DataMill: Rigorous Performance Evaluation Made Easy

Augusto Born de Oliveira
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
a3oliveira@uwaterloo.ca

Jean-Christophe Petkovich
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
j2petkov@uwaterloo.ca

Thomas Reidemeister
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
treideme@uwaterloo.ca

Sebastian Fischmeister
Electrical and Computer Engineering
University of Waterloo
Waterloo, ON, Canada
sfischme@uwaterloo.ca

Abstract

Empirical systems research is facing a dilemma. Minor aspects of an experimental setup can have a significant impact on its associated performance measurements and potentially invalidate conclusions drawn from them. Examples of such influences, often called hidden factors, include binary link order, process environment size, compiler generated randomized symbol names, or group scheduler assignments. The growth in complexity and size of modern systems will further aggravate this dilemma, especially with the given time pressure of producing results. So how can one trust any reported empirical analysis of a new idea or concept in computer science?

This paper introduces DataMill, a community-based easy-to-use services-oriented open benchmarking infrastructure for performance evaluation. DataMill facilitates producing robust, reliable, and reproducible results. The infrastructure incorporates the latest results on hidden factors and automates the variation of these factors. Multiple research groups already participate in DataMill.

DataMill is also of interest for research on performance evaluation. The infrastructure supports quantifying the effect of hidden factors, disseminating the research results beyond mere reporting. It provides a platform for investigating interactions and composition of hidden factors.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

ICPE'13, April 21–24, 2013, Prague, Czech Republic.

Keywords

DataMill; performance; experimentation; infrastructure; robustness; repeatability; reproducibility

1. INTRODUCTION

Empirical computer performance evaluation is essential for computer science and industry alike. The empirical measurement of performance sees widespread use to guide the research of new ideas and the development new technologies. A performance improvement of a few percentage points may mean large savings in dollars, when applied to a large data center with billions of clients. It is also essential, then, that computer practitioners dominate the methodology necessary to evaluate computer performance correctly.

However, the research community [10, 25, 31, 32] has demonstrated that experimental evaluation in computer science is difficult. As a consequence, experiment design, setup, analysis, and data reporting are often neglected. Results obtained from experimentation are sensitive to the experiment design and setup. Conclusions drawn from performance experiments may vary across a combination of hardware and software factors. Sometimes subtle changes to an experiment's setup can have a significant impact on its outcome [21]. The lack of rigorous performance evaluation is often blamed on the fact that experimentation is difficult and costly [31], but those obstacles are not insurmountable.

To improve the current state of experimentation, we propose DataMill. DataMill is a distributed infrastructure for computer performance experimentation targeted at scientists, researchers, and aspiring researchers. DataMill aims to allow the user to *easily* produce robust, replicable, and reproducible results at low cost. To do so, DataMill executes the experiments on real hardware and incorporates the results from existing research on how to setup experiments and hidden factors. For example, the infrastructure automatically varies a selection of hardware and software factors, and therefore reduces the effort required by the user to setup the experiment while simultaneously increasing the robustness and applicability of the experimental results to a wide range of factors. The user need not know the details of the underlying mechanisms required to vary these factors

and may simply take advantage of DataMill infrastructure. DataMill is mainly developed by the University of Waterloo, however, several universities (e.g., Purdue University, the University of Pennsylvania, the University of Lugano, and the Federal University of Santa Catarina) have already joined the effort and have provided benchmarking nodes.

Besides making it easy and low cost to the user, DataMill also aims to alleviate the problem of data availability and the reproduction of experimental setups. Based on our own experience, few researchers responded to inquiries for experiment setup details and raw data used in their publications. In DataMill, all experiment setup parameters and their experiment files remain stored in the infrastructure. Users can choose to make their experiments public facilitating the replication and reproduction of the their experimental setups and results. We believe DataMill has the potential to watermark performance evaluation experiments by making the data publicly and consistently available. Finally, DataMill and the public repository of repeatable experiments can serve as a valuable tool for the education of aspiring researchers.

The contributions of this paper are:

1. The design and implementation of a distributed infrastructure for computer performance evaluation;
2. The reproduction and expansion of a previously published experiment, confirming the existence of significant hidden factors;
3. A survey of current performance evaluation practices in computer science publications, demonstrating the need for more rigorous experimentation.

2. BACKGROUND

In traditional experimental design [2, 20], *factors* are properties of an experiment that affect the *response variable*; i.e., the metric of interest of that particular experiment. It is common knowledge in statistics that the one-factor-at-a-time method (OFAT), which is a sequential exploration of a *design space* (the space defined by the factor dimensions), is a poor method of experimentation. Despite its weaknesses, OFAT is the *de facto* standard in computer science. One alternative to OFAT, factorial design, attempts to ascertain the effect of all factors on the response variable at the same time, avoiding OFAT's "blind-spots".

In order to illustrate the state of experimental methodology in computer science, we conducted a survey of the recent publications to SOSP 2011, ASPLOS 2012, OSDI 2012, ATC 2012, EuroSys 2011, and PLDI 2012. The results are shown in Table 1. The columns, in order, show the total number of papers, the fraction that contains empirical performance evaluation, the fraction that describes their hardware platforms and software versions, the fraction that contains a comparison (either with a baseline or a competitor), the fraction of those that contain a formal statistical test for that comparison, the fraction that performs the experiment under different conditions as a sanity check, the fraction that contains a dispersion metric (standard deviations, confidence intervals, empirical CDFs, etc.), and the fraction that published the software being evaluated, the fraction that uses a publicly available workload (established benchmarks, input files, etc.), and, finally, the fraction that published their resulting data.

While many papers include an empirical performance evaluation, many papers do not list the versions of the software used for these experiments. Worse, even more of these publications do not contain an empirical comparison to a baseline or a competing approach. Out of those papers that include such a comparison, only a small fraction formally tests that their performance figures are different from the baseline. Only a few submissions make their experiment implementation and the results publicly available. Although the benefits for using a dispersion metric have been reiterated by many researchers [31, 10, 32] in the past, a number of papers that do a performance evaluation do not use any measure of dispersion in their performance evaluation.

We performed an experiment to demonstrate the dangers of using a single experimental setup like the majority we found in our survey. Figure 1 shows the performance measurements of Iperf [22], a network performance benchmark, repeatedly running on a loopback interface (the machine was disconnected from the network) in blocks of one hour. The x-axis shows the blocks, and the y-axis shows the bandwidth through the loopback interface. We use a standard boxplot to characterize a block, adding a $\times$ symbol to mark the mean. Each block contains 330 samples, and the machine reboots between blocks. While performance is generally stable, in two out of 72 blocks the mean performance is approximately 10% better, even though everything else *remained the same*. A "lucky" developer could mistake the rare block for the norm, and report results that are a full 10% away from the average case.

Computer science experiments are susceptible to many hidden factors such as the one seen above [34, 18, 21, 13, 19]. The systematic exploration of their effect on metrics of interest is a difficult and time-consuming task. Automated hidden factor exploration is necessary, and would best be handled by an experiment infrastructure. Many published results in computer science cannot be validated on newer platforms. Reasons include loss of data, loss of source code, loss of the original workload, and insufficiently described experimental conditions. Automated replication and publicly accessible setups are necessary.

Access to a wide variety of experimental setups is difficult to obtain, so experimental design spaces are generally small, and are often comprised of just a single machine. Automated exploration of a large design space is also necessary. Furthermore, testing an innovation on various platforms forces one to consider more corner cases. This will overall lead to more robust innovation whose performance is not only known on more platforms but also that works on more platforms. Specific exploits of particular platform features only provide limited value to the research community.

It follows, therefore, that a distributed benchmarking infrastructure with the capability of varying software and hardware factors is in order. Based on these findings, we created DataMill, an experimentation infrastructure that addresses these shortcomings automatically. DataMill enables users to make experiment packages, workloads, and results publicly available. We collect several performance counters available in the Linux environment and compute dispersion figures automatically. We support experiments that contain multiple software packages to enable a baseline comparison or a comparison with a competing approach. We believe by providing a platform that automates the factor variation and many tasks of the experiment set up, empirical performance evaluations will become more popular among researchers.

Conference	No. of Papers	Perf. Eval	HW Desc.	SW. Desc.	A/B Test	Formal Test	Sanity Check	Disp. Metric	Public SW	Public Load	Public Data
SOSP'11	27	93%	92%	60%	91%	0%	24%	44%	4%	68%	0%
ASPLOS'11	36	61%	100%	55%	54%	0%	5%	31%	4%	82%	4%
OSDI'10	30	93%	71%	61%	54%	0%	0%	32%	7%	39%	0%
ATC'12	40	95%	87%	58%	68%	0%	3%	32%	8%	26%	0%
EuroSys'11	22	86%	95%	74%	84%	12%	42%	47%	21%	58%	0%
PLDI'12	45	64%	90%	79%	93%	0%	37%	10%	24%	72%	0%

Table 1: Illustration of Experimental Rigor in Recent Scientific Publications

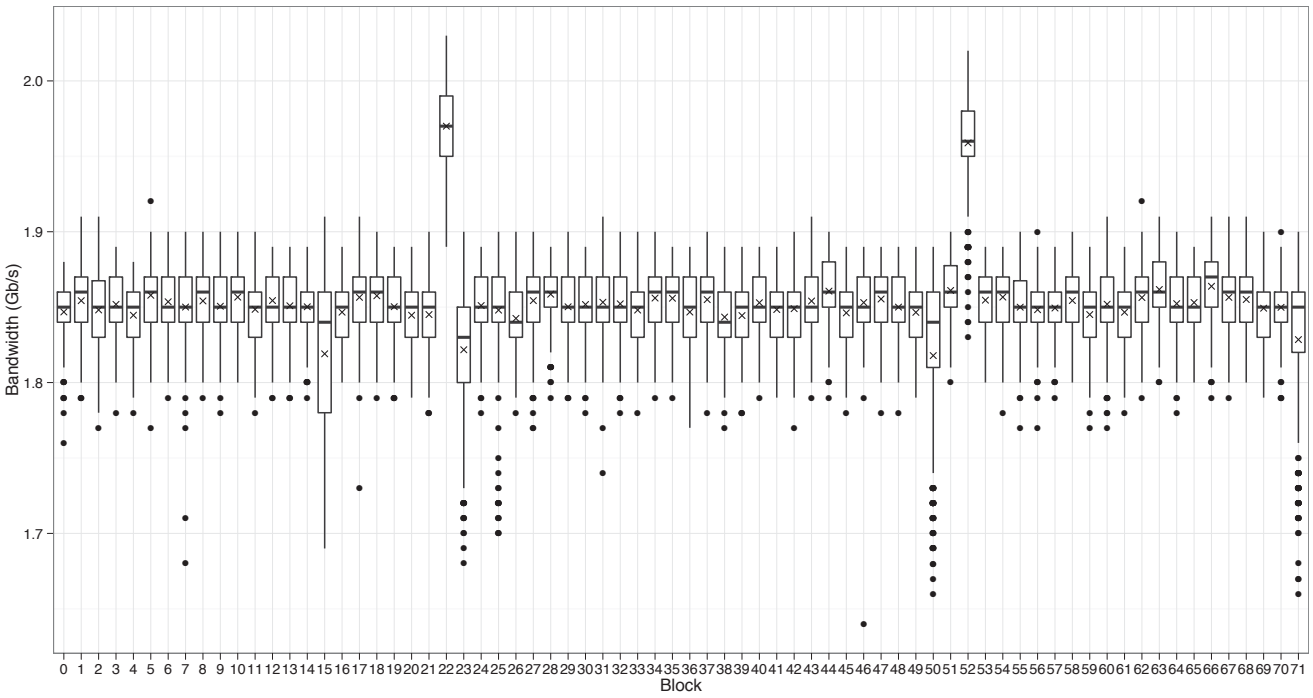


Figure 1: Iperf Performance in Blocks of 1 Hour

We now present DataMill, a world-wide open distributed performance evaluation system that aims to address the aforementioned points and implements the lower two levels of an experiment infrastructure [10].

3. DATAMILL: THE USER EXPERIENCE

DataMill is a distributed computer-performance evaluation infrastructure aimed at minimizing user effort required to get robust, reliable, and reproducible results. DataMill distributes arbitrary benchmarks and experiments to a large number of heterogeneous worker nodes, located all over the world, facilitating the creation of replicable results, and reducing the necessary effort to produce robust results.

DataMill facilitates replicable results by storing details of any particular experiment setup, allowing automated experiment reproduction. DataMill achieves robustness through replication across a wide range of experimental setups including variation of state-of-the-art known hidden factors. DataMill is a public-service infrastructure and available to scientists around world as long as they donate equipment for evaluations themselves.

In this section we describe DataMill as seen by a user, and later describe the inner workings of DataMill in Section 4. The user experience starts with packaging the experiment for DataMill to execute, then defining an experiment space from an array of hardware and software factors, and, finally, collecting and analyzing the resulting data.

3.1 Packages

Each experiment contains one or more packages. A single package experiment can quantify performance over a wide range of setups, while experiments with more packages allow performance *comparisons* over those setups.

Each package contains: (1) the source code and any input data for the experiment, and (2) auxiliary DataMill-specific scripts to set up, execute, and collect the results from the experiment. All package components are encapsulated in a compressed TAR file.

There is no restriction on experiment code, and users have administrator-level access to workers during experimentation. Users can generate free-form results data in arbitrary metrics, and then collect it via compressed packages. These features allow evaluating the performance of a wide range

of software, ranging from user-space applications to kernel modules. Security concerns are minimal at this point, because participating users must contribute to the infrastructure and, therefore, are well known and trusted.

There are only two scripts that every DataMill package must contain: `run.sh` and `collect.sh`. These are the scripts that execute and collect data from the experiment, respectively. If the package requires a setup procedure (such as decompression, compilation, dependency installation, etc.), it may also contain a `setup.sh` script, which will be executed before `run.sh`. Finally, if there is the need for environment variables during execution, an `env` file may be included in the package. The contents of this environment file will be added to the experiment environment before each execution.

To exemplify the construction of a DataMill package, consider the Dhrystone [36] benchmark. It consists of a single source file, `dry.c`, which installs and runs the benchmark. To execute it, one must first set its executable flag, which is done in the setup script shown in Listing 1. Note that Dhrystone is self-compiling, so we change the file to be executable. The script assumes the source file is located in the `/dry/` directory.

```
1 #!/bin/sh
2
3 cd /dry/
4 chmod +x dry.c
```

Listing 1: Setup Script

Listing 2 shows the `run.sh` script that executes Dhrystone, capturing its output (standard out and error) to the `/dry/results` file. The output is appended to the end of the file, allowing multiple executions before collection.

```
1 #!/bin/sh
2
3 cd /dry/
4 ./dry.c >> results 2>&1
```

Listing 2: Run Script

Finally, Listing 3 shows the script that packages the results file for collection. It simply compresses the result file using a unique name, and echoes the final archive's file name. This file will then be downloadable by the user once the experiment finishes executing.

```
1 #!/bin/sh
2
3 results=/dhrystone-`date +%Y%m%d_%H%M`.tar.gz
4 tar czf $results /dry/results > /dev/null 2>&1
5 echo $results
```

Listing 3: Collect Script

The user must package the experiment source code and DataMill scripts in a GZipped TAR file. All DataMill scripts should be in the root directory of this archive. For the Dhrystone package, the file structure of the package would be as shown in Figure 2.

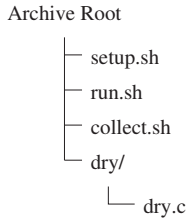


Figure 2: Dhrystone Package Directory Structure

To facilitate package creation, we provide users with a virtual machine image that mimics a DataMill worker. Users can develop, test, and debug their packages in a local environment until they are sure their packages are ready for production, at which point they can submit it for execution via our experiment creation interface.

3.2 Experiment Creation

Once the packages are ready, the user submits them for execution through a file upload interface. Users must then define an experiment for DataMill to execute. Experiment definition has three steps:

1. Package selection
2. Constraint definition
3. Experiment space definition

Package selection consists of choosing which packages will be executed; one or more packages may be selected. In the constraint definition step, users inform DataMill of any limitations to their benchmark. For example, if the experiment can only execute on the ARM9 architecture and requires at least 2GB of RAM, the user expresses these constraints through an intuitive web interface.

Finally, in the experiment space definition step, users select how many experimental setups they wish to explore. The interface presents each dimension of the experiment space, divided in hardware and software categories, in a unified percentage-scale. This percentage represents what fraction of all available options should be *covered*. For example, users can define that their experiment must contain 100% of CPU available architectures and that 75% of GCC optimization flags be tested. This will cause DataMill to select machines such that every architecture in the infrastructure is represented, and to re-execute the benchmarks in a randomly selected subset of optimization flags that covers at least 75% of the total number optimization flags. Setting any dimension to 0% results in the random selection of a single level for that dimension.

After machines are selected according to the hardware dimension settings, the software dimensions are combined to generate each individual *job*. For example, if the user chose to explore all five GCC optimization flags on a single machine, there would be one job generated per optimization flag, all attributed to that machine. If, in conjunction with the optimization flags, both settings of the address randomization feature of Linux (on or off) should be tested, then the number of jobs grows to 10 (5 flags times 2 address randomization settings). Therefore, the number of jobs for each experiment scales with the experiment space defined by the user, and care must be taken to avoid combinatorial explosion and an experiment with an excessive number of jobs.

3.3 Experiment Results

After the user defines the experiment space and DataMill creates jobs, it will distribute the experiment's packages for execution, then collect the individual result files. The web interface dynamically updates experiment information as data arrives, allowing users to monitor their experiment's progress. In addition to the data collected by the collect script, DataMill collects additional metrics with minimal overhead [33], such as total execution time, and the number page faults and cache misses. This data is also made available to the user.

Finally, once all jobs associated with an experiment are finished, users can download the full experiment results file, which contains the result file from every job. Examples of how to analyse large datasets generated by DataMill are provided in Section 5.

4. DATAMILL: THE INFRASTRUCTURE

Making the user experience described in Section 3 a reality requires considerable engineering effort. The DataMill infrastructure is composed of a *master node*, responsible for the distribution of experiment trials and the collection of results, and several *worker nodes*, which execute the experiment packages provided by the users. This section describes how DataMill was implemented and how its different parts interact.

4.1 Master Node

The responsibilities of the master node are split up between four different daemons: the web front-end, the backend, the dispatcher, and the collector. The user interacts with DataMill through a web front-end. Creation of experiment configurations is based on the users' choices (e.g., experiment factors, experiment package, and desired platforms). The backend, which can consist of one or more instances, processes these experiment configurations and schedules individual instances of an experiment, which we refer to as *jobs,* on the workers. The dispatcher submits scheduled jobs to the workers. The collector accumulates results from individual jobs once they are completed on the associated worker. The web-interface provides access to these results as they arrive. In the following sections, we describe the individual components of the management layer in detail. Each of these daemons acts as a simple state-machine as shown in Figure 3. To maintain and ensure consistency of the state of the infrastructure, we have chosen to use a database for central configuration management tasks.

The Web Front-End.

The *web front-end* is the primary window through which the user interacts with DataMill. User submission of experiments triggers the backend daemon process, which processes package information and configurations and subsequently schedules jobs for worker nodes.

The web front-end also houses an XML-RPC [6] service. The service provides an API which workers call to inform the master node about state changes of the individual jobs. Workers use the same API to register with the infrastructure. To separate control and data flows, this XML-RPC interface is only used to notify the master node of state changes. The dispatcher and collector handle transferal of experiment data such as packages and results.

Backend.

The *backend* daemon, shown in Figure 3, is responsible for transforming the user-supplied specifications for a particular experiment into a set of specific trial configurations to run on the worker nodes. The creation of these configurations and the selection of the appropriate worker node to execute them is handled by an optimization solver.

The problem is then to select the minimum number of workers necessary to provide the desired factor coverage. For this, we specified an optimization problem, which the backend solves with GLPK [11] for each submitted experiment. The optimization problem is of the form:

$$w_c = \min \sum_{x_i \in \mathcal{W}} x_i \tag{1a}$$

$$\text{s. to:} \sum_{x_i \in \mathcal{P}_j} x_i \geq l_j, \qquad l_j \in \mathcal{L} \tag{1b}$$

$$l_j \geq x_i, \qquad l_j \in \mathcal{L}, x_i \in \mathcal{P}_j \tag{1c}$$

$$\sum_{l_j \in \mathcal{F}_k} l_j \geq B_k, \qquad k = 1..f \tag{1d}$$

$$x_i \in \{0,1\}, \qquad x_i \in \mathcal{W} \tag{1e}$$

$$l_j \in \{0,1\}, \qquad l_j \in \mathcal{L} \tag{1f}$$

Where $\mathcal{W}$ is the set of workers, $\mathcal{L}$ is the set of factor levels, and $\mathcal{P}_m$ is the set of machines that *provide* level m. $\mathcal{F}_k$ is the set of factor levels for factor k (a subset of $\mathcal{L}$), B_k is the user-requested minimum coverage for factor k. Equation (1b) ensures picking a factor level means at least one machine with that factor level is added to the solution, while, conversely, Equation (1c) ensures that selecting a machine adds all factor levels it provides to the tally, which is checked in Equation (1d) for every factor. This last equation guarantees that all user-provided constraints are met. After a solution is found, each worker is individually selected by their respective x_i variable.

Dispatcher.

The dispatcher daemon in Figure 3 is responsible for transmission and triggering of the execution of the packages and configurations on the appropriate worker as defined by the solution generated by the backend daemon. The dispatcher finds jobs whose associated worker is idle and transmits and calls for the execution of the associated package via secure shell.

Collector.

The collector daemon in Figure 3 is responsible for the collecting of the experiment results from the worker nodes which have completed their current job. Once the results have been collected from any particular worker the worker is marked as *idle* so that it may receive further jobs.

4.2 Worker Node

Each worker node is a separate machine running Gentoo Linux [29] with kernel version 3.3.8, and GCC 4.5.3. Gentoo Linux was chosen since it is a source-based distribution and thus supports a wide variety of architectures. Having the same distribution and tool-chain on all the machines keeps their software homogeneous.

The partition structure of a worker node is shown in Figure 4. The worker comprises two partitions, one with a minimal Gentoo installation, referred to as the *controller*

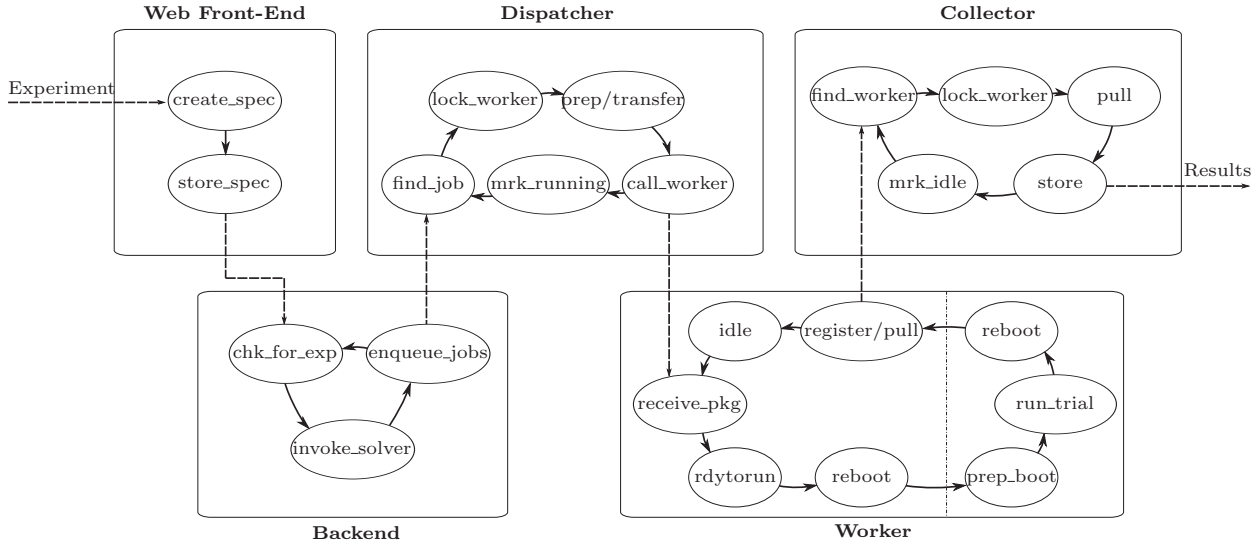


Figure 3: The DataMill Infrastructure

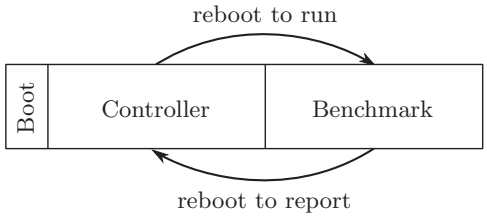


Figure 4: The Worker Node Disk Partitions

partition, which is responsible for communication with the master, and one with a more complete software environment, referred to as the *benchmark partition* on which experiments will run. The state machines for each worker partition are shown in Figure 3, left of the dotted line is the controller partition's state diagram and to the right is the benchmark partition's state diagram.

The master node communicates with workers through secure shell. Communication is kept to a one-way push architecture, from master node to worker node, where possible. The workers do not have the ability to run code on the master barring the limited XML-RPC calls that are provided to them. In this way, the communication between the master node and the worker nodes is strictly limited.

When a worker has been setup for the first time, it must notify the master of its existence. Upon its first boot, the worker collects information about itself and sends it, along with a registration request, to the master node. Once it is registered, it is marked as *idle* in the database and awaits further instruction.

When a worker receives a job from the dispatcher, the dispatcher executes a script on the worker via secure shell. The script transfers the received package to the benchmark partition and prepares it for execution. The machine subsequently reboots into the benchmark partition. The benchmark partition's init process has three tasks to perform: changing the default boot partition, executing the received package, and finally rebooting to the controller partition.

Due to the size of DataMill, worker nodes are expected to fail. The master node detects such occurrences through communication timeouts, incorrect state machine progression, and faulty result files. The master node purposefully inserts redundancies in the job scheduling minimize the impact of worker failures.

The DataMill infrastructure supports remote worker nodes. Already included in the DataMill cluster are several machines from the Purdue University, the University of Pennsylvania, the University of Lugano, and the Federal University of Santa Catarina.

4.3 Factor Variation

DataMill facilitates execution of user supplied benchmarks using a wide variety of experimental setups. Knowledge of the factors affecting performance and the details of the mechanics required to vary them are not necessary to leverage the infrastructure. Nevertheless it is important to discuss some details of DataMill's factor manipulation.

To apply software-controlled factors — compilation flags, library versions, link orders, etc. — the worker node unpacks the configuration file sent by the dispatcher, applying each configuration individually before the setup or execution. Current supported factors include various GCC flags, C and C++ object link orders, the use of address space randomization in the Linux kernel, the addition of padding bytes to the POSIX environment, reboot behavior, and dropping caches before benchmark execution.

While most factors can be easily applied (i.e., interacting with the `proc` filesystem to affect kernel options) some factors require more effort. For example, to support modifications to benchmark link order, we implemented a custom wrapper that intercepts calls to GCC. This wrapper calculates the new order of objects according to the configuration received from the master (i.e., alphabetical, or reverse alphabetical according to object file names), and then forwards the call to GCC, using the new object file order.

This method of applying software factors enables making extensive modifications to benchmarks without requiring users to implement these modifications themselves or to

Algo.	Size (kB)	Reduction (kB)	%
None	51,900	0	100.000
bzip2	13,232	38,668	25.496
XZ	12,120	39,780	23.353

Table 2: Compression Rates for Each Algorithm.

upload a series of different packages with pre-applied modifications. The list of supported factors is currently growing quickly, and we aim to support user-contributed factors in the future.

5. CASE STUDIES

This section presents two case studies: we first perform a compression algorithm performance comparison to demonstrate how easy it is to conduct a performance experiment on DataMill, then we replicate a previously published experiment that revealed performance artifacts related to the link order of a binary, demonstrating the utility of DataMill for the scientific investigation of computer performance.

5.1 XZ vs. bzip2: Best Bang for Your Buck?

XZ [30] and bzip2 [17] are widely-used compression utilities for UNIX-like operating systems. While XZ uses the LZMA2 compression algorithm, bzip2 uses the Burrows-Wheeler algorithm. These two compressors will serve as stand-ins for a "baseline-vs.-proposed-approach" performance comparison, found in the majority of computer science papers that contain empirical performance evaluations. As bzip2 is the older of the two compressors, we will treat it as the baseline.

As described in Section 3, the only preparation step required for this experiment is the creation of two DataMill packages, one for XZ 5.0.4 and another for bzip2 1.0.6. The scripts themselves are omitted for brevity, but their contents are very similar to the ones for Dhrystone and comprise a total of 32 lines. We use the system-wide *emerge* command to install both XZ and bzip2 in order to simplify installation, precluding the need to include their source code in each package.

In addition to the DataMill scripts, the packages contain the data to be compressed. For this experiment, we used the Maximum Compression [35] testset. This testset includes various kinds of files (text, executable, graphics, etc.) and has been used to compare compression algorithms since 2003.

The metrics, which the `collect.sh` script collects, were execution time and compressed file size. Since each compression algorithm leads to a different archive size, the metric used for the comparison is the byte-per-second compression rate, calculated as bytes reduced/execution time. Table 2 shows the uncompressed data size, the resulting archive size under each compressor, the absolute reduction in size, and the resulting archive size as a percentage of the original file size. Note the both compression algorithms use deterministic algorithms, so the resulting compressed files are identical between runs and machines. Machine C, an ARMv7 which uses the ext3 filesystem, reports file sizes 20kB larger than the ones reported by all other machines, which use ext4. This small 0.1% discrepancy was ignored.

If absolute compression is the only metric of interest, the XZ is clearly the winner; however, if execution time or the rate of compression are of interest, then experimentation is necessary. By using DataMill, we can easily compare the two compressors, and measure their susceptibility to different factors. The DataMill experiment space was configured to include all machines, all link orders, all optimization flags, and address randomization on and off. The number of replications was set to 15 to allow the measurement of dispersion. This led to the generation of 6300 jobs, distributed between seven machines.[1] This experiment took approximately five days to complete on the slowest machine, a 600MHz ARMv7 Beagleboard xM. All other machines completed it in less time, and were free to continue with other experiments.

Figure 5 shows an overview of the data set resulting from this experiment. This facet plot is divided by optimization flag (top header) and machine (right-hand header). Machines are indexed with a capital letter, followed by their clock speed and CPU model. Each subplot contains boxplots for each of the compressors, bzip2 and XZ, where the box denotes the median, the lower quartile, and the upper quartile, and the whisker extends to the most extreme data point that is within 150% of the interquartile range of the data set.

The first conclusion is that bzip2 has a better compression rate for all machines under all link orders and all optimization flags tested. This would suggest that, for users interested in compression speed, bzip2 is the better alternative. Also of interest is the fact that bzip2 is unaffected by the optimization flag, which suggests it is entirely I/O-bound. XZ, on the other hand, has a marked performance increase from -O0 to -O1, but negligible differences in performance after that. The omitted experimental dimensions — link order and address randomization — did not significantly affect performance, which is demonstrated by the narrow grouping of all samples in the boxplots of Figure 5 (which contain data from multiple levels in the omitted dimensions).

We use ANOVA [16] to more formally analyze the data. Table 3 shows the ANOVA table for this experiment, fitting a model with up to two-factor interactions on the execution time data for machine F, a 3.4GHz Core i7. The mean square column quantifies the change in execution time attributable to each factor, and the p-value column shows the statistical significance of each factor in respect to execution time. The model was a good fit, with $R^2 > 0.99$ (i.e., the model explains more than 99% of the variability in the data), meaning the model can be confidently used to analyze the data. The table shows that the compressor and optimization flags are significant in isolation ($P < 0.01$), but link order and address randomization are not ($P > 0.05$). Most interestingly, we can prove the interaction between the compressor and optimization flag through their interaction term, which is also significant ($P < 0.01$).

Figure 6 shows the effect of GCC optimization flags on XZ's execution time, with the x-axis ordering the different optimization flags. While optimization flags are not a continuous dimension, this ordering facilitates the visualization of the data and represents the activation of various individual GCC optimization options going from one optimization level to the next. According to GCC's manual, the -Os flag is located between -O1 and -O2 because it activates all options from -O2 that do not increase binary size, being, therefore, a middle point between the two. Data from each worker is

[1]Data for the "alphabetical" link order in the XZ was not generated, as that object order did not link successfully.

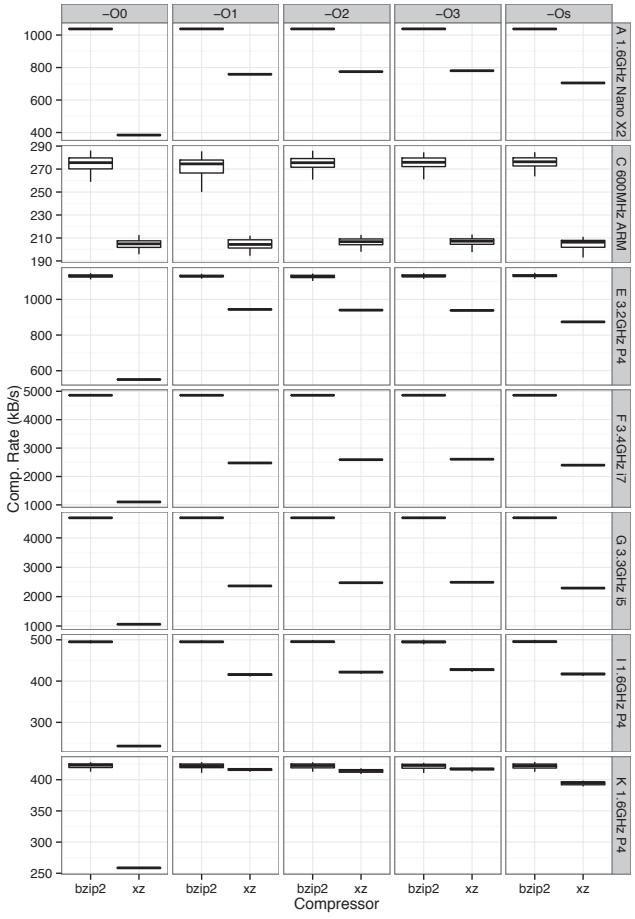


Figure 5: XZ vs. bzip2, Compression Rate

plotted along an individual line, with 95% confidence intervals shown in light gray behind each curve. The plot shows a marked and general improvement in performance going from -O0 to -O1 (as shown in Figure 5), but also reveals a more interesting point: in two of the machines, there is a *decrease* in performance going from -O1 to -O2. Despite being a small decrease, this may merit more investigation, as -O2 is the default optimization flag of several distributions.

This performance comparison demonstrates the utility of DataMill for users interested in evaluating performance: with just 32 lines of code, 6300 jobs were executed in under a week, exercising several dimensions that would normally be ignored, and leading to insight that would be unattainable through manual, one-factor-at-a-time experimentation.

5.2 Perlbench: Link Order Effect

We now demonstrate the use of DataMill for users interested in the study of computer performance evaluation. Mytkowitz et al. [21] report that the link order of a binary is correlated with runtime performance, and that the optimal link order varies from host to host. This is generally understood to be a consequence of different memory and cache layouts leading to different cache and page miss ratios. The authors showed that the performance of Perlbench — part of SPEC CPU2006 [27] — can vary by more than 8% by simply modifying the link order.

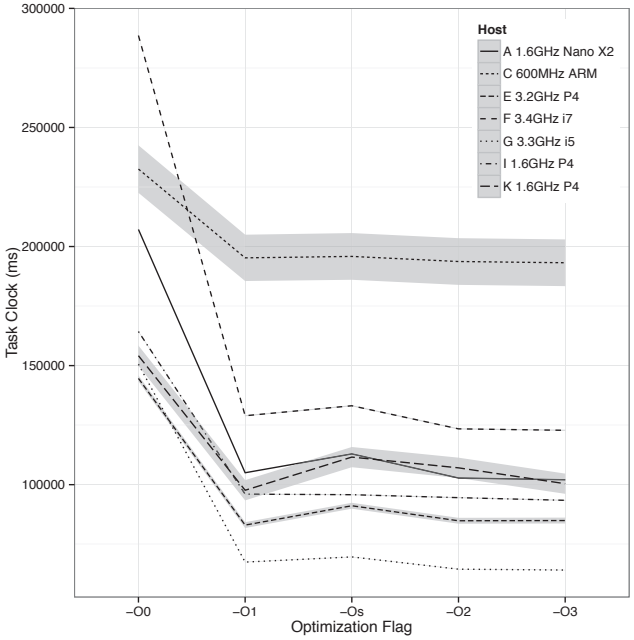


Figure 6: Effect of GCC Optimization Flags on XZ, by Host

Factor	Mean Sq.	p-value
Compressor	1.617e+12	<2e-16
Opt. Flag	1.258e+11	<2e-16
Link Order	7.397e+05	0.5807
Addr. Rand.	2.572e+04	0.8907
Compressor:Opt. Flag	1.883e+11	<2e-16
Compressor:Link Order	1.924e+06	0.2347
Compressor:Addr. Rand.	1.511e+06	0.2922
Opt. Flag:Link Order	1.312e+06	0.4624
Opt. Flag:Addr. Rand.	3.791e+06	0.0257
Link Order:Addr. Rand.	2.738e+05	0.8177
Residuals	1.360e+06	

Table 3: Reduced ANOVA Table for XZ Execution Time on Machine F

Trying to reproduce their results, we created an experiment on DataMill to explore the effect of link order and address randomization on Perlbench performance. We encapsulated Perlbench and SPEC's "train" data set in a DataMill package, with scripts and environment file totaling 33 lines. Three link orders were explored (default, alphabetical and reverse alphabetical), with Linux address randomization on and off. If address randomization is on, one would expect that the affect of link order would be neutralized, since the memory layout will be randomized. In other words, the link order and the address randomization factors should be highly correlated. We chose a number of 15 replications of each configuration to calculate dispersion, generating a total of 630 jobs over 7 machines. The DataMill took approximately 27 hours to finish the full experiment.

Figures 7 and 8 show results for the different metrics for this experiment. These facet plots are divided by address randomization (top header) and host (right header). Each subplot contains three boxplots, one for each link order ex-

Mach.	Factor	Mean Sq.	p-value
A	Link Order	135.77	<2e-16
	Addr. Rand.	0.07	0.0301
	Link Order:Addr. Rand.	0.04	0.0687
	Residuals	0.01	
K	Link Order	1856.5	<2e-16
	Addr. Rand.	1234.6	<2e-16
	Link Order:Addr. Rand.	1864.9	<2e-16
	Residuals	3.3	

Table 4: Reduced ANOVA Table for Perlbench Execution Time on Machines A and K

plored. Figure 7(a) shows execution time, and makes it clear that there is indeed a change in execution time between the different link orders on most cases. An exception to this rule is machine I, a 1.6GHz Pentium 4, where this effect is minor.

It is also clear that the link order effect does not depend on the address randomization feature of Linux being turned off; most machines show different execution times between link orders even when address randomization is turned on. However, machine K (shown in the bottom of Figure 7(a)) shows a link order effect only when address randomization is turned off, contrary to the other machines.

To help understand this effect, Figure 7(b) shows the cache misses for the experiment[2]. This plot shows that there appears to be a correlation between link order and cache misses for most machines, but they do not necessarily mirror the execution time effect seen in Figure 7(a).

Finally, Figure 8 shows that there is no apparent correlation between page faults and link order; even in the case of the Nano X2 and the Xeon machines (top two subplots), the difference in mean page fault counts between link orders is of less than 1%. A possible explanation for this is that the sum of code and data for the benchmark is small enough to fit within a page, no matter the order the object files are linked in.

Table 4 shows the reduced ANOVA table for machines A ($R^2 = 0.977$) and K ($R^2 = 0.855$). This table shows that the link order effect is significant for both of these machines (P < 0.01), but only in machine K do the address randomization factor (P < 0.01) and the interaction between address randomization and link order factors (P < 0.01) play a part. In machine A, both of these are not statistically significant at the 99% confidence level (P > 0.01 in both cases). This suggests that the effect of address randomization, which is present in machine K but not in others, is highly dependent on the machine.

Therefore, while the presence of a cache effect and the lack of a page effect would seem to explain the variations in execution time between link orders, the correlation between link order and execution time still merits more investigation. DataMill is a powerful tool for researchers in performance evaluation, since it allows the systematic variation of correlated factors, such as link order and address randomization, and the simultaneous collection of multiple relevant metrics, such as cache misses and page faults.

[2]Data is missing for the Nano X2 due to the lack of hardware performance counter support.

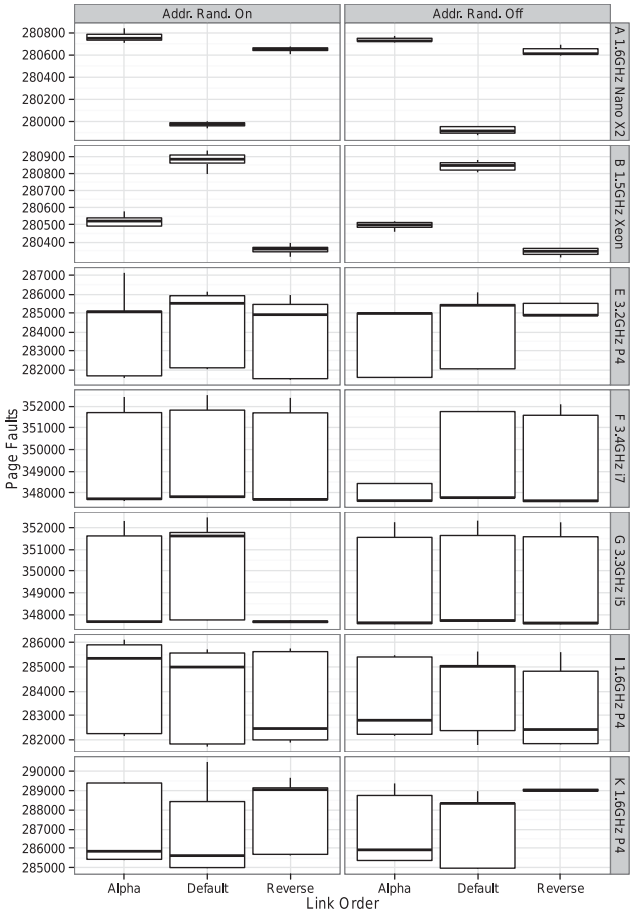


Figure 8: Effect of Link Order on Perlbench Page Faults

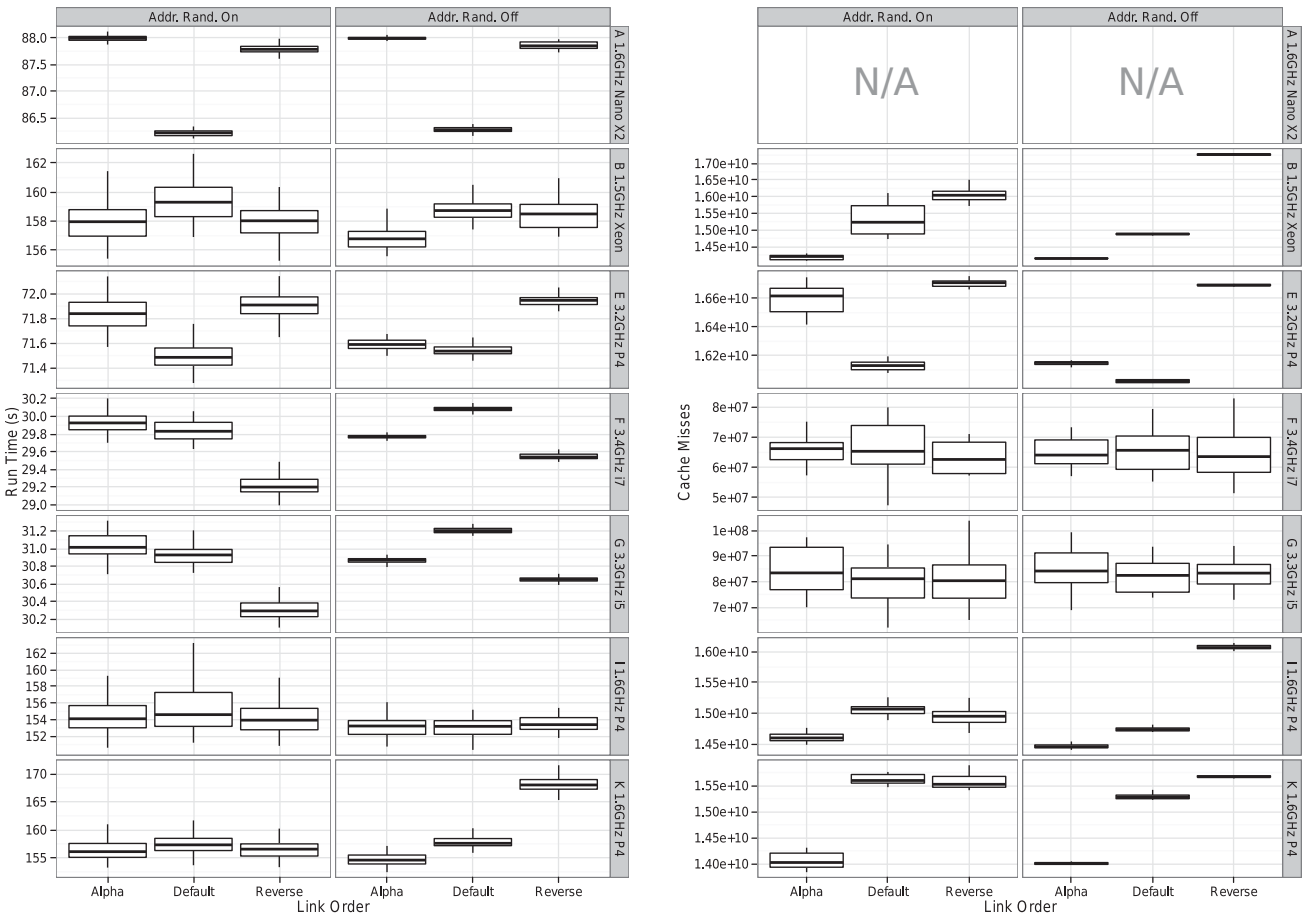
6. LESSONS LEARNED

The implementation of the infrastructure and the execution of the experiments detailed in Section 5 raised several interesting questions.

Debugging packages took considerably longer than expected, especially in the case of Perlbench. This was mainly due to the non-standard build system distributed by SPEC, which requires manual configuration of target architecture parameters. Our experience with it led us to create a virtual worker image on which users can debug their packages. We are currently also investigating a special "debug" experiment type which would execute packages once on each available architecture to ensure it behaves as expected.

The addition of remote nodes (located in remote universities) was a challenge, mainly because of firewalls. Our current implementation uses two-way communication, which requires special treatment for firewalls that reject all incoming connections. In the near future we plan to move to a one-way communication design where the master node is entirely passive, sidestepping this issue and minimizing the effort needed to integrate new workers.

We also realized that even though small embedded targets can run Gentoo, their performance, particularly in compile phases, is prohibitively low for large experiment design



(a) Execution Time

(b) Cache Misses

Figure 7: Effect of Link Order on Perlbench Execution Time and Cache Misses

spaces. In some instances of the compression experiment, we noticed that the compile time exceeded the execution time of the experiment by several orders of magnitude. In the future, we plan to investigate providing cross-compiler support for experiments.

7. RELATED WORK

Various researchers [31, 25, 7, 8, 5, 9] argue for more statistical rigor in computer science. The paper survey by Tichy *et al.* [32] concludes that numerous publications in the field of computer performance evaluation show substantial flaws in experiment design and execution. Vitek and Kalibera [34] report that in PLDI'11, a selective conference where experimental results are commonly published, 39 of the 42 papers that published experimental results did not report a measure of uncertainty in their data, obviating the need for more rigorous statistical analysis in computer science. A survey conducted by Desprez *et al.* [10] draws similar conclusions from a paper survey they conducted. Kalibera and Jones [18] raise a similar point, presenting a random effects model tailored to computer experiments, while also noting that current textbook approaches may be insufficient in the field. Georges et al. [12] argue for the use of statistically rigorous analysis methods, however, their approach is only limited to narrow field of statistical analysis methods.

Mytkowicz *et al.* [21] demonstrate that seemingly innocuous experimental setup details, such as the UNIX environment size or the benchmark link order, can have a significant impact on performance. Harji *et al.* [13] show that the Linux kernel has had a series of performance affecting issues, and that papers that present data measured on Linux could contain incorrect results. Kalibera et al. [19] show that random symbol names generated by a compiler leads to different memory layouts at run time, and, consequently, random variations in performance.

Desprez *et al.* [10] surveys numerous large-scale computing installations that follow a similar objective to our cause; to create reproducible, extensible, applicable, and revisable experiments. The authors provide a survey of experimental methodologies and survey a selection of experimental testbeds. In contrast to our design the testbeds surveyed are often comprised of homogeneous nodes and do not exhibit a lot of variation in the hardware used. While the computing resources are vastly available and the installations provide support for complex experiments (i.e. including distributed systems), the experiment setup is described as a manual arduous process despite exposing high level interfaces.

The infrastructures have been used to implement demonstrators for federated clouds and projects in the field distributed systems. Most notably among the surveyed computing installations is OpenCirrus [4]. In addition to virtualized environments they also provide access to the physical machines. The lowest level service consists of a physical resource set (PRS). A PRS comprises a set of VLAN-isolated compute, storage and network resources. The PRSs are dynamically allocated and managed through a PRS service. Using the PRS paradigm different levels of abstractions can be configured that suit research applications reaching from low level systems research (e.g., the evaluation of OS kernel parameters) to complex distributed systems (e.g., several virtual machines that run a distributed middleware). The reader should note, while the objective of creating create reproducible, extensible, applicable, and revisable experiments are aligned with our cause, the focus of the systems surveyed by Desprez *et al.* [10] is on distributed systems. As a result the hardware infrastructure is decidedly homogeneous, which simplifies the experiment setup and configuration. While testbeds like OpenCirrus [4] support access to low-level hardware features, those features expose little variability compared to the applications we are targeting.

PlanetLab [24, 23] provides planetary-scale data services and is used by the research community to deploy, evaluate and access planetary-scale network services. Planetlab provides so called slivers to its users that consists of distributed networked virtual machines (VMs). The VMs are hosted on physical machines that are maintained in a communal fashion. In order to become a user of PlanetLab, one has to dedicate some servers to PlanetLab. PlanetLab exposes the application interface for provisioning the slivers and has facilities to isolate the network of the individual slivers. PlanetLab is used predominantly to evaluate and deploy distributed system [26], including content-distribution networks, name services, location services, file-streaming services, fault-tolerant scalable services, peer-to-peer networks, distributed anomaly detection, distributed research allocation, routing overlays, and resource discovery.

Because the experiment environment exposed to the user is a virtualized machine, PlanetLab is not an optimal choice for computer performance experiments that seek to evaluate the impact of varying hardware environment factors. This conclusion is consistent with papers listed [26]; none of the publications include computer performance experiments.

Various other experimentation infrastructures have been proposed with similar properties to PlanetLab. Jaffe *et al.* [15] describe a production platform with similar features to PlanetLab. The project links various large data centres in an effort to provide an experimentation platform for distributed systems. Unfortunately, the hardware chosen for the data centres is very homogeneous and does not aid in the exploration of large factor space.

HTCondor [28] is a distributed job scheduler designed for computation-intensive distributed workloads. HTCondor shares some of DataMill's characteristics — such as having users prepare packages and submit them for execution — but since it it geared toward maximizing a cluster's computing throughput, it is not well suited for clean-room performance evaluation.

8. FUTURE WORK

As part of our future work we consider various improvements to the existing system architecture. Huang *et al.* [14] have shown that maintaining security and accountability in a distributed experiment execution framework is challenging. Our current architecture was geared towards providing a proof-of-concept prototype. As part of our future work we want to implement a centrally managed security policy. This would add accountability for individual job executions and transparency to maintainers of remote workers such that the impact of malicious attacks on the infrastructure is minimal.

While our current prototype monitors the performance counters that are exposed by the Linux kernel, it is still up to the submitters of the experiment to analyze their experiment-specific results. In the medium-term we want to provide an application interface for distributed data-analysis tools (i.e., Hadoop [3]), and services (i.e., Amazon Web Services [1]) to facilitate the analysis of experiment data. Our plan is to provide a template engine for various standard benchmarks that can be easily customized by users.

In our current prototype, the factor variation is achieved by a loose collection of shell scripts that are distributed as part of the worker installation. In the future, we want to provide a centrally managed repository of factor variation scripts that can be distributed independently from the worker code, and provide an interface through which users can add custom factors.

9. CONCLUSION

In this paper we introduced DataMill. DataMill provides services to set up and execute robust, replicable, and reproducible experiments. DataMill enables researchers to publish their experiment software, setup, and results. That way experimental results can be reproduced easily. Many aspects of complex performance experimentation are automated by DataMill enabling users to set up performance experiments easily. Due to its support for many different hardware platforms and automated factor variation, DataMill can cover a larger experiment space than typically considered by most researchers. For example, we have shown that a complex performance experiment consisting of 6300 jobs that span various factors can be set up by configuring a package with just 32 lines of shell code. We believe that DataMill can serve as a watermark for experiments conducted for performance-oriented conferences.

10. ACKNOWLEDGMENTS

This research was supported in part by NSERC DG 357121-2008, ORF-RE03-045, ORF-RE04-036, ORF-RE04-039, APCPJ 386797-09, CFI 20314 and CMC, ISOP IS09-06-037, and the industrial partners associated with these projects. The authors would like to thank Andrew Yeung for his work on the ARM platform.

11. REFERENCES

[1] Amazon Web Services LLC. Amazon Web Services. `http://aws.amazon.com/`. Accessed Sep. 17th, 2012.

[2] J. Antony. *Design of Experiments for Engineers and Scientists*. Butterworth-Heinemann, 2003.

[3] Apache Software Foundation. Hadoop. `http://hadoop.apache.org/`. Accessed Sep. 17th, 2012.

[4] R. Campbell, I. Gupta, M. Heath, S. Y. Ko, M. Kozuch, M. Kunze, T. Kwan, K. Lai, H. Y. Lee, M. Lyons, D. Milojicic, D. O'Hallaron, and Y. C. Soh. Open CirrusTMcloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research. In *Proceedings of The 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[5] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young. Computing as a Discipline. *ACM Communications*, 32(1):9–23, Jan. 1989.

[6] D. Winer. XML-RPC Specification. `http://www.xmlrpc.org/spec`. Nov. 1999.

[7] P. J. Denning. ACM President's Letter: What is Experimental Computer Science? *ACM Communications*, 23(10):543–544, Oct. 1980.

[8] P. J. Denning. ACM President's Letter: Performance Analysis: Experimental Computer Science as its Best. *ACM Communications*, 24(11):725–727, Nov. 1981.

[9] P. J. Denning. Is Computer Science Science? *ACM Communications*, 48(4):27–31, Apr. 2005.

[10] F. Desprez, G. Fox, E. Jeannot, K. Keahey, M. Kozuch, D. Margery, P. Neyron, L. Nussbaum, C. Perez, O. Richard, W. Smith, G. von Laszewski, and J. Voeckler. Supporting Experimental Computer Science. Technical report, Argonne National Laboratory Technical Memo, 2012.

[11] Free Software Foundation. GLPK (GNU Linear Programming Kit). `http://www.gnu.org/software/glpk/`. Accessed Sep. 17th, 2012.

[12] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[13] A. S. Harji, P. A. Buhr, and T. Brecht. Our Troubles With Linux and Why You Should Care. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 2:1–2:5, New York, NY, USA, 2011. ACM.

[14] M. Huang, A. Bavier, and L. Peterson. PlanetFlow: Maintaining Accountability for Network Services. *SIGOPS Oper. Syst. Rev.*, 40(1):89–94, Jan. 2006.

[15] E. Jaffe, D. Bickson, and S. Kirkpatrick. Everlab: A Production Platform for Research in Network Experimentation and Computation. In *Proceedings of the 21th Large Installation System Administration Conference*, pages 203–213, 2007.

[16] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing. Wiley, 1991.

[17] Julian Seward. bzip2 and libbzip2. `http://www.bzip.org/`. Accessed Sep. 17th, 2012.

[18] T. Kalibera and R. Jones. Handles Revisited: Optimising Performance and Memory Costs in a Real-Time Collector. In *Proceedings of The International Symposium on Memory Management*, ISMM '11, pages 89–98, New York, NY, USA, 2011. ACM.

[19] T. Kalibera and P. Tuma. Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results. In *Proceedings of the Third European Conference on Formal Methods and Stochastic Models for Performance Evaluation*, EPEW'06, pages 63–77, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] D. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2008.

[21] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! *SIGPLAN Notes*, 44(3):265–276, Mar. 2009.

[22] NLANR/DAST. Iperf. `http://iperf.sourceforge.net/`. Accessed Sep. 17th, 2012.

[23] L. Paterson and T. Roscoe. The Design Principles of PlanetLab. *Operating Systems Review*, 40(1):11–16, January 2006.

[24] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proceedings of The 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[25] L. Peterson and V. S. Pai. Experience-Driven Experimental Systems Research. *ACM Communications*, 50(11):38–44, 2007.

[26] PlanetLab. PlanetLab Bibliography. `http://www.planet-lab.org/biblio` visited 2012-09-28.

[27] Standard Performance Evaluation Corporation. SPEC CPU2006. `http://www.spec.org/cpu2006/`. Accessed Sep. 17th, 2012.

[28] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In T. Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[29] The Gentoo Foundation. Gentoo Linux. `http://www.gentoo.org/`. Accessed Oct. 5th, 2012.

[30] The Tukaani Project. XZ Utils. `http://tukaani.org/xz/`. Accessed Sep. 17th, 2012.

[31] W. F. Tichy. Should Computer Scientists Experiment More? *IEEE Computer*, 31(5):32–40, 1998.

[32] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Systems Software*, 28:9–18, 1995.

[33] Vince Weaver. Perf Event Overhead Measurements. `http://web.eecs.utk.edu/~vweaver1/projects/perf-events/benchmarks/rdtsc_overhead/`. Accessed Sep. 17th, 2012.

[34] J. Vitek and T. Kalibera. Repeatability, Reproducibility, and Rigor in Systems Research. In *Proceedings of The Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 33–38, New York, NY, USA, 2011. ACM.

[35] W. Bergmans. Maximum Compression. `http://www.maximumcompression.com/data/files/index.html`. Accessed Sep. 17th, 2012.

[36] R. P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *ACM Communications*, 27(10):1013–1030, Oct. 1984.