Exploiting Causality to Engineer Elastic Distributed Software

K. R. Jayaram[1]

1 **IBM Thomas J. Watson Research Center**
Yorktown Heights, NY, USA
jayaramkr@us.ibm.com

Abstract

Engineering robust elasticity management systems is vital to modern "born-on-the-cloud" distributed applications, as well as legacy distributed applications when they are migrated to the cloud. While the software engineering process for elasticity management is reasonably well-understood, the mechanisms used are often ad-hoc and not rooted in application semantics or interaction between application components.

This goal of this paper is to anchor elasticity in terms of causality in distributed applications. Assuming a large-scale distributed application architected as a set of interacting components, we motivate the need for (1) analyzing *causality* between interactions, and (2) estimating *casual probability* that an increase in the frequency of interaction i_1 can increase the frequency of interaction i_2 caused by i_1. We present algorithms to estimate causality and causal probability by combining well known sampling, program analysis, path profiling and dynamic slicing algorithms. We apply our algorithms for causal probability to three *generic* distributed applications, to evaluate (a) their effectiveness in the timely provisioning and de-provisioning of compute resources and (b) whether causality and causal probability present a fundamental and widely-applicable way of thinking about and engineering auto-elasticity.

1 Introduction

Elasticity is the ability of a distributed application to *self-adapt* and *dynamically* increase/decrease its use of computing resources to *maintain* its performance in response to varying workloads. The ability to provision and utilize *precisely* the amount of computing resources required by a workload at any point in time has long been the allure of cloud computing systems. Good *horizontal* scalability is a pre-requisite for elastic deployment – the application must increase its performance (e.g. higher throughput, lower latency) as it gets additional resources, and vice-versa. During the software engineering lifecycle, once an application developer establishes horizontal scalability of his design/algorithm/software, the next key challenge is the design and implementation of effective elasticity management components.

Explicitly managing elasticity is *hard* and error-prone for generic distributed applications, as opposed to specialized applications like distributed stream processing. It involves engineering elasticity management software components to (S1) monitor the application's performance during execution, (S2) determine when additional computing resources should be requested from the runtime/cloud management system, (S3) provision relevant components on the new resources while re-distributing the workload, and (S4) continuously monitor the application at runtime and relinquish un-necessary resources when the workload decreases. While the software engineering process in S1–S4 is well understood, the mechanisms employed today have become increasingly *ad-hoc*. One predominant practice is to use DevOps (development and operations) tools provided by IaaS clouds to automate S1–S4; examples are Amazon CloudWatch [3] and AutoScaling [6] for Amazon EC2 [2] and Ceilometer [37] for OpenStack [13] based clouds. DevOps tools typically adopt a black-box approach to elasticity

by using simple externally observable metrics, e.g., network traffic, CPU/memory usage, etc. to decide on elastic scaling at the level of virtual machines – an example CloudWatch rule may be to increase the number of VM instances by one when the average CPU utilization of the application exceeds 75%. Another approach, rooted in machine learning techniques, is to build statistical workload models based on the externally observable metrics reported by DevOps tools, using e.g., Linear Regression [47], Kalman filters [15] and wavelets [35]. Such statistical models have been shown to be effective for *some* workload classes. The main reason why said elasticity management techniques have limited efficacy is that they are *not rooted in application semantics*.

The broad goal of this paper is to examine whether *causality* in distributed applications can serve as a foundation for engineering elasticity. Most distributed applications deployed in consumer-facing settings at scale are architected as a set of interacting components; where each component (e.g., key-value store) is distributed over multiple physical hosts/virtual machines/containers; and components are layered into multiple-tiers (e.g. website, database, co-ordination, inventory, etc.). An external customer request received by a component $C1$ (e.g., web front-end) may trigger messages to components $C2$ *and* $C3$, which process them and send messages to more components. Hence, there is a natural *causal* relationship between incoming and outgoing messages at the component-level. A customer request from outside the application induces a so-called *causal path* among the components, before a response is sent. A distributed application typically has many causal paths, corresponding to different types of messages. When it is deployed in customer-facing scenarios at scale, handling millions of requests, depending on its current workload, causal paths may be exercised at varying frequencies, i.e., there may be *hot* causal paths.

The core contribution of this paper is in demonstrating that identifying such paths enables us to do *selective* elastic scaling of (parts of) components along hot causal paths; selective elastic scaling is more precise and effective than using externally observable metrics to scale all components e.g., in response to increased network traffic. We formalize the use of hot causal paths through the metric of *causal probability*, i.e., the probability that an increase in the frequency of requests to a component A increases the frequency of requests to component B. Causal probability is then used to increase the resources allocated to B.

While causality tracing in distributed systems has been studied extensively [39], [48], [54], [55], [54], etc., in our experience, these techniques have two main drawbacks. First, they focus on dynamic temporal causality, i.e., a message m_1 is said to cause m_2 if it temporally precedes m_2. Temporal precedence is determined either by assuming synchronized clocks, logging and matching log entries; or by using logical timestamps like vector clocks [51] and their variants [54, 48]. We have observed that temporal causality is often ineffective for elasticity management, because not every message m in a distributed application is caused by every message m' temporally preceding it. Second, they require extensive manual intervention and detailed knowledge of the application to identify *spans* [39, 49, 56] by pairing "requests" to and "responses" from every single component, annotate messages and RPCs, handle unique identification of requests in the presence of concurrency, etc. This makes it more challenging to use temporal causality tracing systems in practice.

This paper aims to alleviate the above problems by making the following technical contributions:

1. An new and intuitive way to reason about elasticity using causality (Section 2), along with analytical and empirical evidence (Section 5) of the ineffectiveness of using externally observable coarse-grained metrics.
2. A notion of direct causality, *stronger than temporal causality*, where a message M_1 to

component $C1$ is said to cause M_2 *only* when there is a direct dynamic control and/or data flow between M_1 and M_2. Direct causality is not limited to RPCs, applicable to message-passing systems and is not dependent on programmer defined spans. (Section 3).

3. Direct Causality Analysis (DCA) – a simple, easy-to-use program analysis and transformation technique that builds on *existing* program slicing and profiling techniques and to identify component boundaries, (direct) causal paths in distributed applications, and profile these causal paths at runtime, i.e., calculate the number of times every causal path is executed during the profiling timeframe. DCA only requires the application to be re-compiled, and does not require manual annotations, definition of spans, or any code changes. The paper also empirically evaluates the runtime overhead of DCA. (Section 4.2)
4. A mechanism to use causal path profiles to determine how requests to each component are partitioned among the causal paths originating at that component, and use this partitioning information to determine *causal probability.*(Section 4.4); and a mechanism to use causal probability for elasticity.
5. Empirical evaluation using three real-world applications from different domains and elasticity metrics to demonstrate the efficacy of DCA and causal probability for elasticity management. (Section 5).

2 Causality and Elasticity

In this section, we illustrate, through real-world scenarios why causality is a natural and intuitive way to think about elasticity in distributed applications. The three illustrations here describe (i) uneven workload distributions to *portions* of a component, (ii) distributed program paths among components executed at varying frequencies, and (iii) subtle mutual-exclusion issues.

2.1 Assumption : Application Architecture

Most generic distributed applications deployed in consumer-facing settings *at scale* are architected as a set of interacting software modules; potentially developed by different teams; layered into multiple-tiers (e.g. front-end/web-server, database, co-ordination, inventory, etc.), and could span thousands of machines across multiple physical datacenters. One example of this architecture is the famous "universal search" [17, 39] model of Google web-search, *parts of which* have been adopted *by* open-source projects for *enterprise-scale* deployments e.g., Apache Solr [4]. Figure 1 illustrates universal search [17, 39] – A web-server/front-end service receives the search query ($S1$) and distributes it to many hundreds of query servers ($S2$), each searching within its own partition/shard of the web index. The query is also sent to a number of other sub-systems that process advertisements, check spelling, or look for specialized results, including images, videos, news, and so on ($S3$ and $S4$). Results from all of these services ($S5, S6, S7$) are then aggregated by a separate service, and ranked according to relevance and/or potential for advertising revenue ($S8$) to generate results page ($S9$). In total, tens of machines and many different software modules (deployed in the datacenter as services) are needed to process one universal search query. A component may also contain nodes which are replicas of other nodes in the component (for fault tolerance). Each node in a component executes on a separate physical host/virtual machine/container inside the datacenter.

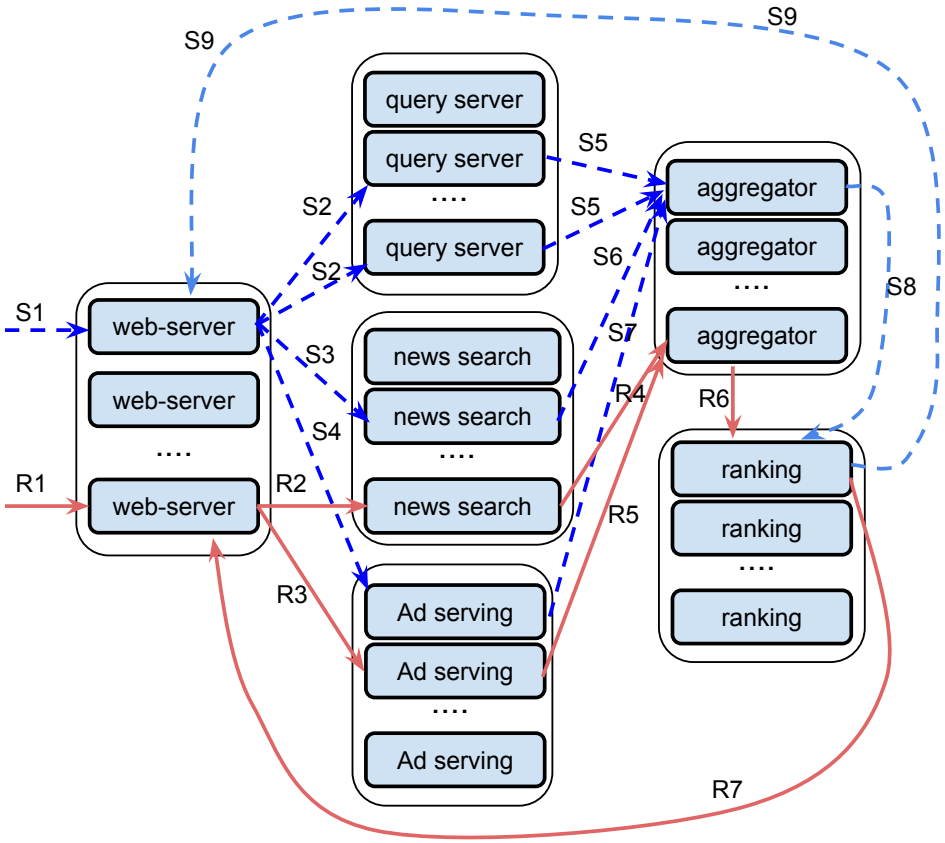


Figure 1 Causal paths in Universal Search [17, 39].

2.2 Uneven Workload Distributions

Workload spikes in applications like Universal Search, illustrated in Figure 1, are seldom uniformly distributed over all search terms. In response to, e.g., hurricane, election, new product like Apple Watch, there are spikes in specific search terms. This, in turn, causes workload spikes on specific portions of the each component corresponding e.g., the shards of the query server/web index component containing these search terms. Monitoring external bandwidth utilization to the web front-end and the query components may indicate that the whole application and *every* constituent component needs to be elastically scaled. But, this is not effective, because of the way each of the components is partitioned; and it will lead to under-utilization of resources added to some of the components because these resources are not going where they are needed most.

Causality, however, offers a more precise way to visualize such workload spikes. In Figure 1, denoting causation by $\longrightarrow$, $S1 \longrightarrow \{S2, S3, S4\}$, $S2 \longrightarrow S5$, $S3 \longrightarrow S6$,$S4 \longrightarrow S7$,$\{S5, S6, S7\} \longrightarrow S8$, and $S8 \longrightarrow S9$. $S1$ thus induces a causal path in the distributed application (illustrated by dashed blue arrows in Figure 1), with each component processing an incoming message and sending outgoing messages to other components. Using program analysis and transformation for identifying various causal paths in the universal search application and measuring their *dynamic frequency*, i.e., the number of times various causal paths are exercised at runtime, will *precisely* identify how an increase in the application

workload (external queries) causes the workload of various nodes to increase. That is, as shown in Figure 1, there is another causal path corresponding to "news search" - $R1 \ldots R7$ in the application, illustrated by straight red arrows. If runtime profiling indicates that 50% of traffic flows along $S1 \ldots S9$ and 5% along $R1 \ldots R7$, then the software modules along $S1 \ldots S9$ have to be allotted *proportionally* higher compute resources during elastic scaling.

2.3 Conditional Flows [27]

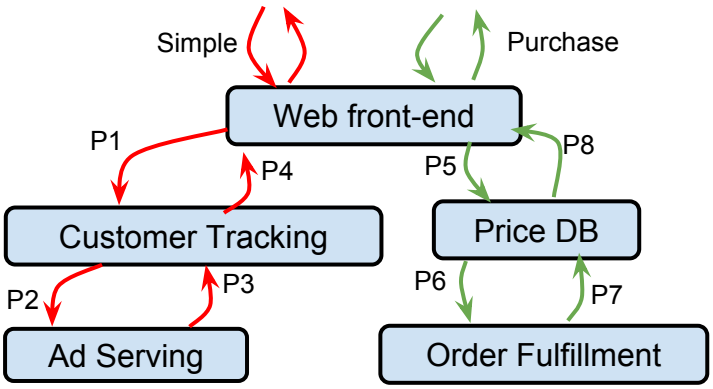


Figure 2 Causal paths and Conditional flows in an E-Commerce application.

The search application previous example illustrated uneven workload distribution *within* a component. The next example illustrates uneven workload distribution *across* components due to *conditional flows*. Consider a multi-tiered electronic commerce application like an online store illustrated in Figure 2. It is typically split into a web-server front-end, inventory management system, pricing database, customer tracking and ad serving components. A simple visit to the store (from a browser) will impact the web front-end, customer tracking and ad serving components. However, a purchase will impact the pricing database and inventory management system. Hence, depending on the context of the visit, two *different* (code) paths are exercised. There is a causal relationship between *Simple*, and $P1, P2, P3, P4$, i.e. $Simple \longrightarrow P1 \longrightarrow P2 \longrightarrow P3 \longrightarrow P4$. Also, $Purchase \longrightarrow P5 \longrightarrow P6 \longrightarrow P7 \longrightarrow P8$. Simply observing the network traffic to the web server is not sufficient to make a good elastic scaling decision, e.g., during Thanksgiving or Christmas sales, the *Purchase* path may be exercised heavily. And components serving that path should be scaled *proportionally more* to increase revenues without worrying much about customer tracking or ad serving. In general, for multi-tier applications with distinct causal paths, engineering elasticity management components requires an understanding of causal paths, path profiling and path frequency estimation along with using internal application logic to make effective decisions. This example focusses on conditional flows at the component-level; identifying causal paths will also enable node-level elasticity in this application when there are "hot items" (popular phones) as illustrated in the previous Search example.

2.4 Concurrency Issues

The third application scenario where causality can lead to effective elastic scaling decisions is when applications have subtle concurrency and mutual exclusion issues. The best example is the class of *datacenter infrastructure applications* like key-value stores (e.g., memcached [33],

Hyperdex [12]), consensus protocols (e.g., Paxos [25]), distributed lock managers (e.g., Chubby [8]) and message queues. A distributed key value store, for example, may have high CPU utilization when there is high contention to update a certain set of "hot keys". Relying on CPU utilization to add additional compute nodes will only degrade its performance further. In the case of a distributed cache like memcached [33], different elements of the cache may have varying access frequencies, and need different levels of elasticity. Using causality, however, will identify numerous causal paths leading *into* the component where there is lock contention, but only a few paths *going out* of said component; and elastic scaling of said component can be prevented because it is unlikely to change application performance.

3 Direct vs. "Happens-Before" Causality

Temporal causality (or "Happens-Before") is the backbone of *causal order broadcast* in distributed systems [51, 50]. Causal order broadcast was primarily concerned with the "causally safe" delivery of messages between processes in a distributed application, which were treated as black-boxes. Denoting temporal causality between messages m_i by $\rightsquigarrow$, $m_1 \rightsquigarrow m_2$ iff m_1 temporally precedes m_2. In the absence of synchronized clocks in large-scale distributed applications, temporal causality is typically detected using logical clocks – initially using Lamport Clocks [51], and currently, predominantly using Vector Clocks [51, 50]. Recent advances in distributed systems and sensor network tracing [52, 48] use optimized logical timestamps like Bi-Lateral clocks [52] and Interval Tree Clocks [48, 53]. Another alternative used by industrial-scale distributed systems tracing infrastructures like Apache HTrace [49] (based on Google Dapper [39]) and Twitter's Zipkin [56] is to uniquely identify each RPC call with a 128-bit identifier, propagate this identifier to the thread executing the RPC call and use physical clocks ("wall clock time", along with a clock synchronization protocol) to identify temporal causality.

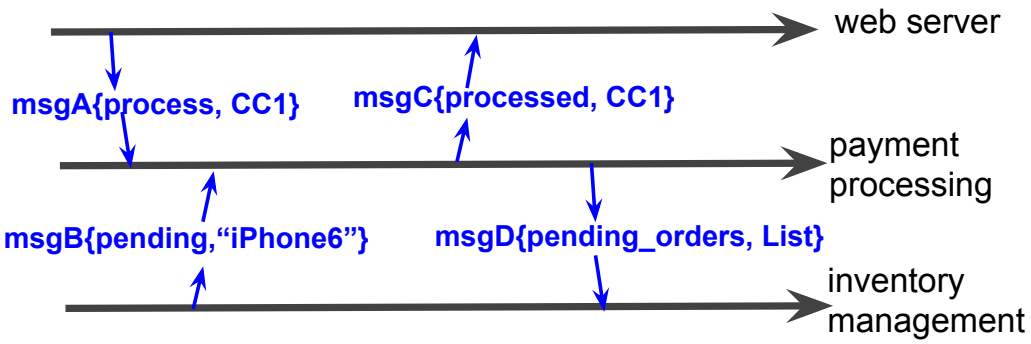


Figure 3 Temporal Causality : Illustration

However, temporal causality has drawbacks when used for elasticity management. This mainly stems from the fact that temporal precedence need not imply actual causality. For example, consider the illustration in Figure 3, $\{msgA, msgB\} \rightsquigarrow msgC$ and $\{msgA, msgB\} \rightsquigarrow msgD$, because both $msgA$ and $msgB$ temporally precede $msgC$ and $msgD$. However, *intuitively speaking*, it is clear that only $msgA$ "actually caused" $msgC$, because $msgA$ is a "process credit card" message to which $msgC$ is a response. Similarly, $msgB$ is a "get pending orders for iPhone6" message to which $msgD$ is a response. Direct causality captures

this intuitive and precise relationship.

We define a message (instance) M_1 to have directly caused message M_2 if there is a dynamic (runtime) control and/or data flow path from M_1 to M_2. Direct causality (denoted by $\longrightarrow$) is also illustrated with an abstract example in Figure 4, which we will use as a running example in this paper for direct causality tracking. Messages instances `msg1[x:150]` and `msg2[y:200]` will cause the emission of an instance of `msg3`. Hence { `msg1[x:150]`, `msg2[y:200]` } $\longrightarrow$`msg3[s:22500]`. Direct causality is thus the most comprehensive and precise notion of causality, capturing several subtle causal dependencies in distributed applications.

4 Direct Causality Analysis (DCA)

Direct causality analysis (DCA) to identify dynamic causal paths consists in: (i) static analysis to decompose a distributed application to identify its components, the kinds (or types) of messages between the components (ii) using program analysis along with transformation to collect minimal information from each component at runtime to identify causal relationships between incoming and outgoing messages, and (iii) storing said causal relationships in an external graph store/database, and using appropriate graph reachability queries to construct causal paths across the components of the application.

For reasoning about elasticity and other program behaviors, causal paths have to be identified at *runtime*. Compile-time identification of all possible causal paths, while easily performed through whole-program *static* control and data flow analysis of the distributed application, is not effective because it leads to combinatorial explosion and does not take workload characteristics into account. The key challenge in dynamic causality analysis and profiling is scalability to large distributed applications and the need to minimize runtime overhead.

4.1 Static Analysis : Application Graphs

The goal of this static analysis is to visualize the distributed application, and statically identify all possible interactions between its various components. DCA's initial static represents a distributed application as a directed graph of communicating *nodes* (called *application graph*), deployed in a cloud environment. A node in the graph is simply a *software module*, executing in a virtual machine (VM) or a container (like LXC [32], Docker [10], Warden [9], etc.). To handle replication and partitioning within components, we intentionally avoid representing *whole* components as nodes, and instead let nodes represent software modules. Consequently, a component is also a set of nodes, and a distributed application is a set of components. To simplify discussion, we assume a message-passing communication model. Other communication modes can be translated to the message-passing paradigm; a remote procedure call (RPC)/remote method invocation (RMI), for example, consists of two messages – an invocation message with the method arguments and a return message with the return value. Given two nodes, i.e., two software modules in the graph, there are a finite number of message types between them. We model message types as edges in the graph. To simplify static analysis and decomposition of an application into components and nodes, we further assume that messages are either sent/received using well-known APIs (e.g., RMI, JMS [38]/ZeroMQ [23] for Java) or that APIs used for sending/receiving messages are provided as input by the developer. The next step is to track direct causality between message (instances) and identify which of the various paths in the application graph are exercised at runtime.

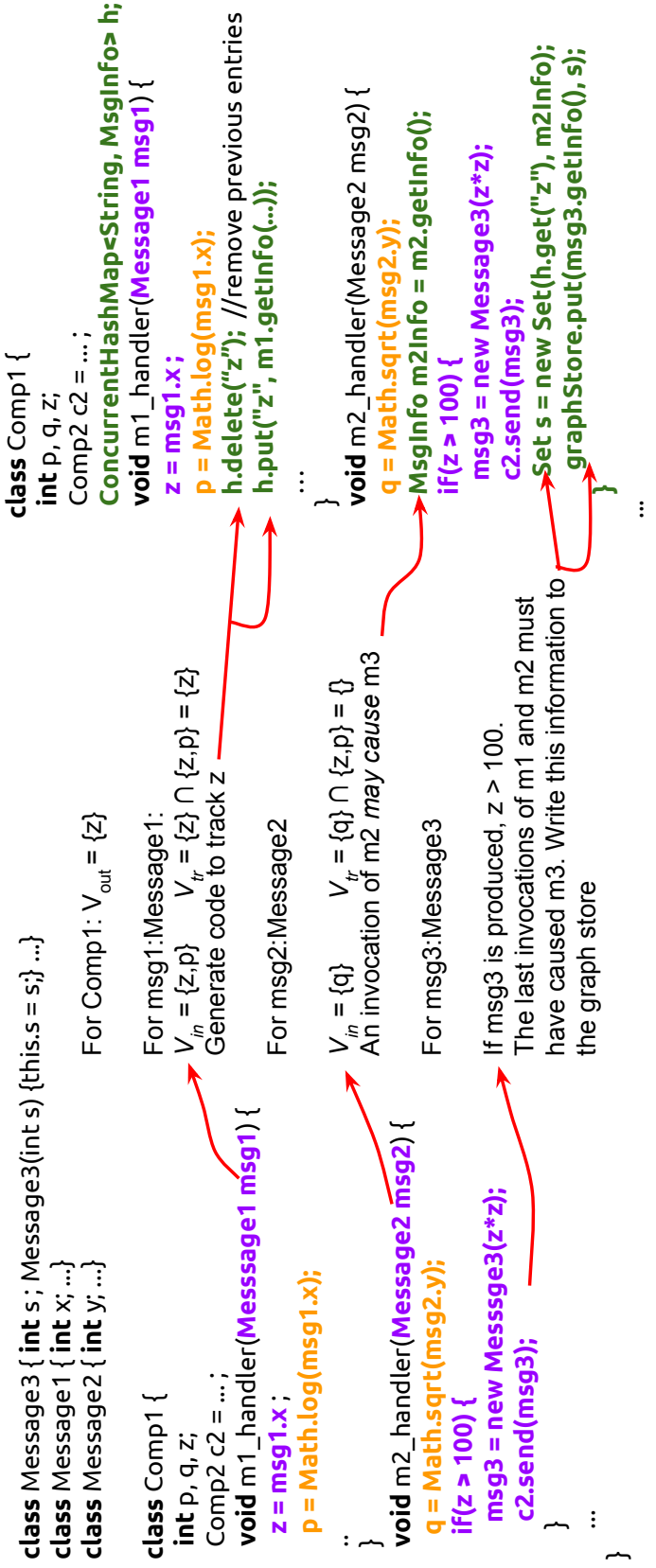


Figure 4 Illustrating DCA. Class `Comp1` and its instrumentation (indicated in green). Purple and yellow in `Comp1` on the left indicate data and control flow from incoming messages

Application	LOC	# causal paths	Runtime Overhead							
			DCA-100%		DCA-5%		DCA-10%		DCA-20	
			Range	Mean	Range	Mean	Range	Mean	Range	
Zookeeper 3.4.6	∼ 37000	2540	28.7–35.5%	32.6%	0.9–2.3%	1.63%	2.6–4.1%	4.12%	7.5–12.4%	
Hedwig 4.2.3	∼ 35500	1795	24.5–28.6%	27.5%	1.5–3.2%	3.38%	3.9–6.2%	5.39%	6.9–12.8%	
Marketcetera 2.2.5	∼ 29000	4236	34.6–39.5%	37.8%	1.9–3.6%	2.89%	5.6–8.5%	5.76%	7.2–11.8%	

Figure 5 Application Characteristics and Runtime Overhead of DCA at various sampling levels

4.2 Tracking Direct Causality

The core of DCA combines static analysis with preferential path profiling and preferential online control flow detection to efficiently track high-fidelity causal relationships between events in distributed applications. For each component C_i:

1. For each outgoing message `msgOut` ∈ class C_i, use backwards static slicing to identify the set S_{out} of variables `v` ∈ class C_i, such that `v` influences $send$(`msgOut`).
2. Compute the set of variables V_{out}, which influence the emission of any message from class C_i, as $V_{out} = \cup_{(msgOut \in C_i)} S_{out}$.
3. For each incoming message `msgIn`, DCA uses (forward) static slicing to:
   a. Identify the set of variables V_{in} that could be written by the execution path from $recv(M)$
   b. Eliminate the variables in V_{in} that can never influence the production of any message in C_i by computing $V_{tr} = V_{in} \cap V_{out}$. V_{tr} is the set of variables whose provenance has to be tracked to identify dynamic causal paths.
   c. Instrument C_i to store information about the variables in V_{tr} in a hash table as in the dynamic control flow algorithm of [43]. That is, for each of the variables in V_{tr}, DCA instruments the program (C_i) to dynamically store information about the messages that resulted in a write to the variable.

Figure 4 illustrates DCA and the corresponding code transformation of `Comp1`. `Comp1` emits only one message `msg3`, and the only variable that controls its emission is `z`. Hence, $V_{out} = \{z\}$. `Comp1` receives two messages `msg1` and `msg2`. For each of those messages, the sets V_{in} and V_{tr} are computed following the procedure described above. `msg1` writes to `z` and `p`, and `p` can be ignored because it is $\notin V_{out}$. Similarly, for `msg2`, it controls the emission of `msg3`, but its write to `q` can be ignored (because $q \notin V_{out}$). Our static analysis here also includes the analysis of `Math.sqrt` and `Math.log`, but these are pure functions with neither any side-effects, nor any indirect data/control flow through state variables in the `Math` class. To optimize DCA, we have pre-analyzed Java's standard libraries to identify APIs which have no side effects, so that they do not have to be repeatedly "re-analyzed" when DCA is applied to large applications. This is a key distinction between DCA and existing whole-program dynamic slicing and dynamic control/data dependence detection algorithms [43] – we reduce the overhead by only considering information flow *from* input messages *to* output messages. However, if other libraries are used, their source code/bytecode should be analyzed by DCA to check if there is data and control flow through the library from an incoming message to an outgoing message.

When `msg3` is produced, the events that directly caused it, i.e., the instances of `m1, m2` are uniquely identified using the method `getInfo`, and are stored in a graph store (Apache Titan [5]) for further analytics and construction of causal graphs. We uniquely identify each message M through a tuple, denoted by uid_M, consisting of $\langle IPAddress, ProcessId,$

$PerProcessSequenceNumber\rangle$. The unique identifier of an message uid_M along with information about the message (like data carried by a message, etc.) together form a node n in the causal graph. Two nodes are connected by a directed edge if one caused another, i.e., if $n_1 = \langle uid_{M_1}, info_{M_1}\rangle$, and $n_2 = \langle uid_{M_2}, info_{M_2}\rangle$ are two nodes in the causal graph, a directed edge connects n_1 and n_2 if and only if $M_1 \longrightarrow M_2$.

4.3 Identifying Causal Paths

Once causal relationships between incoming and outgoing messages within a component have been identified, e.g.,$n_1 \longrightarrow n_2 \equiv \langle uid_{M_1}, info_{M_1}\rangle \longrightarrow \langle uid_{M_2}, info_{M_2}\rangle$, the next step is to identify the causal path p where $n_1 \longrightarrow n_2$ fits. Since edges $n_1 \longrightarrow n_2$ are stored in a graph store, causal paths are determined through a simple breadth-first search (BFS). Indexing the elements, i.e., nodes n_i stored in the graph store by the unique identifiers of messages (uid_M) makes BFS extremely efficient. A causal path is determined by initiating BFS starting with the unique identifier of message corresponding to the external user request, until the node corresponding to the response from the application is obtained. Since, at every step in the BFS, after traversing an edge, the next edge is determined by a simple hash index lookup ($O(1)$), the time complexity of determining the causal graph induced by a message M is $O(|causal\ graph(M)|)$. The computation of this causal is triggered at the graph store when the edge corresponding to last message in the causal path, i.e., response to from the application is stored in the graph store. Since we have statically analyzed the application to construct the architectural graph, and have statically identified all possible causal paths in the application, we store information about these paths in the *profiler*. The profiler is executed on a "system monitoring and management" VM/physical host external to the application; and stores information about all possible paths, with their respective *path counts* set to zero. Whenever a causal path is identified by the graph store, it is sent to the profiler, which increments the corresponding path count.

4.4 Causal Probability and Elasticity

Causal probability is a "historic" metric, based on previously observed causal paths over a time interval (60 minutes in our case, which is configurable). Assuming the application architecture described in Sec 2, and denoting the set of all causal paths in the application by $ProgPaths$, causal probability P^c is intended to estimate the likelihood that a external customer request traces a specific $p \in ProgPaths$. At any point in time, the application would have serviced a number of external customer requests, which would have exercised different causal paths in the application incrementing their counts. Then the causal probability of a path p is determined as:

$$P^c(p) \ = \ count(p) / \sum_i (count(p_i) \mid p_i \in ProgPaths)$$

That is, given the path counts of all previously exercised program paths starting at a component in the distributed application, causal probability of path p is the probability that a newly arriving external request/message will induce p. Consider the e-commerce application in Figure 2 – let's say that the causal probability of paths *Purchase* and *Simple* at the web front-end are 0.69 and 0.31 respectively. These causal probabilities are useful because, once they have been computed, they provide a precise mechanism to autoscale the application once its workload increases. For example, if the workload of the web front-end increases by 2×, using external metrics and tools like Amazon CloudWatch [3] will dictate that the resources allotted to all components must be increased 2×. But, given current

causal path profiles and causal probabilities, resources allotted to Customer Tracking and Ad Serving should be increased 1.31× and resources allotted to Price DB and Inventory Management/Fulfillment should be increased 1.69× for optimal scaling. To see why this helps, consider, for example that all components are initially allotted 10 machines/VMs, for a total of 50 machines. Without using causal probability, 50 more machines would have been added to the application. Using causal probability reduces this number to 30 (10 to the web front-end whose workload increases by 100%, 7 each to price DB and inventory management whose workloads increase by 69% and 3 each to customer tracking and ad serving whose workloads increase by 31%). We round the number of machines to the nearest whole number. We use a linear regression model whose features are physical/virtual machine characteristics (CPU clock speed, RAM, network bandwidth), external workload and observed performance (throughput/latency) to build a linear regression model to predict the overall resource requirements of the application. Once a decision is made to increase or decrease the amount of resources available to the application, we use causal probability for proportional allocation of resources.

4.5 Implementation

As demonstrated in Section 5, DCA where all incoming requests and resulting causal paths are analyzed has a high overhead even when we only consider variables written by and read from messages. Hence, we also implement a variant of DCA where we randomly sample the messages and resulting causal paths for computing causal probability. We have analyzed three different sampling sizes for DCA – 5%, 10% and 20% of incoming messages. Our current implementation of DCA covers Java applications; it uses source code when available but can also work with byte code. We have implemented DCA using IBM's Watson Libraries for Analysis [22]. For the graph store, we use Apache Titan [5], which is a distributed fault-tolerant high-performance graph store. The Titan store and profiler are external to the application.

5 Evaluation

This section addresses the following research questions (RQ) empirically:

- **RQ1**: What is the runtime overhead of DCA?
- **RQ2**: Can DCA-based elastic scaling provision *exactly* the requisite amount of compute resources under dynamically varying workloads?
- **RQ3**: Does runtime overhead cause more resources to be provisioned than required? And, does runtime overhead obviate the benefits of elastic scaling?
- **RQ4**: Is sampling necessary, and at what level is it effective?
- **RQ5**: Since some applications and organizations can tolerate an excess of resources but not shortage, how frequently and to what extent are application SLAs violated while using DCA-based elastic scaling?

5.1 Applications Used

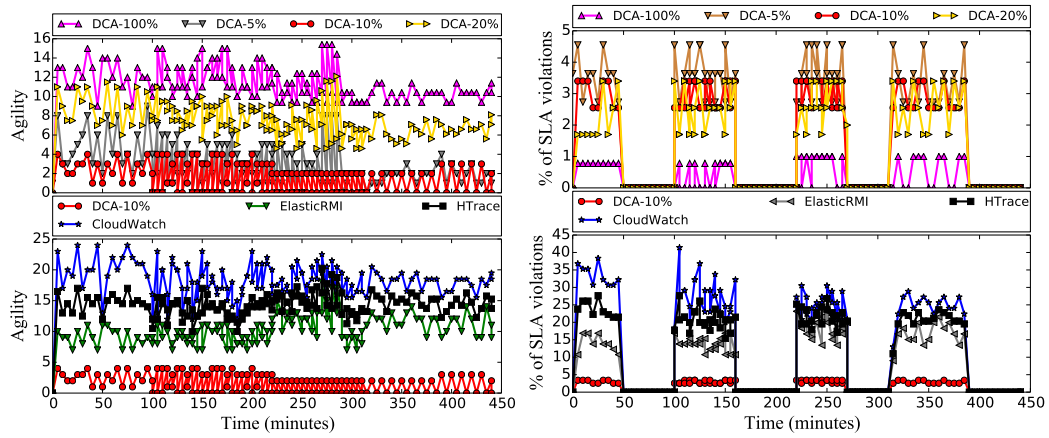
To illustrate DCA's broad applicability, we choose three widely-used, open-source, distributed applications:

- **Marketcetera [14]** is an NYSE-recommended [36] fault-tolerant algorithmic trading platform. The order routing system is the component that accepts orders from traders/automated strategy engines and routes them to various markets (stock/commodity), brokers and other financial intermediaries using the Financial Information Exchange (FIX) protocol – the QuickFiX/J implementation in a fault tolerant manner. For fault-tolerance, the order is persisted (stored) on two nodes. The workload for this system is a set of trading orders generated by the simulator included in the community edition of Marketcetera [14].
- **Apache Hedwig [19].** is a topic-based publish-subscribe system designed for reliable and guaranteed at-most once delivery of messages from publishers to subscribers. Hedwig is reliable, offering "exactly" once and "at-most" once delivery guarantees for messages, which results in agreement/consensus between nodes responsible for tracking the delivery state of messages.
  Clients are associated with (publish to and subscribe from) a Hedwig instance (also referred to as a region), which consists of a number of servers called hubs. The hubs partition the topic ownership among themselves, and all publishes and subscribes to a topic must be done to its owning hub. The workload for this system is a set of messages generated on a set of named publish-subscribe "topics" by the default Hedwig benchmark included in the implementation.
- **Zookeeper [46].** is a distributed co-ordination service for datacenter applications, similar to Google's Chubby [8]. Zookeeper has a hierarchical name space which can be used for distributed configuration, synchronization (mutual exclusion, distributed locks, etc.). Updates are totally ordered. It is used for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.The workload for this system is the default benchmark included in Apache Zookeeper [46].

5.2 DCA vs. ElasticRMI, CloudWatch and HTrace

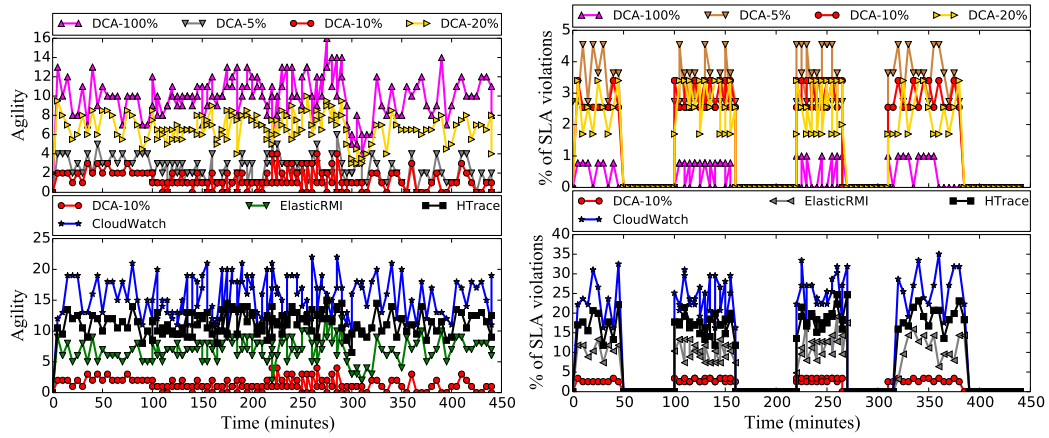
We compare elastic scaling with direct causality analysis (DCA) against two state-of-the-art technologies for engineering elastic distributed applications – ElasticRMI [26] and Amazon AutoScaling + CloudWatch [3] and with Apache HTrace [49], a distributed tracing framework for temporal causality. We use four variants of DCA – DCA-100% (where all external requests and the resulting causal paths are profiled), and DCA-5%, DCA-10% and DCA-20% (which correspond to DCA with 5%, 10% and 20% sampling). For ElasticRMI, we reimplemented the three applications with ElasticRMI and used explicit elastic scaling but not causality. Explicit elastic scaling, as explained in the our previous work [26] uses resource utilization metrics (CPU/RAM/disk) along with fine-grained information internal to the application for elastic scaling. This includes localized information about internal data structures, locks etc., but does not include information about workload history or path traces across nodes in a component and across components. In the CloudWatch scenario, we use a monitoring service – Amazon CloudWatch [3] to collect externally observable utilization metrics (CPU/Memory) from the nodes in the cluster and use a linear regression model on these metrics to decide whether to increase or decrease the number of nodes. We also combine CloudWatch's linear regression model along with path/span profiles (corresponding to temporal causality) obtained from HTrace [49] to perform proportional scaling of overloaded paths.

5.3 Experimental Setup



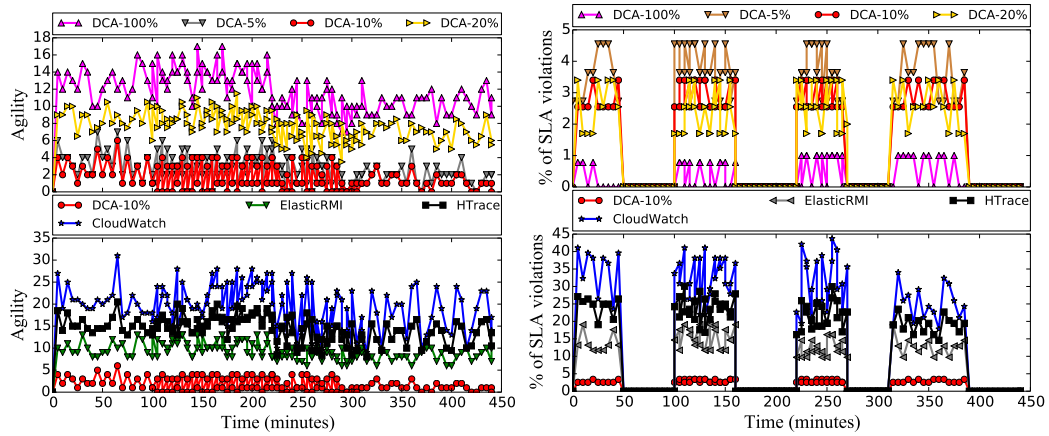
(a) Marketcetera – agility

(b) Marketcetera – SLA violations.



(c) Hedwig – agility

(d) Hedwig – SLA violations.



(e) Zookeeper – agility

(f) Zookeeper – SLA violations.

Figure 6 Provisioning efficacy, measured through agility and % of SLA violations

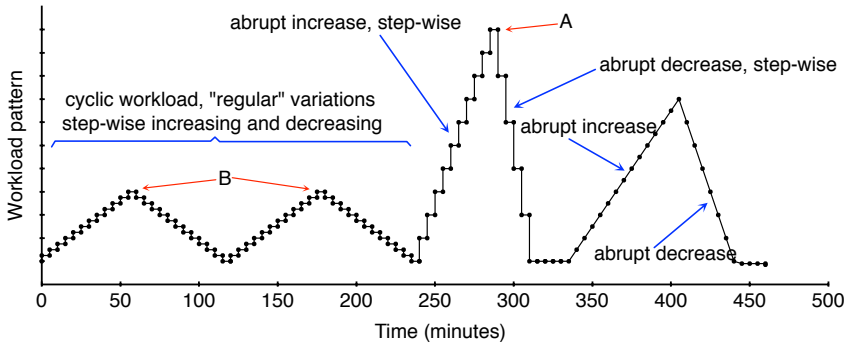


Figure 7 Workload pattern for all the three systems.

Evaluating elasticity is seldom about "normal" workload patterns, but rather about "irregular" workload patterns. To measure how well the system adapts to the changing workload, we use two patterns shown in Figures 7 – a cyclic pattern and a more abrupt pattern. These two patterns capture all common scenarios in elastic scaling which we have observed by analyzing real world applications deployed in the HP/IBM Cloud, and employed in our previous work on ElasticRMI [26]. A cyclic change in workload is shown by the first portion of the pattern in Figure 7, both continuous and stepwise. The abrupt pattern shown in Figure 7 has all possible scenarios regarding abrupt changes in workload – gradual non-cyclic stepwise increase, abrupt stepwise decrease, continuous increase and rapid continuous decrease in workload. Note however, that although the pattern remains the same for varying the workload for the four systems, the magnitude differs depending on the benchmark used, i.e., the values of points A and B in Figures 7 are different for the four systems depending on the benchmark. While the workload rate and pattern changes over the 450 minute run for both patterns, the elements of the workload are sourced from the default benchmarks included with the applications as described in Section 5.1.

5.4 RQ1: Runtime Overhead of DCA

Clearly, tracking dynamic data and control flow, even selectively for messages and the data items contained in them, incurs runtime overhead in application components. Figure 5 illustrates the runtime overhead of DCA at various levels of sampling for the workload pattern described in Section 5.3 and Figure 7. For each variant of DCA, Figure 5 lists the average runtime overhead, as well as the range into which 95% of the measured overheads (over the 450 minute run) fall. As expected, DCA-100% has the largest runtime overhead because all causal paths are tracked for all user inputs. Such mean runtime overheads between 27% and 38% are unacceptable because they will lead to more compute resources being provisioned to maintain the application's performance and SLA. However, using random sampling decreases runtime overhead significantly, as shown in Figure 5. Hence, the vital question we have to answer in the following sub-sections is whether there is a sweet-spot (RQ2–RQ4)between the additional resources required to handle DCA's runtime overhead and the (beneficial) reduction in compute resources that DCA-based elastic scaling provides.

5.5 RQ2: Provisioning Efficacy

Measuring elasticity is different from measuring scalability.

(Recall that) Elasticity is the ability of the application to adapt to increasing or decreasing workload; adding or removing resources to *maintain* a specific level of "performance" or

"quality of service (QoS)" [40]. Performance/QoS is specific to the application – typically a combination of throughput and latency. Hence, one way to characterize elasticity is "resource provisioning efficacy", for which we use a (SPEC standardized) metric called *Agility*, which, in a nutshell, measures the mean excess/shortage of compute resources.

5.5.1 Agility.

Agility characterizes the ability of a system provisioned to be as close to the needs of the workload as possible [40]. Assuming a time interval $[t, t']$, which is divided into N sub-intervals, Agility maintained over $[t, t']$ can be defined as:

$$\frac{1}{N}(\sum_{i=0}^{N} Excess(i) + \sum_{i=0}^{N} Shortage(i))$$

where $Excess(i)$ is the excess capacity for interval i as determined by $Cap\_prov(i) - Req\_min(i)$, when $Cap\_prov(i) > Req\_min(i)$ and zero otherwise; $Shortage(i)$ is the shortage capacity for interval i as determined by $Req\_min(i) - Cap\_prov(i)$, when $Cap\_prov(i) < Req\_min(i)$ and zero otherwise; $Req\_min(i)$ is the minimum capacity needed to meet an application's quality of service (QoS)/Service-level agreement (SLA) at a given workload level for an interval i; $Cap\_prov(i)$ is the recorded capacity provisioned by the elasticity management system (e.g., DCA) for interval i, and N is the total number of data samples collected over a measurement period $[t, t']$, i.e., one sample of both $Excess(i)$ and $Shortage(i)$ is collected per sub-interval of $[t, t']$.

Meaning of Agility. Agility has been standardized by the Standard Performance Evaluation Corporation (SPEC) [40]. Agility measures the shortage *and* excess of computing resources over a time period. For example, an agility value of 2 over $[t, t']$ when there is no excess means that there is a mean shortage of 2 physical hosts/VMs over $[t, t']$. *Lower agility values imply better elasticity.* For an ideal system, agility should be as close to zero as possible – meaning that there is neither a shortage nor excess.

Application	CloudWatch	ElasticRMI	HTrace	DCA-100%	DCA-5%	DCA-10%	DCA-20%
Marketcetera	18.19	10.27	14.23	11.35	2.91	1.57	7.53
Hedwig	15.45	6.91	11.18	9.9	2.29	1.27	6.74
Zookeeper	19.43	9.19	14.31	11.69	2.72	1.75	7.71

Figure 8 Average agility

5.5.2 Agility Results

The agility results of the four systems are illustrates in Figures 6 and 8 for all the three applications under the elasticity managers being evaluated. For computing Agility, the relevant QoS/SLA metrics are throughput and latency, though their precise semantics are different. Across all the three applications, we observe that using externally observable metrics like CPU/Memory utilization, as in Cloudwatch, is the most ineffective. Recall that lower agility values are better, and a value of zero is perfect. CloudWatch's agility is at least 50% higher than even DCA-100%. It never reaches zero; in fact, it is never lower than ten, as seen in Figure 6. ElasticRMI is better than CloudWatch because it employs fine-grained internal metrics of the application, and doesn't have to blindly rely on CPU/memory utilization. However, ElasticRMI requires *re-writing* all the three applications; not merely

re-factoring but a deep understanding of the code to use fine-grained application-specific metrics and properties of internal data structures.

Also, using HTrace with CloudWatch improves Agility, but only marginally. This is because Trace supports only temporal causality, which as illustrated in Section 3 is not precise and leads to several false positives in identifying messages that actually caused a given message. This imprecision compounds over several hundred causal paths in the three large-scale applications being evaluated. Recall that HTrace requires manual annotations and identification of spans in RPCs, but provides better temporal causality than logical clocks (vector clocks, interval tree clocks and bilateral clocks). Hence, we believe that the use of logical clocks will only further degrade the elasticity (compared to HTrace) of the applications being evaluated.

On the other hand, DCA-100%, where causal paths induced by all external inputs are tracked, has better agility than CloudWatch but not ElasticRMI. This is because of the runtime overhead of DCA-100%, which requires more resources to be provisioned to track causal paths than to handle workload changes. All variants of DCA, however, do not require the application to be re-written, merely re-compiled. With sampling, runtime overhead decreases, and agility improves significantly. DCA at 5%, 10% and 20% sampling levels has better agility than CloudWatch and ElasticRMI. The subtle difference in agility between the 5% and 10% sampling levels is especially interesting. DCA-5% has the lowest runtime overhead as illustrated in Figure 5 but still has worse agility (up to 80% higher) than DCA-10%. This is because, while a 5% sampling decreases overhead, it doesn't give the most accurate picture of the workload. DCA-10% has higher overhead but is able to make much better predictions of the amount of resources required to handle the workload. We also observe that the agility of DCA-10% reaches zero most frequently, around 48% of the time during the experimental run over 450 minutes. The trends in Figure 6 also illustrate that all the elasticity managers have the capability to adapt to changing workloads, trying to decrease agility. But, DCA-10% is the most effective.

5.6 RQ3 : Effects of Runtime Overhead

The answer to this RQ is that runtime overhead does cause more resources to be provisioned than required. This is because the throughput and latency of the application change due to runtime overhead of DCA, and more resources are needed to meet the SLA in the presence of the overhead. In fact, for DCA-100%, runtime overhead worsens Agility. If the overhead of completely re-writing the application is *not taken into account*, it does indeed obviate the benefits of elastic scaling. This, however is not true for DCA-5% and DCA-10%. Runtime overhead is much lower, as shown in Figure 5, and the benefits of DCA are significant.

5.7 RQ4 : Sampling Efficacy

We also observe from Figure 5, 6 and 8 that sampling is essential to get a sweet spot between resource provisioning efficacy and runtime overhead. From our experiments, we observe that sampling around the 10% threshold seems most effective. Again, sampling should be uniformly random across the workload; this requires careful examination of the application's architecture. The approach we took was to examine the front-end components receiving user-requests to see whether they were partitioned and/or replicated for load balancing and fault tolerance. For x% sampling with k front-end servers, we randomly chose $\frac{x}{k}$% of user-requests at each server.

5.8 RQ5 : Percentage of SLA violations

Due to the fact that some applications and organizations can tolerate an excess of resources but not shortage, it is important to evaluate how frequently and to what extent are application SLAs violated while using DCA-based elastic scaling. An agility value of zero means that application SLAs are always met despite increasing or decreasing workloads. But a positive value can either be due to a shortage of resources or an excess. When there is a shortage, some application SLAs are violated. Hence, percentage of SLA violations is another metric to measure the effectiveness of elasticity management mechanisms.

Figure 6 illustrates the percentage of SLA violations for all four systems over the entire 450 minute run. We observe that SLA violations do not occur when the workload is decreasing, because there are excess resources which have to be de-provisioned. Until de-provisioning happens, they are used to meet SLAs and the application is not starved. This explains why the percentage falls to zero in all six graphs of Figure 6. We also observe that the percentage of SLA violations is below 5% for all variants of DCA. The trend in SLA violations is different from that of agility, in a subtle way. Although DCA-100% has worse agility than DCA at all other sampling levels, its SLA violations are the lowest (less than 1%). This is further evidence that the larger agility values of DCA-100% are primarily a result of DCA's runtime overhead than as a result of either shortage or excess, i.e., SLA is almost always met by DCA-100% but more resources are needed to handle runtime overhead. On the other hand, in ElasticRMI, agility is better because runtime overhead is low, but SLAs are violated more (typically between 10–15%). Using externally observable metrics with CloudWatch results in the most SLA violations; using HTrace with temporal causality helps, but only marginally. This trend is similar to that of Agility. Sampling increases SLA violations, but not significantly – it is between 3-4% for 5% and 10% sampling levels, which is an acceptable threshold for many distributed applications.

6 Related Work

Also, while DCA is novel with respect to causality analysis in distributed applications, it builds on, tweaks, and refines three decades of previous work on dynamic slicing and dynamic control dependence. In this section, we focus on the most closely related work.

Early work on slicing focussed on slicing of sequential programs. Examples include [21, 1, 20] and [29]. We point the reader to [41] for a detailed survey of slicing techniques circa 1995, and [44] provides a more recent survey of dynamic slicing algorithms. [34, 45] present a static slicing algorithms and optimizations for slicing concurrent programs, but do not discuss their applications to distributed applications or evaluate their runtime overheads. [30] discusses context-sensitive and context-insensitive slicing in detail, and presents a context-sensitive slicing algorithm for concurrent programs. The algorithms of [34] and [30] are analyzed/compared in detail and their implementation and runtime overhead is evaluated in [16]. It is possible to extend these algorithms to track causality in distributed applications; however our starting point for DCA was the (further optimized) dynamic slicing algorithms of [44].

Distributed program slicing for debugging and partial re-execution of message passing distributed applications was first introduced by [11]. As opposed to finding out the exact execution path and input sequence that caused any possible bug, our objective is to simply extract the causality relationship between messages. The runtime overhead of [11] is unknown. We are able to keep overheads as low as possible by not tracing irrelevant code paths and by not serializing and storing intra-component dependence graphs after every execution.

Furthermore, by uniquely identifying messages with IP address, process id, thread id and per-process sequence number, we obviate the need to construct dynamic communication dependence edges as in [11], which relies on applications co-operating to execute a distributed graph traversal algorithm re-construct the distributed dependence graph. That, in turn, requires group communication (and a consensus protocol), which severely limits its scalability. On the other hand, our use of a dedicated graph store, transfers the overhead of constructing causal paths at runtime from the executing application to the machines/VMs hosting the graph store. Also, using a graph requires group communication and querying to be run on a much smaller set of machines (typically < 10 even to handle terrabytes of data). We believe that the above issues may not have been a concern for [11] because it seems to have been intended for debugging a distributed application in a LAN-like environment. All the above arguments also hold for subsequent work in [28].

While DCA has not been implemented for programs written in other languages, DCA is not fundamentally limited to Java, and can be extended to handle C, C++, Scala, etc., as demonstrated for more generic dynamic slicing by [44]. Observation-Based Slicing (ORBS) [**?**] is a recent, language-independent slicing technique capable of slicing polyglot systems, including systems which contain (third party) binaries. A potential slice obtained through repeated statement deletion is validated by observing the behaviour of the program: if the slice and original program behave the same under the slicing criterion, the deletion is accepted. The resulting slice is similar to a dynamic slice. This is an excellent and promising approach to slicing real applications; although extending it to handle distributed applications is non-trivial because it would involve building a platform to observe the execution of a distributed program and a slice across several machines. Also, applying this deletion-based approach can be applied to track aspects of performance (like causality and elasticity) in production deployments would probably require provisioning substantial additional resources to execute slices with deleted statements.

Our ongoing work [27] extends ElasticRMI [26] to handle multiple languages and programming idioms, but at present, retains the need to re-write applications. Also, our causal probability technique builds on previous work and insights gained from path profiling [7] and preferential path profiling [42]. We would also like to emphasize that while DCA is only one contribution of this paper, this paper primary goal is to demonstrate that causality can serve as an effective foundation for engineering elasticity. Another paper related to DCA is [31], which uses logging to detect attack provenance on single-machine applications. Key differences include the purpose of the analysis (elasticity vs. attack provenance for security), handling of distributed application through graph stores, and the techniques used (logging vs. analysis and transformation). [31] also has higher runtime overhead because it is not focussed on messages, and would requires offline analysis of logs using map-reduce/Spark to infer causality relationships, which adds to provisioning latency and increases agility and percentage of SLA violations when compared to graph database triggers to compute causal paths in DCA.

7 Conclusions

In this paper, we have presented a novel and intuitive way of thinking about, designing and engineering elasticity management components in distributed applications based on causal relationships between messages. We have defined a notion of direct causality that takes into account control and data flow in determining causal relationships between messages, and have presented a program analysis and transformation technique called direct causality

analysis (DCA) to track these relationships at runtime with low overhead. We have evaluated our techniques on three real-world applications and compared causality tracking with other state-of-the-art tools like CloudWatch and ElasticRMI to gain significant software engineering insights w.r.t the performance of DCA. We have demonstrated the effectiveness of causality tracking in elastic scaling through a significant (more than 10×) reduction in agility values (lower values are better) and percentage of SLA/QoS violations under dynamically varying workloads. We have also empirically analyzed the subtle interplay between runtime overhead of DCA and its resource provisioning efficacy, and demonstrated that runtime overhead doesn't obviate provisioning efficacy when sampling is used. We have also demonstrated that sampling at around 10% achieves a sweet spot between runtime overhead, agility and percentage of SLA violations.

References

1. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI'90*
2. Amazon Web Services (AWS). Amazon Elastic Compute Cloud (EC2). *http://aws.amazon.com/ec2/*, 2012.
3. Amazon Web Services (AWS) Inc. Amazon CloudWatch. *http://aws.amazon.com/cloudwatch/*, 2012.
4. Apache. The Solr Enterprise Search Platform *http://lucene.apache.org/solr/*.
5. Aurelius. The Tian Distributed Graph Store *http://thinkaurelius.github.io/titan/*.
6. AWS Inc. Amazon Auto Scaling. *http://aws.amazon.com/autoscaling/*, 2012.
7. T. Ball and J. R. Larus. Efficient path profiling. In *MICRO'96*.
8. M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06*.
9. Cloud Foundry. Warden Containers *https://github.com/cloudfoundry/warden*.
10. Docker. Docker Containers *https://www.docker.com/*.
11. E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, 1993.
12. R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. In *SIGCOMM '12*.
13. O. Foundation.
14. G. Miller and T. Kuznets and R. Agostino. Marketcetera Automated Trading Platform. *http://www.marketcetera.com/site/*, 2012.
15. A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 57–64, Philadelphia, PA, June 2014. USENIX Association.
16. D. Giffhorn and C. Hammer. Precise slicing of concurrent programs. *Automated Software Engg.*, 16(2):197–234, June 2009.
17. Google. Universal Search *http://googleblog.blogspot.com/2007/05/universal-search-best-answer-is-still.html*. 2007.
18. Hadoop. *http://hadoop.apache.org*. 2015.
19. Hedwig. *https://cwiki.apache.org/ZOOKEEPER/hedwig.html*.
20. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88*.
21. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, Jan. 1990.
22. IBM. T. J. Watson Libraries for Analysis *http://wala.sourceforge.net/wiki/index.php/Main_Page*.
23. iMatix. ZeroMQ *http://zeromq.org/*. 2015.

24 M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07*.

25 J. Kirsch and Y. Amir. Paxos for Systems Builders. *http://www.cnds.jhu.edu/pub/papers/cnds-2008-2.pdf*, 2008.

26 K. R. Jayaram. Elastic remote methods. In *MIDDLEWARE'13*.

27 K. R. Jayaram. Towards explicitly elastic programming frameworks. In *ICSE'15*.

28 M. Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *ICSM'95*.

29 B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, Oct. 1988.

30 J. Krinke. Context-sensitive slicing of concurrent programs. In *ESEC/FSE'03*.

31 K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS'13*.

32 Linux. Linux Containers *https://linuxcontainers.org/*.

33 Memcached. *http://www.memcached.org*. 2015.

34 M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA '00*.

35 H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC'13*.

36 NYSE/Marketcetera Press Release. NYSE Technologies and Marketcetera Launch New Era Software-as-a-Service Trading Platform on SFTI Network. *http://www.marketcetera.com/site/node/153*, 2009.

37 OpenStack Foundation. Ceilometer *https://wiki.openstack.org/wiki/Ceilometer*. 2015.

38 Oracle. Java Message Standard *https://docs.oracle.com/javaee/7/api/javax/jms/package-summary.html*.

39 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.

40 SPEC Open Systems Group (OSG). Report on Cloud Computing to the OSG Steering Committee. *http://www.spec.org/osgcloud/docs/osgcloudwgreport20120410.pdf*, 2012.

41 F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

42 K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *POPL '07*.

43 B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *ISSTA '07*.

44 X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE '03*.

45 J. Zhao, J. Cheng, and K. Ushijima. Static slicing of concurrent object-oriented programs. In *ACSAC'96*.

46 Zookeeper. *http://zookeeper.apache.org/*.

47 K. Kim and J. Steele and Y. Qi and M. Humphrey. Comprehensive Elastic Resource Management to Ensure Predictable Performance for Scientific Applications on Public IaaS Clouds UCC'14.

48 J. Mace and R. Roelke and R. Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems ACM SOSP'15

49 Apache Software Foundation. HTrace: A tracing framework for use with distributed systems *http://htrace.incubator.apache.org*

50 K. Birman. Guide to Reliable Distributed Systems: Building high Assurance Applications and Cloud-Hosted Services Springer-Verlag. 2012

51 C. Cachin and R. Guerraoui and L. Rodrigues Introduction to Reliable and Secure Distributed Programming Springer-Verlag. 2011.

52 V. Sundaram and P. Eugster. Lightweight Message Tracing for Debugging Wireless Sensor Networks DSN'13.

53 P. Almeida and C. Baquero and V. Fonte. Interval Tree Clocks: A Logical Clock for Dynamic Systems *http://gsd.di.uminho.pt/members/cbm/ps/itc2008.pdf*

54 U. Erlingsson and M. Peinado and S. Peter and M. Budiu and G. Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. ACM TOCS 30 (4). 2012.

55 R. Fonseca and G. Porter and R. Katz and S. Shenker and I. Stoica. X-Trace: A pervasive network tracing framework NSDI'07.

56 Twitter Inc. Zipkin: A Distributed Tracing System. *https://twitter.github.io/zipkin/*, 2015.

57 K. R. Jayaram. Exploiting Causality to Engineer Elastic Distributed Software *http://www.jayaramkr.com/files/causalitytechreport.pdf*, 2015.