An Infrastructure for Automating Large-scale Performance Studies and Data Processing

Deepal Jayasinghe, Josh Kimball, Tao Zhu, Siddharth Choudhary, and Calton Pu.
Center for Experimental Research in Computer Systems, Georgia Institute of Technology
266 Ferst Drive, Atlanta, GA 30332-0765, USA.
{deepal, jmkimball, tao.zhu, siddharthchoudhary, calton}@cc.gatech.edu

***Abstract*—The Cloud has enabled the computing model to shift from traditional data centers to publicly shared computing infrastructure; yet, applications leveraging this new computing model can experience performance and scalability issues, which arise from the hidden complexities of the cloud. The most reliable path for better understanding these complexities is an empirically based approach that relies on collecting data from a large number of performance studies. Armed with this performance data, we can understand what has happened, why it happened, and more importantly, predict what will happen in the future. However, this approach presents challenges itself, namely in the form of data management. We attempt to mitigate these data challenges by fully automating the performance measurement process. Concretely, we have developed an automated infrastructure, which reduces the complexity of the large-scale performance measurement process by generating all the necessary resources to conduct experiments, to collect and process data and to store and analyze data. In this paper, we focus on the performance data management aspect of our infrastructure.**

***Keywords*-Automation, Benchmarking, Cloud, Code Generation, Data Warehouse, ETL, Performance, Visualization.**

I. Introduction

An application that performs one way in the data center may not perform identically in computing clouds [16]. Hence, companies need to ensure that their applications can move safely and smoothly to the cloud, because failing to do so could result in significant impacts to the business. Neglecting the possible performance impacts could ultimately lead to lower user satisfaction, missed Service Level Agreements (SLAs), and worse, reduced profit. To prevent such outcomes, a rigorous experimentation and holistic data analysis effort must accompany any cloud migration effort. This approach can help us to understand what has happened, explain why it happened, and more importantly, anticipate what will happen in the future. However, this empirical approach is not without its own set of challenges.

These challenges arise from the nature of large-scale performance experimentation. The introduction of the cloud significantly increases the number of possible system configuration permutations, which increases both the amount of testing and the degree of experimental data heterogeneity–diversity and volume. These data management challenges alone make large-scale experimentation impractical to manage using manually intensive techniques. Finally, the nature of the cloud increases the complexity of other, more pedestrian testing activities such as application deployment, configuration, workload execution and monitoring.

We address these challenges through a flexible automation framework that we have developed to create, store and analyze large-scale experimental measurement data–called Expertus. Automation removes the error prone and cumbersome involvement of human testers, reduces the burden of configuring and testing distributed applications and accelerates the process of reliable applications testing. The main contribution of this paper is the tools and approaches we have developed to automate the data (structure, size, patterns, and noise)-related aspects of the large-scale experiment measurement process.

Our approach addresses the three fundamental data management issues–generation, extraction/collection and storage/-analysis. First, Expertus generates all of the resources that are necessary to automate the execution of an experiment. Using a provided domain-specific language (or provided web portal), a user merely provides a description of the experiment to execute. To address the extraction and data processing challenges, we extend ETL (extract, transform, and load) tools and approaches [17], [18] to build a generic parser to process the collected data. The current parser can handle a significant fraction (over 80%) of the most commonly used file formats in our experimental domain. To address the storage and analysis challenges, we have designed a special, fully dynamic data warehouse (Experstore) to store performance measurement data. Finally, we have built a web portal to address the related challenges of navigating and analyzing an enormous amount of performance measurement data. This tool in addition to embedded data analysis provided by the the *R* framework helps the user to navigate, visualize and analyze the data in the warehouse. The details related to data analysis facilitated by the web portal appear do not appear here but instead in a longer form of this research.

The remainder of this paper is structured as follows. In Section II we provide the big picture of the experiment automation framework. We discuss the code generator framework to aid the automation in Section III, and Section IV provides our approach to performance data generation. Section V presents our data warehouse solution, and in Section VI, we discuss our approach to extracting the performance data. We evaluate the effectiveness of our approach in Section VII. Finally, we provide a discussion of the related state of the art approaches

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

in Section VIII, and we conclude the paper with Section IX.

II. Automated Performance Measurement Infrastructure

We address the above challenges by leveraging automated techniques for performance measurement. More concretely, we have developed Expertus — an automated infrastructure to fully automate the performance measurement process. In our approach, a user provides the configuration file for the experiment, and the infrastructure generates all of the required resources (shell scripts and other configuration files), runs the experiments (i.e., deploy and configure applications, run the workloads), and collects and uploads the data to the data warehouse. Finally, the user can analyze the data using either command line tools (*R* or other means) or a web portal. The complete process for experiment measurement using our approach is illustrated in Figure 1. A brief description for each item in the figure is provided below:

- *Code generator:* is the core, which generates all the necessary resources to automate the experiment management process. In a nutshell, code generator takes experiment configuration files as the input and generates resources to automate the process.
- *Experiment Driver:* is designed to use the generated resources and controls the experimentation flow, which involves application deployment, configuration, initialization, workload execution, and data collection. Code generator generates all the scripts, and a special script called `run.sh`, which maintains the sequence for script execution. Experiment driver uses `run.sh` to find the order of execution. It connects to all the nodes through `SSH/SCP` and executes the scripts on the corresponding nodes.
- *Data Extraction:* Each experiment produces gigabytes of heterogeneous data for resource monitors (e.g., CPU, Memory, thread pool usage, and etc...), response time, throughput and application logs. The structure and amount of collected data vary based on sundry factors, including: system architecture (`64-bits` vs `32-bit`, `2-core` vs. `4-core`), monitoring strategy and monitoring tools (e.g., `sar`, `iostat`, `dstat`, `oprofile`), logging strategy (e.g., Apache access logs), and the number of deployed nodes and workloads. Data extractor is written to help users easily import experiment data into the data warehouse. It supports the most commonly used data formats and has built-in flexibility to extend to new data formats.
- *Data Warehouse:* Due to the nature of large-scale performance experiments, creating a priori, fixed schema to store measurement data is difficult. Even if one could be defined, data processing becomes extremely inefficient due to the magnitude of the data. To overcome these challenges, we have created a flexible data warehouse tailored to handle performance measurement data.
- *Data Analysis:* The reason for conducting large-scale experiments is to find and resolve performance issues. To this end, data analysis plays an integral role; yet, due to the magnitude of the data and structure of the data warehouse, data analysis becomes a non-trivial task. To address these challenges, we have provided two types of tools: a web portal for graphical users and *R* scripts for command line users. Both of these tools understand the internal data structure, and they help to make data analysis efficient.

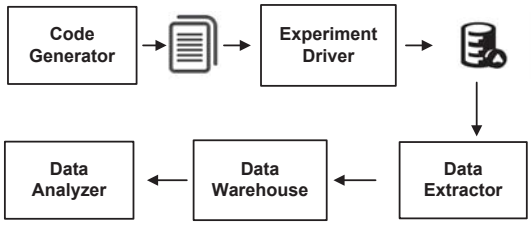

Fig. 1. A Typical Performance Measurement Process with Our Approach.

III. Automation through Code Generation

In our approach, we enable automated experiment measurement through code generation, which generates all the necessary resources to automate the measurement process. From an architectural viewpoint, our code generator adopts a compiler-based approach of multiple, serial transformation stages – a code generation pipeline. The intuition behind this approach is to deliver more extensibility and flexibility by dividing the larger problem into smaller pieces and processing them one at a time. The hallmarks of our approach are two-fold: the stages typically operate on an XML document that is an intermediate representation, and XSLT performs the code generation. We address challenges that originate from differences among clouds, applications, users and other cross cutting requirements (e.g., monitoring) through aspect oriented programming (AOP) techniques.

Use of XML provides the code generator with a high degree of extensibility. This stems from XML's simple, well-defined syntax requirement and its ability to accept arbitrary new tags, thereby bypassing the overhead encountered when managing both XSLT templates and AOP. For example, a template can add an arbitrary element to the intermediate XML; however, unless the processing code is written to process this new tag, the newly added tag remains untouched. XSLT transformation is the process of converting an XML document into another document through the use of XSL. Typically, XSLT converts an XML document into another XML document (e.g., HTML) or any other type of document. Expertus consists of two types of templates, namely `Resource templates` and `Aspect templates`. The former is used to generate application/-platform independent part of a resource, and the latter is used to modify (weave) the generated resource for the target application/platform (e.g., Emulab vs. EC2).

Expertus takes an XML document and produces another XML document through XSLT transformation. Expertus treats the first and last stage differently as compared to the rest

of the pipeline. In the first stage, it takes the experiment specification as the input, and in the final stage, it generates the automation resources for the target file system (in lieu of an intermediate XML). At each stage, Expertus uses the intermediate XML document created from the previous stage as the input to the current stage. It uses the intermediate XML file to retrieve the names of the templates that are needed for the current stage, and it transforms the intermediate XML document, which produces yet another XML document. If needed, AOP `pointcuts` are added to the intermediate XML during the transformation phase. Consequently, Aspect Weaver is used to weave such `pointcuts` into the intermediate XML. Aspect Weaver processes the `pointcuts` through `Aspect templates` and creates the woven XML. The woven file is then written to the file system using the file writer. During the final stage of the pipeline, the automation scripts are written to the file system; while at each of the other intermediate stages of the pipeline, an intermediate XML is generated, and the next stage in the pipeline is called.

IV. Approach to Performance Data Generation

In our approach to performance measurement, we execute workloads by deploying actual or representative applications (e.g., benchmarks like RUBBoS [10], RUBiS [11], Cloudstone c [7]) on actual or representative deployment platforms (e.g., Amazon EC2). These large-scale experiments produce a huge amount of heterogeneous performance data. The heterogeneity of the data arises from the assemblage of applications, clouds, monitoring tools, and monitoring strategies used. We conduct large-scale experiments and collect data by fully automating the process, and our code generator generates all the necessary resources to automate this process.

Experiment driver takes care of the experimental deployment and configuration, and once deployed, the driver executes the workload against the specific, deployed configuration. In this step, we run the planned experiments according to the availability of hardware resources. For example, we usually run the experiments by increasing the workload. For each workload, we run the easily scalable (browse only) scenario first, followed by read/write scenarios. After each batch of experiments, we collect data, ramp-down the system, stop all servers, and start the next batch of experiments. This sequence allows for sufficient ramp-up time, which minimizes cache inter-dependencies across experiments. The iterations continue until all of the experiments have been completed.

During experiment execution, the experiment driver collects information about system resources (e.g., CPU, memory), application specific data (e.g., thread pool usage), application logs (e.g., apache logs), high level data like throughput and response time, and any other data that the user wants to collect. This process continues for each and every workload. In fact, experiments in our domain consist of 50 to 60 workloads, and each workload runs for approximately 30 minutes. The framework is capable of collecting, managing and storing data without any help from the user. The data extractor, as the name implies, extracts this data and stores it in the data warehouse after the experiment has completed.

V. Flexible Data Warehouse

During large-scale performance measurement, researchers do not know beforehand which resources need to be monitored (whether it be high-level data like response time or throughput or low-level data like resource utilization data and application logs). Monitoring all the potential resources is infeasible because of the performance overhead. Researchers may choose different monitoring regimens, change testing strategies or software and test on hetergenous platforms. Because of this type of variability, experimental data cannot be feasibly stored in a set of static tables. Moreover, failures during experimentation usually lead to incomplete or faulty data that waste database resources and slow data processing. Even without failures, these data tables tend to be quite large, so processing becomes very expensive if these tables cannot be loaded into primary memory.

To address above challenges, we have designed Experstore – a special data warehouse designed to store performance measurement data. Expestore is fully dynamic that is its tables are created and populated on-the-fly based on the specific experimental data. At the end of each experiment, we create a set of tables to store the data, and the resultant schema is solely based on the structure of this data (e.g., how many columns, tables, relations, etc.).

The experiment measurements in our domain consist of multiple workloads running against a deployed system (a unique configuration of hardware and software); hence, each experiment produces measurement data for each and every workload. We have designed the experiment driver to store measurement data for each experiment in a separate directory. In most cases, each directory follows the same structure (names and number of files). During the data loading stage, the data loader iteratively processes all of the directories i.e., it recurses over all of the directories and loads the corresponding data contained in each. The data loading configuration maps directories to workloads, and the loader uses the information about the data parser to process data files contained in a given directory.

As mentioned earlier, large-scale experiments commonly result in failures, and storing failures is incredibly wasteful. During the data loading stage, the data loader creates db scripts to remove all of the failure data for a given experiment. The data engine uses these same scripts to recover in the event of a loading failure. To minimize the possibility of such a failure, we reduce data loading overhead by not loading data in a transaction.

During data parsing, each file is matched to a profile, and the parser uses this profile to update the database accordingly. More concretely, data processors provide an API for the parser such that the parser only needs to provide values for each row, and the data engine does the rest of the work. This approach enables the parser and the data engine to be loosely coupled.

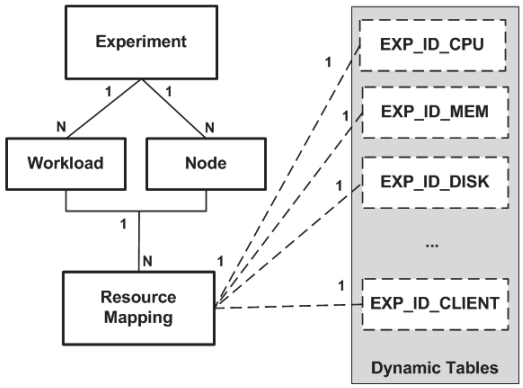

Fig. 2. Experstore - Static and Dynamic Tables

The data loading configuration file provides all the necessary data for the data loader and the data engine. For example, a user can specify how to format a given data field (e.g, datetime), where to begin and end in a file, and how to relate a data column (row) to columns in the database table and etc.... This configuration file can be reused across experiments as long as the directory structures are identical across experiments. For each resource type, we create a 'profile', which maps a file's structure to an applicable schema, i.e. relating the columns in a CSV file to a particular database table. Next, we have a mapping, which specifies what profiles apply to a given node. A mapping contains node name, file name and corresponding profile. A sample profile and a mapping file is shown below:

Listing 1. Code Listening for Profile and Mapping

```
<profile>
    <separator>,</separator>
    <resource-name>CPU0</resource-name>
    <processor-class>dataimport.filter.CSVFileProcessor
    </processor-class>
    <column index=''0'' colname=''user'' datatype=''double''/>
    <column index=''1'' colname=''system'' datatype=''double''/>
    <start-index>10</start-index>
    <end-index>0</end-index>
</profile>

<mapping nodename=''Apache'' filname=''169.254.100.3.csv''
         startwith=''false'' endwith=''false''
         profiles=''CPU0,DISK,CPU1,NETWORK,SYSTEM''/>
```

The structure of the data warehouse is shown in Figure 2. As shown in the figure, it consists of four static tables that store experimental metadata (e.g., experiment name, platforms, node and workload information), which are typically fixed across experiments. As shown in the figure, the highlighted tables are the tables that are created on-the-fly,. 'Resource Mapping Table' stores the names of the dynamically created tables along with the resource names. For example, it has a record for CPU utilization for experiment ID (EXP_ID), and the value is EXP_ID_CPU. Likewise, all the monitoring data for a given experiment is stored. In fact, it has a record for each unique node, workload, and resource.

VI. Automated Data Extraction

In general, the problem of extracting data from various log file formats reduces to a problem of attempting to disambiguate presentation concerns from those related to data. While this reduction narrows the thinking around this problem, it does not account for the numerous points of variability that occur, particularly related to any given log file's layout and structure, e.g. the presentation of the embedded data. At the highest level, files can be described as containing unstructured, semi-structured or structured data. Most of the log files presented in our domain fall into the semi-structured category (the remaining portion are structured). Hence, an approach that accommodates semi-structured files could be used for structured, so we focused on solving semi-structured files.

Laender *et* al. [22] suggests wrapper inductive approaches might be particularly relevant for this problem because of their reliance on format and presentation. This observation serves as the foundation for the intuition for our design. In short, wrapper inductive approaches rely on format and structure to impart order when order is not explicit. The extractor begins by creating a replica of the log file in memory. Next, it performs a "first pass" to probabilistically encode rows of the file such that they are coded as containing: the header ("header rows"), data ("data rows") or some other type of information ("misc rows") such as generic batch job information. Next, it attempts to match the data rows to headers (many of the files contain more than one header in the file.) To accomplish this matching, the extractor uses order and presentation information (particularly invisible ASCII characters) to compute the probability that a given row of data corresponds to a given header in the log file of interest. Once we have a match, we use the presentation information of the header row - layout and structure - to extract data from the matched data rows. Once the data has been extracted, it is loaded into a data warehouse loading file. During this entire process, the operator is asked to evaluate or validate rows that do not have significant statistical power, e.g. the rows received low encoding or matching probabilities. In this case, the operator provides input to the extractor to either encode or match the row (depending on the specific algorithm) based on his judgment.

This design primarily relies on two algorithms: the `Row-Encoding Algorithm` and the `Matching Algorithm`. Row-Encoding Algorithm works by prompting a user only when the system "thinks" the row is a header row. Headers have two distinct characteristics. They contain more alphabetic and more special characters relative to the total length of a given string. "Misc" rows, i.e rows that should be ignored for later processing, have one of these two properties but not both, which differentiates them from header rows. The core of this algorithm works by calculating string length-weighted character frequencies. The second, the Header-to-Data Row Matching Algorithm, operates similarly to Row-Encoding. First, it computes character frequencies and scales the "weights" corresponding to these frequencies by the type of character identified, e.g. visible ASCII vs. invisible ASCII. The algorithm makes a match by calculating two metrics: the bytewise difference of "invisibles" between a header row and a given data row in addition to the (vertical) distance between a header row and a given data row. The lowest sum of these two metrics yields a match.

VII. Effectiveness of the Infrastructure

We have used Expertus extensively to perform a large number of experiments on different computing clouds; through experimentation, we have collected a huge amount of data with various data formats, stored these in the data warehouse, and observed interesting performance phenomena. In this section, we evaluate the success of our approach managing performance measurement data.

A. Usability of the Tool

Here, we present how quickly a user can change an existing specification to run the same experiment with different settings (e.g., MySQL Cluster vs. C-JDBC), on different clouds (e.g., Emulab vs. EC2), with different numbers of nodes (e.g., two vs. four app servers), or entirely different applications (e.g., RUBBoS vs. Cloudstone). In our analysis, we created a specification (say `a.xml`) to run the RUBBoS application on Emulab with a total of 16 nodes and generated automated resources using Expertus. We then changed `a.xml` to generate automated scripts for EC2, which required only a single line change (i.e., `<param name=''platform'' value=''EC2''/>`) in `a.xml` and an IP address modification. Even though only a few lines changed in the configuration file, the changes to the generated code were material and non-trivial. We followed the same procedure and modified `a.xml` to change the database middleware from C-JDBC to MySQL Cluster. This change required modifying only 36 lines (mostly MySQL Cluster-specific settings), but the differences in generated code were huge. Similarly, by changing only 4 lines in `a.xml`, we were able to move from 2 to 8 Application servers. Furthermore, with only 52 template line changes, we were able to extend the support from RUBBoS to Cloudstone.

B. Generated Script Types and Magnitude

The biggest advantage of our approach becomes apparent when automating experiments for complex applications. The number of resources generated by Expertus depends on the application (e.g., RUBBoS, RuBiS), software packages (e.g., Tomcat, JBOSS), deployment platform (e.g., Emulab, EC2), the number of experiments, the number of servers, and the number of configuration parameters. To show the difference in the generated code, six different hardware configurations (on Emulab) were selected, and the number of generated lines for each configuration was counted. When the number of nodes increases, the size of the generated code grows significantly, as do the differences among the generated code bases. The magnitude of the generated code implies two conclusions: the effectiveness of our approach and the enormous hurdles confronting manual approaches. For example, an experiment with 43 nodes would require approximately 15K lines of shell scripts–a non-trivial undertaking for manual-based approaches.

C. Richness of the Tool

Richness is considered as the breadth and depth of supported software packages, clouds, and applications the infrastructure supports. Expertus has been used over three years to conduct a large number of experiments spanning five clouds (Emulab, EC2, Open Cirrus, Wipro, and Elba), three applications (RUBBoS, RUBiS, and Cloudstone), five database management systems (C-JDBC, MySQL Cluster, MySQL, PostgreSQL, Oracle), various resource monitoring tools (dstat, sar, vmstat), and varying numbers and types of nodes.

D. Success of Data Generation

Table I provides a high level summary of the many different experiments performed using the RUBBoS, RUBiS, and Cloudstone benchmarks. In the table, experiment refers to a trial of a particular experiment i.e., execution of a particular workload against a combination of hardware and software configurations. Typically, a trial of an experiment takes one hour which is the aggregated value of: reset time, start time, sleeping time, ramp-up time, running time, ramp-down time, stop time, and data copy time. As such, in Emulab, we have spent approximately 8,000 hours running experiments. In the table, nodes refer to the total number of machines we have used during our experiments. We calculated the number of nodes by multiplying the number of experiments by the number of nodes for each experiment. Configuration means the number of different software and hardware configurations that have been used in our experiments. Finally, the number of data points collected describes the amount of data we have collected from executing these experiments.

TABLE I
Number of Experiments Performed with Expertus

Type	Emulab	EC2	Open Cirrus	Elba	Wipro
Experiments	8124	1436	430	2873	120
Nodes	95682	25848	4480	8734	430
Configurations	342	86	23	139	8
Data points	3,210.6M	672.2M	2.3M	1328.2M	0.1M

E. Testing for Heterogeneous Data Formats

For the purpose of evaluating the robustness of the extractor (or parser), the following file patterns were tested: 1) one header, 2) multiple header rows with sequentially corresponding data, 3) multiple header rows with non-sequential corresponding data, and 4) multiple header rows appearing randomly in the file with data occurring non-sequentially, (e.g., data does not correspond to the header it follows). These patterns were distilled by sampling the known domain of log files. During testing, we used actual collected performance data that adhered to these aforementioned patterns, and Table II outlines the observed results. The file patterns also differed in header structure. Based on the sample, rows designated as headers contained either one row or two rows. A header with one row, *Only Field Row Header*, only contained data fields. Alternatively, a header with two rows, *Record & Field Row Header* contained a row, which enumerated the data records, and another row, which listed the corresponding data fields for each record. For this latter case, the numbers of records and fields were varied from 1 to 8 (number of records) and 16 (number of fields) respectively. If the headers were correctly matched to the applicable row of data, the specified test received a *PASS* grade; otherwise, it received a *FAIL* grade.

TABLE II
EVALUATION SUMMARY OF SUPPORTED FILE FORMATS

Pattern	Only Field Row Header	Record & Field Row Header
One header	PASS	PASS
Multiple header (sequentially data)	PASS	PASS
Multiple header (non-sequential data)	PASS	PASS
Multiple header (randomly headers)	N/A	FAIL

VIII. RELATED WORK

Benchmarking is an essential approach used in both academia and industry to gain an understanding of some or all of the following: system behavior, hypothesis formulation and testing, systems configuration and tuning, solution development, and performance bottleneck resolution. However, few efforts have had dual aims of building software tools for large-scale testing of distributed applications and reducing the complexity associated with benchmarking [1]–[6]. The ZOO [3] has been designed to support scientific experiments by providing experiment management languages and supporting automatic experiment execution and data exploration. Zenturio [4] on the other hand, is an experiment management system used for parameter studies, performance analysis and software testing of cluster and grid architectures. One of the closest approaches to ours is Weevil [8], which also focuses on workload generation and script creation. In their later studies, the Weevil team observed some of the limitations in their approach and obstacles for reaching higher levels of confidence [5] with their results. To our knowledge, these efforts haven't explored the issues of extensibility, flexibility, or modularity that is presented in this paper.

IX. CONCLUSION

Expertus, our automated experiment management framework, has been developed to minimize human errors and maximize efficiency when evaluating computing infrastructures experimentally. We have used the framework for a large number of experimental studies, and through these, we have collected a huge amount of data, which we have used for identifying interesting performance phenomena. In this paper, we discussed the use of the infrastructure for efficiently creating, storing and analyzing performance measurement data. The code generator generates the necessary resources to fully automate the experiment measurement process, and using these generated scripts, users can run experimental studies to actually generate performance data. The automated data processor processes heterogeneous data and stores this data in a flexible data warehouse, built specifically for measurement data. We evaluated the proposed automation framework based on its usage, the amount of data it can accommodate, different monitoring and logs formats it supports, and finally, the overall effectiveness of the approach based on the needs of the scientific community. Our future work includes, extending the data parser to support additional data formats, extending the data warehouse to use No-SQL databases, and extending the visualization tool to support more customizable graphing capabilities.

ACKNOWLEDGMENT

This research has been partially funded by National Science Foundation by IUCRC/FRP (1127904) , CISE/CNS (1138666), RAPID (1138666), CISE/CRI (0855180), NetSE (0905493) programs, and gifts, grants, or contracts from DARPA/I2O, Singapore Government, Fujitsu Labs, Wipro Applied Research, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

REFERENCES

[1] Y. Ioannidis, M. Shivani and G. Ponnekanti. ZOO: A Desktop Experiment Management Environment. In *Proceedings of the 22nd VLDB Conference*, Mumbai(Bombay), India, 1996.
[2] K.L. Karavanic and B.P. Miller. Experiment management support for performance tuning. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, Mumbai(Bombay), India, 1996.
[3] R. Prodan and T. Fahringer. ZEN: A Directive-based Language for Automatic Experiment Management of Distributed and Parallel Programs. In *ICPP 2002*, Vancouver, Canada.
[4] R. Prodan and T. Fahringer. ZENTURIO: An Experiment Management System for Cluster and Grid Computing. In *Cluster 2002*.
[5] Y. Wang, A. Carzaniga and A.L. Wolf. Four Enhancements to Automated Distributed System Experimentation Methods. In *ICSE 2008*.
[6] S. Babu, N. Borisov, S. Duan, H. Herodotou and V. Thummala. Automated Experiment-Driven Management of (Database) Systems. In HotOS 2009, Monte Verita, Switzeland.
[7] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox and D. Patterson. Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement tools for Web 2.0. In CCA 2008.
[8] Y. Wang, M.J. Rutherford, A. Carzaniga and A. L. Wolf. Automating Experimentation on Distributed Testbeds. In ASE 2005.
[9] Emulab - Network Emulation Testbed. http://www.emulab.net.
[10] RUBBoS: Bulletin board benchmark. http://jmob.objectweb.org/rubbos.html.
[11] RUBiS: Rice University Bidding System. http://rubis.ow2.org/.
[12] Open Cirrus: Open Cloud Computing Research Testbed. https://opencirrus.org/.
[13] WIPRO Technologies. www.wipro.com/.
[14] Amazon Elastic Compute Cloud. http://aws.amazon.com.
[15] S. Malkowski, M. Hedwig and C. Pu. Experimental evaluation of N-tier systems: Observation and analysis of multi-bottlenecks. In *IISWC 2009*.
[16] D. Jayasinghe, S. Malkowski, Q. Wang, J. Li, P. Xiong and C. Pu. Variations in performance and scalability when migrating n-tier applications to different clouds. CLOUD 2011.
[17] P. Vassiliadis. A Survey of Extract-Transform-Load Technology. Integrations of Data Warehousing, Data Mining and Database Technologies: Innovative Approaches (2011).
[18] R. Baumgartner, G. Wolfgang and G. Gottlob. Web Data Extraction System. Encyclopedia of Database Systems (2009): 3465-3471.
[19] R. Kohavi, R.M. Henne and D. Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the HiPPO. In ACM KDD 2007.
[20] S. Malkowski, D. Jayasinghe, M. Hedwig, J. Park, Y. Kanemasa and C. Pu. Empirical analysis of database server scalability using an n-tier benchmark with read-intensive workload. ACM SAC 2010.
[21] S. Malkowski, Y. Kanemasay, H. Chen, M. Yamamotoz, Q. Wang, D. Jayasinghe, C. Pu, and M. Kawaba, Challenges and Opportunities in Consolidation at High Resource Utilization: Non-monotonic Response Time Variations in n-Tier Applications. IEEE Cloud 2012.
[22] A. Laender, B. Ribeiro-Neto, A.S. da Silva and J.S. Teixeira. A Brief Survey of Web Data Extraction Tools. ACM Sigmod Record 31.2 (2002).
[23] G. Linden. Make Your Data Useful, Amazon, November 2006. [Online]. http://home.blarg.net/~glinden/StanfordDataMining.2006-11-29.ppt