization, we hope to be able to provide concrete answers to what-if questions about the performance and running time of vertices. We contend that only through this approach can we arrive at accurate information for performance analysis and effective resource utilization in large clusters where confounding factors escape characterization through passive monitoring alone.

7. REFERENCES

[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *9th USENIX OSDI*, Oct. 2010.

[2] R. Apte, L. Hu, K. Schwan, and A. Ghosh. Look Who's Talking: Discovering dependencies between virtual machines using CPU utilization. In *2nd USENIX HotCloud*, June 2010.

[3] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I.Jordan, and D. A.Patterson. Automatic exploration of datacenter performance regimes. In *1st ACM ACDC*, 2009.

[4] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *6th USENIX OSDI*, Dec. 2004.

[5] G. F. Creţu-Ciocârlie, M. Budiu, and M. Goldszmidt. Hunting for problems with Artemis. In *1st USENIX WASL*, Dec. 2008.

[6] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *22nd ACM SOSP*, Oct. 2009.

[7] G. Jiang, H. Chen, and K. Yoshihira. Discovering likely invariants of distributed transaction systems for autonomic system management. In *3rd IEEE ICAC*, June 2006.

[8] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[9] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: Automating performance prediction for web services. In *7th USENIX NSDI*, Apr. 2010.

[10] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-box diagnosis of MapReduce systems. *ACM SIGMETRICS Perf. Eval. Review*, 37:8–18, Dec. 2009.

[11] J. Pearl. Causal diagrams for empirical research. *Biometrika*, 82(4):669–688, 1995.

[12] J. Pearl. *Causality: Models, reasoning, and inference*. Cambridge University Press, 2000.

[13] J. Pearl and T. Verma. A theory of inferred causation. In *Principles of knowledge representation and reasoning*, 1991.

[14] L. B. Rabiner and H. L. Huang. An introduction to hidden Markov models. *IEEE ASSP Magazine*, 3(1):4–16, 1986.

[15] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *21st ACM SOSP*, pages 131–144, Oct. 2007.

[16] L. Wasserman. *All of statistics*. Springer, 2004.

[17] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX OSDI*, Dec. 2008.

[18] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys Conference*, Apr. 2010.