SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2015; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

DataMill: A Distributed Heterogeneous Infrastructure for Robust Experimentation

J. C. Petkovich*, A. Oliveira[1], Y. Zhang[2], T. Reidemeister[1], S. Fischmeister[1]

[1] *Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, N2L 3G1, Canada*
[2] *School of Computer Science, University of Waterloo, Waterloo, ON, N2L 3G1, Canada*

SUMMARY

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. How can one trust any reported empirical analysis of a new idea or concept in computer science?
DataMill is a community-based services-oriented open benchmarking infrastructure for rigorous 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.
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.
This paper discusses experience earned through creating and using an open benchmarking infrastructure. Multiple research groups participate and have used DataMill. Furthermore, DataMill has been used for a performance competition at the International Conference on Runtime Verification (RV) 2014, and is currently hosting the RV 2015 competition. This paper includes a summary of our experience hosting the first RV competition.

Received . . .

KEY WORDS: DataMill; performance; experimentation; infrastructure; robustness; repeatability

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 of new technologies [1]. 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 [2, 3, 4, 5] has demonstrated that experimental evaluation in computer science is difficult. As a consequence, experiment design, setup, analysis, and data

*Correspondence to: Jean-Christophe Petkovich, Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, N2L 3G1, Canada. Email: j2petkov@uwaterloo.ca

Contract/grant sponsor: CMC ISOP; contract/grant number: IS09-06-037

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 [6]. The lack of rigorous performance evaluation is often blamed on the fact that experimentation is difficult and costly [4], but those obstacles are not insurmountable.

To improve the current state of experimentation, we proposed DataMill [7, 8]. 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 design experiments and evaluate 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 and configure the experiment while simultaneously increasing the robustness and applicability of the experiments results to a wide range of factor levels. The user need not know the details of the underlying mechanisms required to vary these factors and may simply take advantage of DataMill's 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. Over the past year, DataMill executed 641 experiments and 53 530 jobs. A total of 908 days of computing time were spent running scripts submitted by users in experiment packages.

Besides easing experimentation effort and cost for 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 a distributed infrastructure for computer performance evaluation spanning multiple universities; (2) Revisiting design rationals on the architecture and implementation with over two years of benchmarking experience and hosting a benchmarking competition; (3) The reproduction and expansion of a previously published experiment, confirming the existence of significant hidden factors; (4) A survey of current performance evaluation practices in computer science publications, demonstrating the need for more rigorous experimentation; and (5) A summary of our experiences and insights facilitating a performance competition on the DataMill infrastructure.

BACKGROUND

In traditional experimental design [9, 10], *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 in [7]. We have updated the survey with more recent conferences, including HiPC 2013, and ASPLOS 2014, the results are shown in Table I. The columns, in order, show the total number of papers, the fraction that contains empirical performance evaluation, the

DOI: 10.1002/spe

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%
ASPLOS'14	52	92%	75%	46%	46%	4%	29%	52%	54%	83%	0%
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%
HiPC'13	49	80%	92%	59%	95%	0%	80%	13%	0%	0%	0%

Table I. Illustration of Experimental Rigor in Recent Scientific Publications

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 [4, 2, 5] in the past, a number of papers that do a performance evaluation do not use any measure of dispersion in their performance evaluation. With the conferences included in our survey, we cannot completely isolate the effect of time on experimentation practices, although we can get close with ASPLOS 2011/2014, where we see a substantial increase in published experimentation software. Kalibera and Jones [11, 12] also report that many experiments from ASPLOS, ISMM, PLDI, TACO, and TOPLAS are not repeatable or do not give enough evidence to support performance improvement claims.

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 [13], a network performance benchmark, repeatedly running on a loopback interface (the machine was disconnected from the network) in experiment blocks of one hour. The x-axis shows the blocks, and the y-axis, which does not start at zero for space reasons, shows the bandwidth through the loopback interface. For example, the boxplot graphic at block 6 comprises all of the iperf data collected during the 6th hour of the experiment. 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*. To this date we have been unable to find an evidence supported explanation for the phenomenon. Although no cron daemon was running on the machine, because the performance change appears to be roughly periodic, and occurs every 24 hours, we can theorize that it has to do with the time of day. 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 [14, 15, 6, 16, 17]. 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

DOI: 10.1002/spe

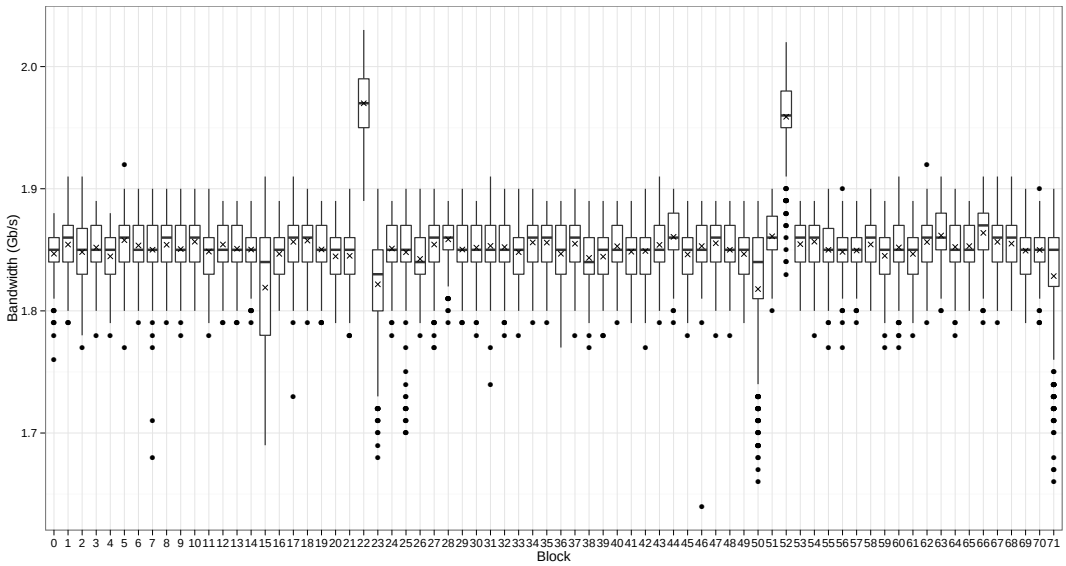


Figure 1. Iperf Performance in Blocks of 1 Hour

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. DataMill collects several performance counters available in the Linux environment and computes dispersion figures automatically. DataMill supports 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. 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 [2].

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 each individual experiment setup, allowing automated experiment reproduction. DataMill achieves robustness by varying experimental setups across a wide set of factors, including variation of state-of-the-art known hidden

DOI: 10.1002/spe

factors. DataMill is a public-service infrastructure and available to scientists around world as long as they donate equipment for evaluations themselves.

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.

Packages

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

Each package contains,

1. the source code and any input data for the experiment,
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 collect them via compressed archives. These simple features permit performance evaluation of a wide variety 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`. Beyond these simple requirements, the package may contain anything else the user desires.

To exemplify the construction of a DataMill package, consider benchmarking bzip2 [18]. We run Gentoo's emerge command to compile and install bzip2 from source. Our compiler wrapper redirects calls to GCC during the compile phase to apply various compiler factors discussed in Section Pluggable Software Factors. While settings for these factors are set through the web interface, all that is required in the package setup script is show in Listing 1.

```
1 #!/bin/sh
2
3 emerge bzip2
```

Listing 1: Setup Script

Listing 2 shows the `run.sh` script that records performance data. First, the script changes into the directory bzip2 which contains files with various binary patterns for a compression benchmark. Next, a bzip2 archive of the files in the bzip2 directory is created and timed. The output of the time command is recorded into the bzip2-results file. Finally, the size of the bzip2 archive is appended to bzip2-results.

```
1 #!/bin/sh
2
3 cd bzip2/
4 (time tar -c --bzip2 -f tmp.tar.bz2 A10.jpg AcroRd32.exe \
5 FP.LOG FlashMX.pdf MSO97.DLL english.dic ohs.doc rafale.bmp \
6 vcfiu.hlp world95.txt) >> bzip2-results 2>&1
7 (echo size `du tmp.tar.bz2 | cut -f 1`) >> bzip2-results 2>&1
```

Listing 2: Run Script

DOI: 10.1002/spe

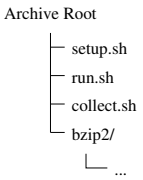


Figure 2. Bzip2 Package Directory Structure

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=/bzip2-results.tar.gz
4 tar czf $results /bzip2/bzip2-results > /dev/null 2>&1
5 echo $results
```

Listing 3: Collect Script

The user must package the experiment source code, input data, and DataMill scripts in a GZipped TAR file. All DataMill scripts should be in the root directory of this archive. For the Bzip2 package, the file structure of the package would be as shown in Figure 2. The `bzip2` folder contains various files to be compressed in `run.sh`.

To facilitate package creation, DataMill provides 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. Experiments submitted to the infrastructure are extensively logged in a non-intrusive way, and results are reported to users through the DataMill web interface.

Experiment Creation

Once the package(s) 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.

In the experiment space definition step, users select which experimental setups they wish to explore. The interface presents each dimension of the experiment space, divided in hardware and software categories. Hardware factors are adjusted with sliders, and levels for software factors are selected with checkboxes. With sliders, users can select a range for factors such as the amount of memory on the machines for the experiment. Our user interface dynamically updates the visible list of available workers as the user adjusts sliders and checkboxes for these factors. Showing a list of eligible workers based on the factor selection gives the user an idea of the variety of machines used for their experiment. Our user interface gives the user an overview of the workers at a glance, showing important worker information through a tooltip as in Figure 3. Besides the pool of workers, DataMill also calculates the number of jobs for the experiment as factors are changed. This provides the user with an estimate of the queue priority for their experiment and for how long it might take. By default, the sliders for hardware factors are set to cover the entire range. However for software

DOI: 10.1002/spe

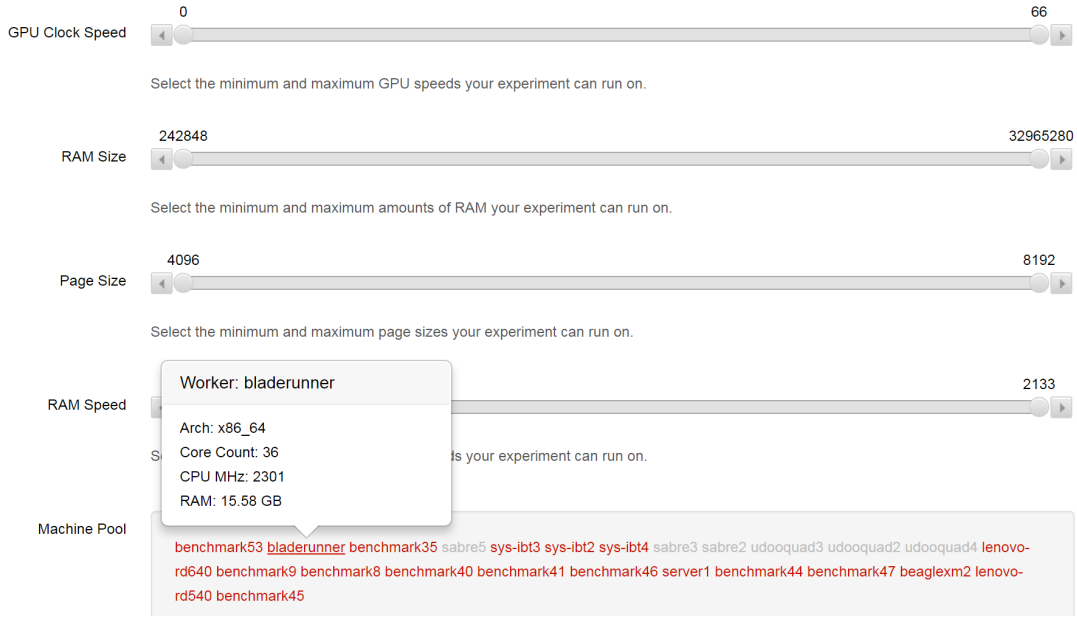


Figure 3. Tooltip for a Worker on Experiment Create Page

factors, only the default levels are selected. If the user wishes to explore the effect of a software factor, he or she must select additional levels.

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 Space Layout Randomization (ASLR) feature of Linux ("on" or "off") should be tested, then the number of jobs grows to 10 (5 flags times 2 ASLR 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.

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 as they are produced. The web interface dynamically updates experiment information as data arrives, allowing users to monitor their experiment's progress. This page also lists the jobs for the experiment and their status as shown in Figure 4. During experiment execution, partial results files are available for download.

In addition to the data collected by the collect script, DataMill collects additional metrics with minimal overhead [19], such as total execution time, and the number page faults and cache misses. Where it makes sense, data that DataMill collects is automatically plotted and made available to the user in a dynamically updated set of graphs. An example graph produced by DataMill for the time taken to run the `collect.sh` script for an experiment is shown in Figure 5. The y-axis is the time measured in minutes for collect.sh to execute, and the names on the x-axis are the worker hostnames for two of the machines included in the experiment. On hovering over a box and whisker plot, the values for the quartiles of a worker are displayed. The horizontal line across the graph marks the average of all collect times.

Once all jobs associated with an experiment are finished, users can download the full experiment results file. This results file contains individual results from every job. Examples of how to analyze large datasets generated by DataMill are provided in Section Case Studies.

DOI: 10.1002/spe

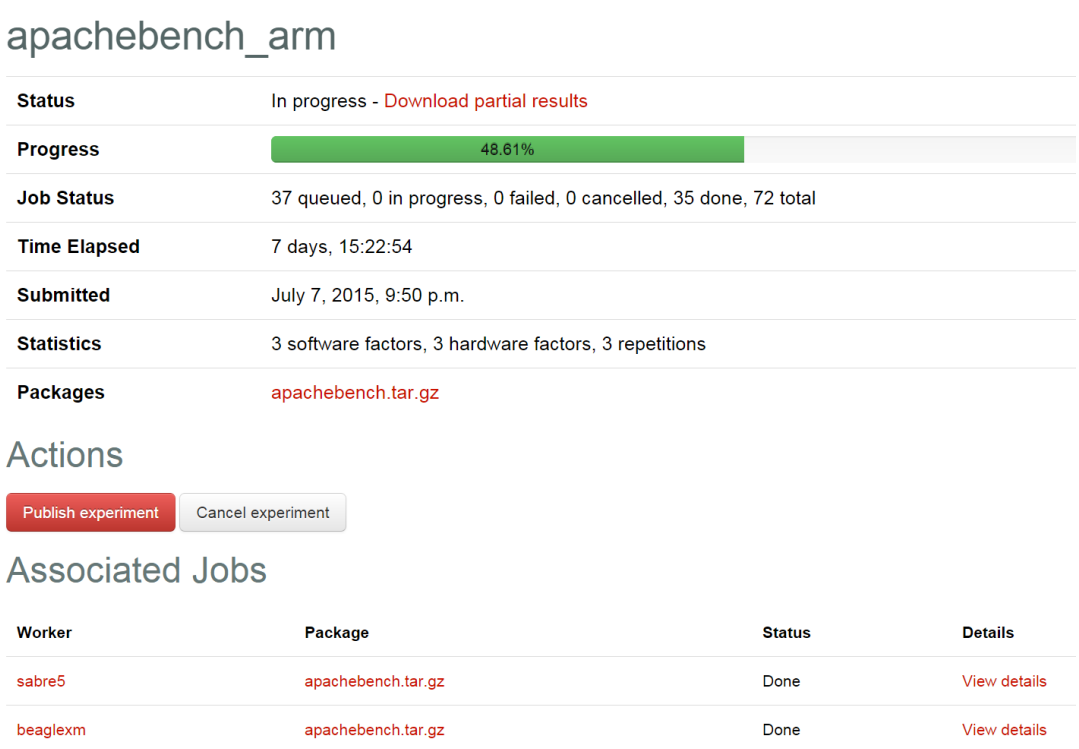


Figure 4. Example Experiment Details Page While an Experiment is in Progress

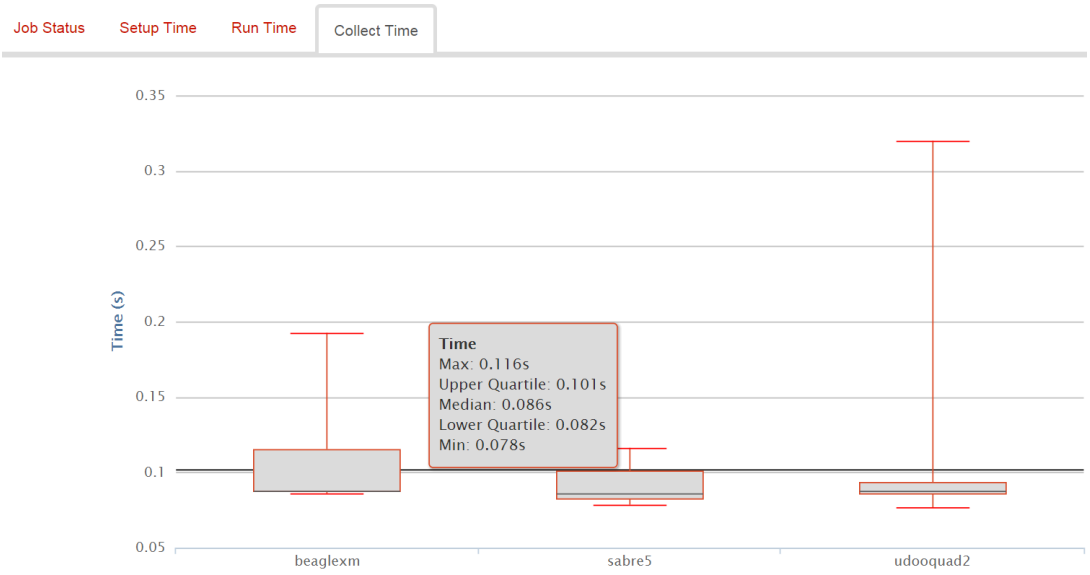


Figure 5. Collect Time of an Experiment on Three Workers

Inevitably, jobs will occasionally fail for many possible reasons. Despite the ability to test user packages on a DataMill worker prototype, subtle differences between architectures can cause a plethora of different issues, especially during dependency installation. Debugging problems that occur on remote machines can be extremely difficult, but DataMill equips users as best it can with the logging information that it provides. The output of the `setup.sh`, `run.sh`, and

DOI: 10.1002/spe

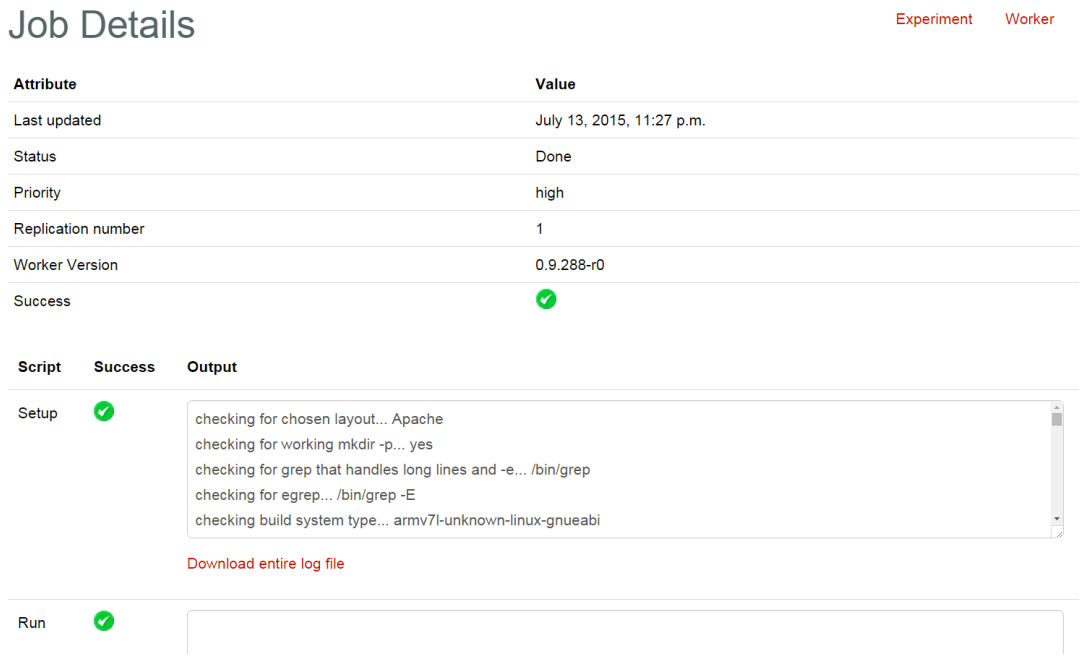


Figure 6. Example Job Details Page

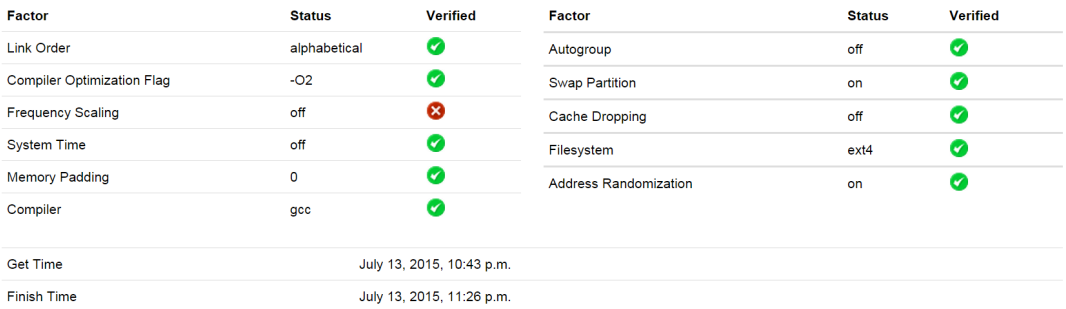


Figure 7. Example Job Factor Verification

`collect.sh` scripts are logged on the job details page as displayed in Figure 6. In addition to the data collected from a given experiment, DataMill provides users with extensive logging of performance-insensitive portions of their experiment to ease debugging effort when something goes wrong. For example, the success of setting each factor to a level is logged with each job as in Figure 7. In this case, frequency scaling could not be turned off on an ARM board. All the other factors were set successfully. This kind of logging affords the user with some assurances when they examine their results that the experiment's trials executed as they expected. Individual failed jobs and their logs and meta-data are also displayed the web interface. This permits convenient and simple debugging of the majority of problems encountered during package development, testing, and execution.

DATAMILL: THE INFRASTRUCTURE

Making the user experience described in Section DataMill: The User Experience 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*

DOI: 10.1002/spe

nodes, which execute the experiment packages provided by the users. The original design and implementation of DataMill is reported in [7], but a great deal of the architecture has changed in response to several challenges that we have encountered through heavy use of the infrastructure. This section includes the highlights of the changes since the previous iteration of DataMill.

Reliability Through Factor Variation

DataMill has facilitated producing reliable results through automatic and controlled variation of heterogeneity in user's experiments. Although DataMill optionally provides a set of factors which can be varied out-of-the-box for any given user's experiment, it is crucial that we allow users to vary factors that matter specifically to them and their experiments, even when they are outside of the out-of-the-box factors DataMill provides.

DataMill has a generic plugin system that users can leverage to support a variety of interesting software factors. DataMill supports this functionality by separating user's packages into three core tasks, experiment setup, experiment execution, and data collection. These are encapsulated in the three experiment package scripts: `setup.sh`, `run.sh`, and `collect.sh`. By differentiating between three different basic experimentation tasks, DataMill can instrument and manipulate the environment unobtrusively during different phases of the experiment, from simple manipulation of environment variables, to swapping out the compiler used by the package's build system. DataMill provides hooks for user-developed factors to run arbitrary code before and after each of the three core phases of an experiment package.

Besides software factors, DataMill also provides heterogeneity by supporting not only new and powerful computer hardware, but also much older legacy systems. Table VII shows a list of the hardware of devices currently in DataMill. Repeated rows indicate that there is more than one of the same device available for running benchmarks. Maintaining and supporting those systems is a difficult challenge. Toward supporting a variety of different hardware, the DataMill worker software is necessarily simple in its requirements. The most basic requirements for the worker software is the ability to set the root filesystem for the next reboot through software. DataMill has a simple library that sniffs for the current boot-loader and adjusts the boot-loader configuration as necessary. Other similar testing infrastructures, such as Linaro Automated Validation Architecture (LAVA) [20] only support platforms compatible with a particular boot-loader, and cannot support systems which cannot boot from a remote image. Several worker features necessary for the out-of-the-box factors that DataMill supports are machine dependent, and DataMill automatically tests workers for support for these features when they boot. Experiments that require those features will never be scheduled on workers that do not support them.

Keeping Machines Consistent

Each worker node is a separate machine running Gentoo Linux with kernel version 3.18.11, and GCC 4.8.3. Gentoo Linux was chosen since it is a source-based distribution which offers several distinct advantages for DataMill. Gentoo Linux has official support a wide variety of architectures, nearly any architecture which has support for GCC will also support running Gentoo Linux. Having the same distribution and tool-chain on all the machines keeps their software as homogeneous as possible.

DataMill store the details of package versions for each experiment executed on our worker nodes so that users can treat them as a factor during analysis if necessary. We also intend to use this data to perform best-effort rollbacks for reproductions of previous experiments. Using a source-based distribution allows us to avoid typical problems with package rollbacks easily, Gentoo provides several tools for automatically detecting and resolving broken link-level dependencies which would otherwise force us to rollback library versions and package versions in lockstep. This allows greater isolation of performance problems, particularly when software libraries are concerned. Although Gentoo is a source-based distribution, it does support binary packages as well. The master node acts as a binary package mirror for the common base set of packages that are included or expected to be used by the worker nodes, avoiding recompiling most packages redundantly.

Since DataMill workers run a source-based distribution, package updates and dependency resolution during experiment execution can require workers to compile and link software projects.

DOI: 10.1002/spe

Some of our worker platforms are resource constrained, and do not meet the requirements of compiling large and complex projects such as GCC. Or, if they were to build these projects, would take a disproportionately long time to do so. The DataMill master node houses several distcc [21] virtual machine servers and caches compiled object files, permitting resource constrained workers access to hardware capable satisfying the requirements of demanding software projects. With this infrastructure in place, resource constrained platforms such as the raspberry pi can effectively serve as DataMill worker nodes.

Tolerating Failures

Worker nodes are expected to fail. We have paid special care to the design of the worker software main-loop to ensure that failures are handled carefully. The primary objective of our failure handling procedures is to ensure that any results produced by an experiment trial are both valid and complete. Any experiment trial that does not proceed precisely as expected will still report any obtained results, but the trial will be marked as a failure. The failures are reported to users, indicating that associated results are to be analyzed with appropriate skepticism.

To respond to different possible failures caused either by poorly written experiment packages, or unforeseen circumstances, workers are capable of some level of self-repair. Any incorrect or unexpected states reached during experiment progression that are repeated for a certain time threshold are escalated. Since these failures are more symptomatic of a systematic failure, rather than a temporary or user initiated failure, they are escalated and responded to by detailed log and diagnostic reports followed by deletion of all persistent state on that worker.

Debugging Experiments

Developing experiment packages that will run unaided and unguided on a remote machine is challenging. DataMill's answer is to fail as noisily as possible, producing logs of as much of the non-performance-sensitive portions of user's packages as possible, and providing access to these logs from the DataMill web interface.

Although we make debugging remote experiment package execution as simple as possible, the turn around time for these tests still leaves much to be desired. We created a virtual worker image that is a fully capable DataMill worker node, on which users can test their packages on a local virtual machine, and observe and modify their experiment interactively. In this way, they do not need to spend precious worker node execution time on debugging, and can get the majority of their experiment's bugs solved with a much smaller turnaround time. Users have reported that this has greatly improved their overall experience with the infrastructure.

Mitigating Long Execution Times

Once an experiment package is finished and working, it can still take weeks for a full experiment to complete is a large set of factors are explored. This is because with each additional factor and factor level, the size of the experiment space increases combinatorially. DataMill attempts to and minimize the length of user's experiments by using an optimization solver (see [7] for implementation details) to reduce the set of trials down to what is necessary for estimating the factor effect sizes of interest to the user, as in typical factorial experiment design [10].

Remote Workers and Network Access

Supporting remote workers proved more challenging than originally anticipated as differing network management policies hampered our original push-style architecture. The master node does not initiate any communication with workers, it is entirely passive. Previously it was active, and triggered job execution on workers through secure shell, but this proved problematic when registering remote workers where network policies were restrictive, and secure shell was not permitted. We solved this problem by discarding our original Secure-Shell-based infrastructure with an entirely passive centralized authority. The master node houses a simple dead-drop API that allows workers to download their next available task, and to upload their results or error logs, or notifications of failure or other problems. In this way, the communication between the master node

DOI: 10.1002/spe

and the worker nodes is strictly limited, and conforms with most network management policies implemented by university and business networks.

FACTORS: ARCHITECTURE AND IMPLEMENTATION

DataMill facilitates execution of user supplied benchmarks using a wide variety of experimental setups. Knowledge of factors that can influence performance and the details of the underlying mechanics required to vary them are not necessary to leverage the infrastructure. DataMill currently supports over twenty factors, and facilitates user contribution of factors through a plugin system. These factors are categorized into software and hardware factors.

When an experiment is created, only workers which satisfy the selected hardware constraints are scheduled for the experiment. For instance, if a user selects 512 MB to 1 GB of RAM for their experiment the scheduler is limited to machines that satisfy the memory constraint. This may come at the expense of also limiting the number of CPU cores since most of our machines with more than 2 CPU cores have more than 1 GB of RAM. Hence, DataMill shows a machine pool of eligible workers with tooltips. DataMill currently supports SPARC, ARM, Intel x86, and Intel x86-64 architectures.

While hardware factors are applied during the scheduling stage, settings for software factors are serialized and transferred to each worker along with the associated package. The worker nodes unpack the configuration files sent by the scheduler, applying each configuration individually before setup or execution. Software factors supported by DataMill include compilation flags, environment paddings, and link orders. While some factors can be applied through the use of the `proc` filesystem, other factors require a more complex implementation.

Pluggable Software Factors

Our factor plugin architecture eases the addition of new factors in two ways:

1. Automatic detection and reporting of factor level support by workers, and
2. A simple plugin format and API requiring users to implement only a simple set of methods.

All DataMill factors are implemented using the plugin API. Only three methods — for setting, reading, and testing a factor's level — and a constant holding the possible factor states are required for factor plugin implementations. From this small API, a factor can be configured and tested on worker nodes during experiment execution, or during factor level support detection, where workers report their support for particular factors. The reported factor level support is used to schedule experiments appropriately based on a user's experiment's factor configuration.

Besides the minimum required methods for a factor implementation, factors support several hooks for manipulating an experiments environment: prepopulate, preboot, presetup, prerun, and precollect. The prepopulate hook is executed before populating the benchmark partition, and the preboot hook is executed before booting into the benchmark partition. The last three correspond to the three user submitted scripts.

Several new factors are supported by DataMill since the original report on the infrastructure [7]. New factors in DataMill version 2.0 include support for altering the underlying filesystem type, several compiler wrapper enhancements, support for the autogroup kernel parameter, support for CPU frequency scaling, and support for altering system time.

DataMill supports modifying the underlying filesystem type during experiment execution. It is well known that differing file system types have differing IO performance [22]. For every benchmark job executed on a machine, DataMill reformats the benchmark partition with a particular filesystem before installing Linux and the DataMill worker software for the upcoming experiment trial. Currently DataMill supports btrfs, ext3, and ext4. One possible improvement to our current approach is to take into account the disk size when testing filesystems. For example, btrfs turns on the extrefs feature for disks of sufficient size which may impact performance.

To support modifications to binary link order and compiler optimization level, we implemented a custom wrapper that intercepts calls to GCC/clang. This wrapper calculates the new order of objects

DOI: 10.1002/spe

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 the relevant compiler using the new object file order. Similarly it intercepts compiler flags that it recognizes and adjusts them based on the user's preference, all other factors are passed through without modification.

The DataMill compiler wrapper also supports compiling programs as Position-Independent-Executables by taking advantage of the -fpie flag new in GCC 3.4. It allows executables to behave like a shared libraries so that their base addresses are relocatable [23]. Since -fpie is only available for GCC, only the GCC wrapper class was modified to add the flag when the Position-Independent-Executable factor is turned on. For both GCC and clang, the -fpie flag is removed from the user's list of flags when calls to them are intercepted through the compiler wrapper.

When writing a new compiler-based factor, the existing compiler wrapper for GCC or clang must be modified. This is an exception to our one file per factor plugin architecture. However, this structure is necessary so that the compiler wrapper intercepts calls to GCC and clang transparently. For a compiler-based factor, the factor plugin sets the factor level which is read by the compiler wrapper on calls to GCC and clang during the execution of the experiment. If the user simply wishes to use a custom CFLAG that is currently not supported as a factor, the flags appended to GCC and clang calls are passed to the compiler. General-purpose transformations that are not already implemented as factors are applied in the same way as on the user's local computer.

Since version 2.0, the DataMill compiler wrapper transparently supports different compilers during experiment execution. All compilers on the system point to our compiler wrapper via symlinks, although the compiler wrapper can simply pass-through calls, or users can opt out of the compiler wrapper entirely. For instance if both GCC and clang were selected for compilers, calls to GCC would be redirected to clang in one experiment run and to GCC in another.

Besides compiler factors, DataMill also supports kernel parameters such as autogroup, which is a scheduling policy that aims to reduce scheduling latency on the desktop. It groups tasks by the terminal to improve responsiveness on an interactive system. On servers which fork processes, disabling the autogroup feature tends to keep child processes from migrating. When an experiment involves multiple concurrent processes, disabling and enabling autogroup may reveal a performance effect on multi-core systems [24]. To implement the autogroup feature, autogroup scheduling is compiled into the kernel and a boolean value in the proc filesystem is toggled to enable and disable the autogroup scheduler.

Similarly, changing the frequency scaling governor often has an impact on performance. Currently DataMill supports the "performance" and "ondemand" schedulers in the frequency scaling factor. This factor implemented by calling cpufreq-set to set the governor. During installation, the kernel is configured to enable CPU frequency scaling and the performance and ondemand governors are built into the kernel. We chose these two governors to represent "on" and "off" because performance keeps the CPU frequency at maximum while ondemand scales the CPU frequency levels.

Another factor of interest is the effect of the system time on performance. Before running a benchmark, the time is set to 2:00pm and the time of the system is saved. After running a benchmark, the system time is restored. The time elapsed is calculated and added to the time saved before running the benchmark.

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.

DOI: 10.1002/spe

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 II. Compression Rates for Each Algorithm.

XZ vs. bzip2: Best Bang for Your Buck?

XZ [25] and bzip2 [18] 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 DataMill: The User Experience, 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 included in Section DataMill: The User Experience 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 [26] 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 II 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 that 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, then 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 ASLR "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.[†] 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 8 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

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

DOI: 10.1002/spe

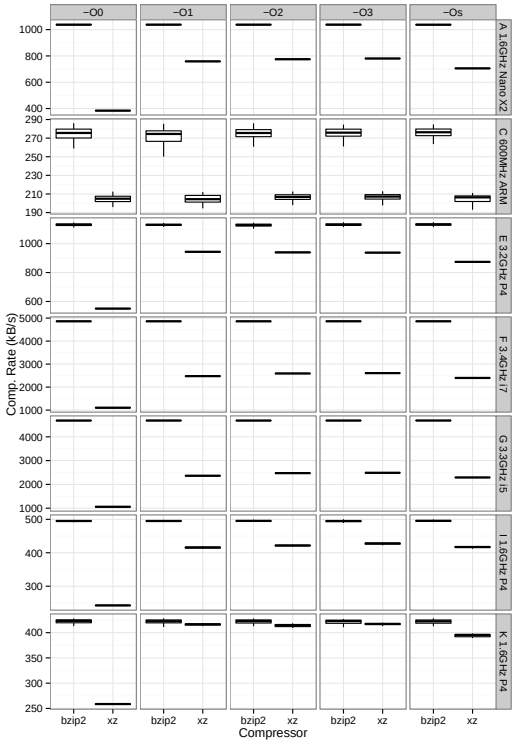


Figure 8. XZ vs. bzip2, Compression Rate

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 ASLR — did not significantly affect performance, which is demonstrated by the narrow grouping of all samples in the boxplots of Figure 8 (which contain data from multiple levels in the omitted dimensions).

We use ANOVA [27] to more formally analyze the data. ANOVA is a special-case of the F-test for differences in variance, and allows us to separate the influence of different factors on variance into first-order effects and interaction effects, and allows us to understand whether or not factors have both a significant and substantial effect on our metrics of interest, compression rate, and execution time in seconds. Although we are measuring performance on a wide variety of machines, we do not include them as factors in our models. Since we expect different machines to behave differently, and since it is impractical to isolate hardware internal to machines (e.g., producing otherwise identical machines with differing amounts of memory, or differing CPU speeds), we analyze results from different machines seperately. We conduct a multi-way ANOVA on the remaining factors, compression type, GCC optimization flag, link order, and ASLR, and their second order interactions. We set the sensitivity of our tests to a confidence level of 95% with a Bonferroni correction since we block against machine ($\alpha = 0.05/7 = 0.007$). Prior to computing the ANOVA table, we verified the homoscedasticity of the data within each experiment block through a widely accepted rule of thumb, that the largest standard deviation is less than twice the smallest standard deviation. After computing the ANOVA table for each experiment block, model residuals were examined by hand using Q-Q plots, each was found to be a reasonable approximation of the normal distribution.

Table III shows the ANOVA table in 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 tables computed for the remaining machines are similar, as are the conclusions we were able to draw from their analysis. We are careful to isolate our conclusions to the machine and experiment from which they were drawn, as we are testing the same hypothesis on many different machines. Because of the

DOI: 10.1002/spe

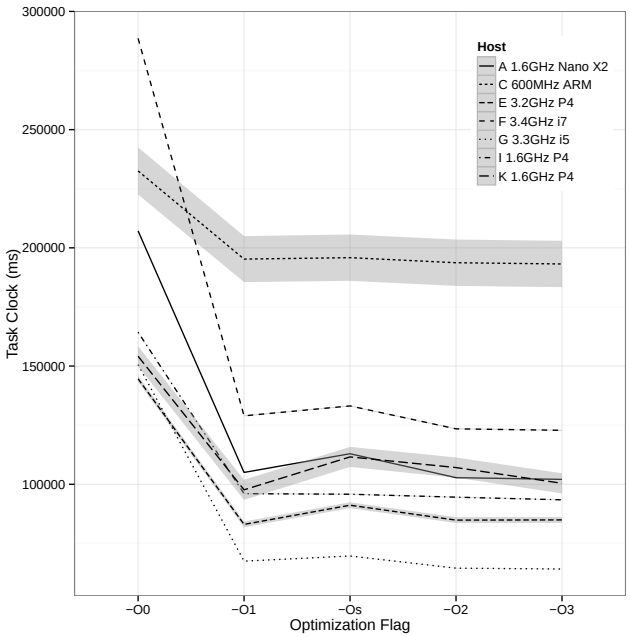


Figure 9. Effect of GCC Optimization Flags on XZ, by Host

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 III. Reduced ANOVA Table for XZ Execution Time on Machine F

isolation of our conclusions, we are subject to the multiple comparisons problem, and as such we conduct a Bonferroni correction. 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 for machine F, the compressor and optimization flags are significant in isolation (P <0.007), but link order and ASLR are not (P >0.007). Most interestingly, we can prove the interaction between the compressor and optimization flag through their interaction term, which is also significant (P <0.007).

Figure 9 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

DOI: 10.1002/spe

-O2 that do not increase binary size, being, therefore, a middle point between the two. Data from each worker is 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 8), 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.

Perlbench: Link Order Effect

We now demonstrate the use of DataMill for users interested in the study of computer performance evaluation. Mytkowitz et al. [6] report that the link order, the order in which object files are included in the linking process, 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 [28] — can vary by more than 8% by simply modifying the link order.

Trying to reproduce their results, we created an experiment on DataMill to explore the effect of link order and ASLR on Perlbench performance. We encapsulated Perlbench and SPEC's "train" data set in a DataMill package, with scripts and environment file totaling 33 lines. As a result of time constraints, only three link orders (default, alphabetical and reverse alphabetical) were explored rather than the full 33 tested by Mytkowitz. Testing the full set of link orders would have extended the length of the experiment to approximately 12 days. On top of these factors, we also varied Linux's ASLR feature. If ASLR is on, one would expect that the effect of link order would be affected, since ASLR shifts the start point of the region where shared objects are placed in memory. In other words, the link order and the ASLR 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 10 and 11 show results for the different metrics for this experiment. These facet plots are divided by ASLR (top header) and host (right header). Each subplot contains three boxplots, one for each link order explored. Figure 10(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 ASLR feature of Linux being turned off; most machines show different execution times between link orders even when ASLR is turned on. However, machine K (shown in the bottom of Figure 10(a)) shows a link order effect only when ASLR is turned off, contrary to the other machines.

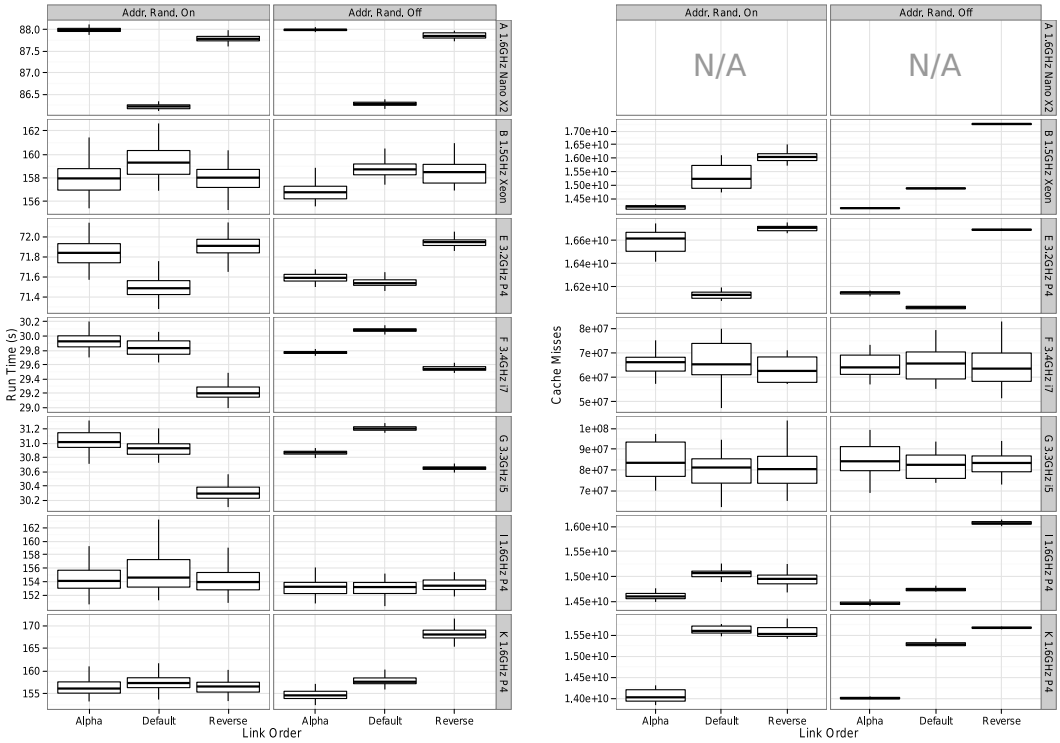
To help understand this effect, Figure 10(b) shows the cache misses for the experiment[‡]. This plot shows that there appears to be a correlation between link order and cache misses for most machines, but they do not necessarily mirror the execution time effect seen in Figure 10(a).

Finally, Figure 11 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.

We conducted a multi-way ANOVA analysis using a procedure similar to that described in Section XZ vs. bzip2: Best Bang for Your Buck?. We use the same significance level, $\alpha = 0.007$, and are subject to the same Bonferroni correction. Table IV shows the reduced ANOVA table

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

DOI: 10.1002/spe



(a) Execution Time (b) Cache Misses

Figure 10. Effect of Link Order on Perlbench Execution Time and Cache Misses

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 <0.007), but only in machine K do the ASLR factor (P <0.007) and the interaction between ASLR and link order factors (P <0.007) play a part. We confirmed that ASLR and link order could interact by examining how Linux's ASLR feature shifts the start location of linked libraries in memory. In machine A, both of these are not statistically significant at the 99% confidence level (P >0.007 in both cases). This suggests that the effect of ASLR, 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 ASLR, and the simultaneous collection of multiple relevant metrics, such as cache misses and page faults.

LESSONS LEARNED

The implementation of the infrastructure and the execution of the experiments detailed in Section Case Studies, and our experiences maintaining and using DataMill over the past two years, raised several interesting problems.

Implementation and Maintenance

We anticipated problems arising from maintaining a heterogeneous compute cluster, as each platform may require individual attention, and experience a variety of mutually exclusive problems,

DOI: 10.1002/spe

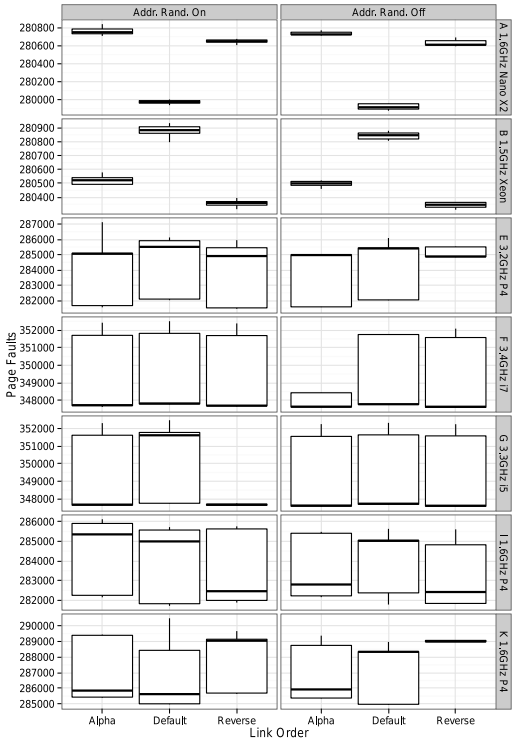


Figure 11. Effect of Link Order on Perlbench Page Faults

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 IV. Reduced ANOVA Table for Perlbench Execution Time on Machines A and K

but we faced several unexpected challenges while keeping DataMill maintained and running over the course of the two years of its operation.

No matter how reliable we make our infrastructure, problems will inevitably occur, either from lack of foresight, or from things completely out of our control. For example, even in a city with relatively stable power-grids, power outages happen, and can cause a plethora of complex problems, even more-so when an experiment is running. Machines must be able to boot back and self-heal to a sane state. They also need correctly handle partial, incomplete, and tainted experiment results, and to rerun experiment trials when necessary. Thus, we needed a policy for handling a misbehaving worker. The most important product of DataMill are the user's experiment results, and we must ensure their validity, thus when a worker fails or is otherwise tainted, we need to evaluate and ensure the validity of the results produced by that worker. The *correct* thing to do when incomplete or possibly tainted results are produced is to collect as much as possible (for debugging), but mark those results as unusable with clear warnings to the user not to use them to draw conclusions, and to notify the administrators of the problem. DataMill has several checks in place on the controller

Prepared using speauth.cls
DOI: 10.1002/spe

partition to determining whether or not an experiment completed successfully, and when anything unusual occurs, DataMill reports everything it can about what happened, and marks the job as failed.

Self-healing a broken or misbehaving machine is a challenging problem. The primary challenges in creating a self-healing worker are, determining when a worker is broken, and determining how it should work towards fixing itself. We needed to design the main-loop of the controller partition so that it would never fail so completely that it would not be able to pick up software updates and patches from upstream that could potentially resolve whatever issue that particular worker was experiencing.

Exceptions are handled carefully, and any exceptions that reach the outer-loop are treated implicitly as a trial failure. The only time that an exception is escalated to the top-level is when the last successful return to sleep mode without any successful job completion (indicating that the worker is in an infinite loop of exceptions) was over a certain number of days in the past. Once this condition is satisfied, the worker does a complete self overhaul consisting of the following steps,

1. Gather and upload log files and other information such as *dmesg*, *df*, and *mount* output to the master, continuing when exceptions are encountered.
2. Attempt to remove its registration on the master, continuing regardless of success.
3. Clear any persistent DataMill state present on disk.
4. Request a re-install of all DataMill worker software and its dependencies.

The master node must also detect worker failures. When a worker has not communicated for three days, the master triggers a re-registration process for that worker if it ever reconnects. In the majority of cases, this process is sufficient to self-heal workers when something goes amiss, and it benefits the DataMill developers and admins by automatically collecting diagnostic information when something goes wrong. The most frequent unhandled worker failures we encounter are on machines which cannot boot back up after to a power outage.

Keeping software versions synced between differing architectures is sometimes challenging and time consuming. The primary goal of DataMill is robust, reliable, and reproducible experimentation. One of the caveats of this goal is that we need to keep the differing experimental environments as controlled as possible. Maintaining a heterogeneous cluster of machines means uncovering many package incompatibilities in older and newer architectures. For the SPARC architecture in particular, we have ran into problems installing OpenJDK for user's Java-related experiments. We attempt to keep the base set of software installed on each machine identical. We also attempt to control for software versioning by recording all system package versions used during the execution of an experiment package. Although software versions are already factored out by arranging experiment results into blocks by worker, which serves both to factor out differences in architecture and software versions, we do not wish to confound architecture and system package version unless completely necessary. We control and utilize our own package mirror for the purposes of keeping machines consistent, and carefully test package updates before deploying them to all workers.

Testing DataMill worker software is an interesting challenge. It is not always obvious how each architecture will respond to a DataMill software update, so an extensive test suite that is run against all supported architectures is critical before pushing out a worker update. DataMill comprises a heterogeneous cluster of machines distributed across the globe, worker updates need to be as flawless as possible, since it is difficult and time-consuming to debug and fix problems on remote donor machines.

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.

DataMill master software is also challenging to update. Master unit and functional tests are conducted locally during development, and then remotely on a machine identical to the master server in every way, and which is periodically cloned from the master. This lets us test every aspect of the master software including the deployment process itself, we have run into several problems

DOI: 10.1002/spe

related to schema migration using real user data vs fabricated sample data. We also have the benefit of testing in an environment identical to that which is used in production, catching bugs before they become a problem. We also have virtual worker machines that connect to our test-master and which can run full experiments, letting us perform functional tests of the entire experimentation process.

Lessons from Community Usage

One general usage pattern we observed is that many users collect data for multiple trials of an experiment per job. Although the workers reboot between each job to clear the cache, many researchers ignore the effect of caches in favor of gathering research data quickly. While DataMill supports partial results files as jobs finish, researchers who run experiments that do not reboot between trials have asked for a live view of the results. This is because results are uploaded after jobs finish. Jobs with multiple trials take a longer amount of time to finish, and DataMill is currently designed to reboot for every trial.

New users of DataMill found our wiki page on getting started with DataMill helpful to learn how to use DataMill. This page explains how to make an experiment package including the setup, run, and collect scripts. It also gives a brief set of steps on testing an experiment package with DataMill. To test a package, the user only needs to launch the DataMill virtual worker and type in a public URL to download their package onto the virtual machine. To debug a package, log benchmark and controller logs are provided in a folder. One user who benchmarked only C++ programs found that he only needed to run the virtual worker once to get his first package working. In the process of debugging his package with the virtual worker, he found the benchmark and controller logs helpful.

Users prefer to use DataMill over personal equipment for benchmarking, mainly due to the different architectures and heterogeneous hardware available on DataMill. Secondly, they also favor the quick and easy interface for submitting experiments and collecting the results. In particular, users have found the experiment create user interface, which has sliders for hardware factors and checkboxes for software factors, to be intuitive and effortless to for selecting and adjusting various factors. However, researchers who only need to gather performance data for software are in a different situation. They tend to leave the software factors as default and run their experiments over the full range of hardware.

Lessons from the RV Competition

The Runtime Verification (RV) conference started in 2001 as an annual workshop. It is now an international conference for researchers to share their findings in runtime verification, which is the study of how to design artifacts for monitoring and analyzing system executions [29, 30]. Competition of Software for Runtime Verification (CSRV), first introduced in RV 2014, aims to allow researchers to share software tools, libraries, and frameworks for the instrumentation and runtime verification of software. A total of eleven institutions participated in CSRV 2014.

During CSRV 2014, DataMill provided the infrastructure necessary to run experiments, collect results, and share experiment setups. Users were provided with individual accounts to a web interface to submit their experiments. For the purposes of RV competition, DataMill executed a total of 151 experiments with 2677 jobs for the competition. Of these jobs, 832 were executed on a machine set aside for collecting final results. Of the 383 jobs that failed, most of them were from packages that were not properly tested. One common mistake made in these packages was not outputting the file to be collected for analysis at the end of the `collect.sh` script. As a result, jobs would fail due to missing result files. Another common mistake made was the selection of architectures unsupported by software used in the package. Many packages on ARM in Gentoo need to be unmasked before they can be installed. Thus, some packages failed to compile their necessary dependencies when ARM was selected as an architecture. All of failed jobs were soft failures handled by our worker design. Workers recovered automatically and continued to fetch new jobs after these failures.

The main issues encountered during CSRV 2014 came from the submission of experiments and collection of results. As a result of the structure and organization of the competition, participants ran experiments individually and submitted their results by hand to a spreadsheet, a manual-labor-intensive situation which is precisely what DataMill is designed to avoid. The infrastructure

DOI: 10.1002/spe

necessary to automate the entirety of the repetitive steps needed to carry out the competition exist in DataMill, but were not leveraged by the competition organizers at the time.

For RV 2015, the organizers decided to run the competition as a single experiment with multiple packages after all the individual submissions are tested. This ensures that the experiments are executed on equal footing, on the same machines with the same software environments, and blocked and randomized automatically by DataMill. It also allows the utilization of the entire DataMill infrastructure, and its full set of machines, providing greater confidence in the results of the competition. Finally, running the competition in this manner completely automates the collection and collation of results, so that no additional manual effort would be required.

RELATED WORK

Various researchers [4, 3, 31, 32, 33, 34] argue for more statistical rigor in computer science and groups such as the Evaluate Collaboratory [35] have been created to pursue more rigorous experimentation practices. The paper survey by Tichy *et al.* [5] concludes that numerous publications in the field of computer performance evaluation show substantial flaws in experiment design and execution. Vitek and Kalibera [14] 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.* [2] draws similar conclusions from a paper survey they conducted. Kalibera and Jones [15] 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. [36] 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.* [6] 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.* [16] 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. [17] show that random symbol names generated by a compiler leads to different memory layouts at run time, and, consequently, random variations in performance.

Curtsinger *et al.* [37, 38] reports performance regressions of up to 57% as a result of manipulating link-order, and developed a tool, STABILIZER, for combating this effect. STABILIZER improves the robustness of computer performance measurements by randomizing the memory layout of the stack, code, and data memory regions of a process at runtime. This affords fine control of the effects of memory layout during execution. Further, the tool forces any memory layout effects to approximate a Gaussian distribution, permitting the use of traditional statistical analysis techniques.

Yasugi *et al.* [39] have developed a method to benchmark programs that cancels code placement effects. The problem is that code placement changes instruction addresses of otherwise equivalent executable programs. These similar programs often exhibit different performance. Yasugi *et al.* developed a proper evaluation scheme that statistically summarizes the performance effect of several artificial programs with different code placement.

We have previously investigated the impact of memory layout factors [40] reported by Mytkowicz, Kalibera, and others. We found that, for the set of machines and benchmarks tested, out of ASLR and POSIX environment padding, only ASLR had a statistically significant impact on performance, but that even in the worst-case, ASLR had a negligible effect size for most practical purposes.

Desprez *et al.* [2] 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

DOI: 10.1002/spe

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 [41]. 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.* [2] is on distributed systems. As a result the hardware infrastructure is decidedly homogeneous, which simplifies the experiment setup and configuration. While testbeds like OpenCirrus [41] support access to low-level hardware features, those features expose little variability compared to the applications we are targeting.

PlanetLab [42, 43] 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 slices 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 an API for provisioning the slices and has facilities to isolate the network of the individual slices. This API is accessible through the PlanetLab shell and XMLRPC. To add nodes to his slice, the PlanetLab website and the PlanetLab API are two options. With the XMLRPC interface, the user authenticates with his PlanetLab username and password to add nodes to his slice. PlanetLab uses CoDeploy to efficiently and scalably distribute software from one source to many receivers [44]. CoDeploy requires HTTP access to a directory on the user's local filesystem [45]. CoDeploy is designed to push content to hundreds of PlanetLab nodes without consuming lots of bandwidth at the source node. Once the application the user wishes to run is deployed to all nodes, CoDeploy provides a tool called MultiQuery for the user to run a command simultaneously on a large number of nodes via SSH [46]. In summary, PlanetLab's infrastructure is used predominantly to evaluate and deploy distributed systems [45], 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 [45]; none of the publications include computer performance experiments. On the contrary, DataMill runs all experiments without virtualization and even reboots machines between each job. Although DataMill does not provide SSH access to workers, an interface for selecting the hardware and software factors for experiments is provided.

Various other experimentation infrastructures have been proposed with similar properties to PlanetLab. Jaffe *et al.* [47] describe a production platform with similar features to PlanetLab. The project links various large data centers in an effort to provide an experimentation platform for distributed systems. Unfortunately, the hardware chosen for the data centres is homogeneous and does not aid in the exploration of large factor space.

CloudLab [48] is an infrastructure for researching cloud computing. In terms of hardware, CloudLab is a distributed infrastructure with about 5000 CPU cores, 300-500 Terabytes of storage, and 2x10 Gbps network interfaces with Software Defined Networking at each site. CloudLab provides researchers with a state of the art environment for researching cloud computing. In terms of software, CloudLab's software stack is based on Emulab, which provides access to emulation, live-internet experimentation, 802.11 wireless, and Software-Defined Radio. Although emulation gives the user root access, it is not suitable for running benchmarks.

DOI: 10.1002/spe

HTCondor [49] 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.

Open Curation for Computer Architecture Modeling (OCCAM) [50] is a project for computer architecture experimentation that aims to make experiments reproducible. OCCAM proves infrastructure components such as standardized simulators and benchmarks for computer architecture researchers. OCCAM aims to allow researchers to more quickly and easily show that their techniques improve the state-of-the-art. Experiments in OCCAM are created with a workflow builder that allows researchers to connect various objects together. OCCAM provides tools to build, debug, and deploy these objects including simulators, benchmarks, and tracers. The workflow builder contains forms for configuring simulators and a GUI for connecting objects together.

LAVA from Linaro is an open source software QA and validation system for ARM boards that shares a few similarities with DataMill. Like DataMill, LAVA is a distributed and heterogeneous experimentation infrastructure, it has master and worker nodes. Jobs from the master node are distributed to workers. The two infrastructures differ in that LAVA is a validation lab for hardware, and DataMill is a platform for performance evaluation. As a result, their main feature sets also differ as shown in Table V. For example, LAVA results are reported as pass or fail whereas results in DataMill are specified by the user. DataMill allows researchers to select hardware and software factors, and LAVA only partially supports the selection of hardware factors through specifying boards in the test definition. LAVA offers a diverse selection of ARM boards, a commandline XML-RPC interface, and testing of U-boot images. LAVA requires the use of either a KVM switch or devices that support PXE-boot to run tests on targets [51], but DataMill supports any device that can run Linux and connect to a network. For debugging software packages, LAVA offers the LAVA test shell, which is an SSH session, while DataMill provides a virtual machine with the same set of software on workers. Furthermore, in LAVA test definitions are written in JSON and commands to run inside the test are written in YAML [52]. Contrarily, DataMill only requires the run and collect BASH scripts.

	LAVA	Datamill
VM for testing benchmark packages	No	Yes
Selection of hardware factors	Partial (boards only)	Yes
Selection of software factors	No	Yes
Commandline XML-RPC interface	Yes	No
Testing of U-boot images	Yes	No
OpenID login	Yes	No
Web interface to submit jobs	Yes	Yes
Types of tests	Pass/Fail unit tests	User specified
Debugging benchmark packages	Test shell	Virtual machine

Table V. LAVA feature comparison with DataMill

FUTURE WORK

As part of our future work we consider various improvements to the existing system architecture. Huang *et al.* [53] 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

DOI: 10.1002/spe

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 basic performance metrics of user's submitted experiments, users must ultimately analyze 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 [54]), and services (i.e., Amazon Web Services [55]) 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.

LAVA and DataMill share many similarities which we plan to leverage. LAVA's architecture is similar to the architecture of DataMill, and it is conceivable that features currently unique to DataMill could be implemented as specific test images in the LAVA infrastructure. We plan to experiment with using LAVA images as a starting point for installing Gentoo on ARM boards. In particular, the availability of a working kernel alleviates the problem of finding a working kernel tree and configuration. Further more, we plan to contribute DataMill worker images for x86 machines which would report results to the master node, allowing LAVA users to collect more complex results.

CONCLUSION

In this paper we discuss DataMill version 2.0. DataMill provides services to set up and execute robust, replicable, and reproducible experiments. DataMill enables researchers to publish their experiment software, setup, and results. In this 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. DataMill can be accessed and utilized through its web interface here [8].

REFERENCES

1. Oliveira A. Measuring and predicting computer software performance: Tools and approaches. PhD Thesis, University of Waterloo 2015.
2. Desprez F, Fox G, Jeannot E, Keahey K, Kozuch M, Margery D, Neyron P, Nussbaum L, Perez C, Richard O, *et al.*. Supporting Experimental Computer Science. *Technical Report*, Argonne National Laboratory Technical Memo 2012.
3. Peterson L, Pai VS. Experience-Driven Experimental Systems Research. *ACM Communications* 2007; **50**(11):38–44, doi:10.1145/1297797.1297820. URL http://doi.acm.org/10.1145/1297797.1297820.
4. Tichy WF. Should Computer Scientists Experiment More? *IEEE Computer* 1998; **31**(5):32–40.
5. Tichy WF, Lukowicz P, Prechelt L, Heinz EA. Experimental Evaluation in Computer Science: A Quantitative Study. *Systems Software* 1995; **28**:9–18.
6. Mytkowicz T, Diwan A, Hauswirth M, Sweeney PF. Producing Wrong Data Without Doing Anything Obviously Wrong! *SIGPLAN Notes* Mar 2009; **44**(3):265–276, doi:10.1145/1508284.1508275. URL http://doi.acm.org/10.1145/1508284.1508275.
7. de Oliveira AB, Petkovich JC, Reidemeister T, Fischmeister S. DataMill: Rigorous Performance Evaluation Made Easy. *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, ACM: New York, NY, USA, 2013; 137–148, doi:10.1145/2479871.2479892. URL http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/2479871.2479892.
8. The DataMill Team. DataMill. http://datamill.uwaterloo.ca/. Accessed Jul. 12th, 2015.
9. Antony J. *Design of Experiments for Engineers and Scientists*. Butterworth-Heinemann, 2003.
10. Montgomery D. *Design and Analysis of Experiments*. John Wiley & Sons, 2008.
11. Kalibera T, Jones R. Rigorous benchmarking in reasonable time. *ACM SIGPLAN Notices* 2013; **48**(11):63–74.
12. Kalibera T, Jones RE. Quantifying performance changes with effect size confidence intervals. *Technical Report*, Technical Report 4–12, University of Kent 2012.
13. NLANR/DAST. Iperf. http://iperf.sourceforge.net/. Accessed Sep. 17th, 2012.

14. Vitek J, Kalibera T. Repeatability, Reproducibility, and Rigor in Systems Research. *Proceedings of The Ninth ACM International Conference on Embedded Software*, EMSOFT '11, ACM: New York, NY, USA, 2011; 33–38, doi: 10.1145/2038642.2038650. URL http://doi.acm.org/10.1145/2038642.2038650.
15. Kalibera T, Jones R. Handles Revisited: Optimising Performance and Memory Costs in a Real-Time Collector. *Proceedings of The International Symposium on Memory Management*, ISMM '11, ACM: New York, NY, USA, 2011; 89–98, doi:10.1145/1993478.1993492. URL http://doi.acm.org/10.1145/1993478.1993492.
16. Harji AS, Buhr PA, Brecht T. Our Troubles With Linux and Why You Should Care. *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, ACM: New York, NY, USA, 2011; 2:1–2:5, doi:10.1145/2103799.2103802. URL http://doi.acm.org/10.1145/2103799.2103802.
17. Kalibera T, Tuma P. Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results. *Proceedings of the Third European Conference on Formal Methods and Stochastic Models for Performance Evaluation*, EPEW'06, Springer-Verlag: Berlin, Heidelberg, 2006; 63–77, doi:10.1007/11777830_5. URL http://dx.doi.org/10.1007/11777830_5.
18. Julian Seward. bzip2 and libbzip2. http://www.bzip.org/. Accessed Sep. 17th, 2012.
19. Vince Weaver. Perf Event Overhead Measurements. http://web.eecs.utk.edu/~vweaver1/projects/perf-events/benchmarks/rdtsc_overhead/. Accessed Sep. 17th, 2012.
20. Linaro Limited. Welcome to LAVA. https://validation.linaro.org/. Accessed Mar. 27th. 2014.
21. distcc. distcc: a fast, free distributed C/C++ compiler. https://code.google.com/p/distcc/. Accessed Feb. 10th. 2014.
22. Bouqata B, Carothers C, Szymanski B, Zaki M. Understading filesystem performance for data mining applications. *Proceedings of the 6th International Workshop on High Performance Data Mining: Pervasive and Data Stream Mining (HPDM: PDS 03)*, Citeseer, 2003. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.4945&rep=rep1&type=pdf.
23. Payer M, Gross TR. String Oriented Programming: When ASLR is Not Enough. *Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, ACM: New York, NY, USA, 2013; 2:1–2:9, doi:10.1145/2430553.2430555. URL http://doi.acm.org/10.1145/2430553.2430555.
24. Kemp, S. Optimizing servers - Tuning the GNU/Linux Kernel. http://tweaked.io/guide/kernel/. Accessed Mar. 7th. 2014.
25. The Tukaani Project. XZ Utils. http://tukaani.org/xz/. Accessed Sep. 17th, 2012.
26. W Bergmans. Maximum Compression. http://www.maximumcompression.com/data/files/index.html. Accessed Sep. 17th, 2012.
27. Jain R. *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing, Wiley, 1991.
28. Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006/. Accessed Sep. 17th, 2012.
29. Bonakdarpour B, Navabpour S, Fischmeister S. Time-triggered runtime verification. *Formal Methods in System Design* Aug 2013; **43**(1):29–60, doi:10.1007/s10703-012-0182-0. URL http://link.springer.com/10.1007/s10703-012-0182-0.
30. Watterson C, Heffernan D. Runtime verification and monitoring of embedded systems. *IET Software* Oct 2007; **1**(5):172–179, doi:10.1049/iet-sen:20060076.
31. Denning PJ. ACM President's Letter: What is Experimental Computer Science? *ACM Communications* Oct 1980; **23**(10):543–544, doi:10.1145/359015.359016. URL http://doi.acm.org/10.1145/359015.359016.
32. Denning PJ. ACM President's Letter: Performance Analysis: Experimental Computer Science as its Best. *ACM Communications* Nov 1981; **24**(11):725–727, doi:10.1145/358790.358791. URL http://doi.acm.org/10.1145/358790.358791.
33. Comer DE, Gries D, Mulder MC, Tucker A, Turner AJ, Young PR. Computing as a Discipline. *ACM Communications* Jan 1989; **32**(1):9–23, doi:10.1145/63238.63239. URL http://doi.acm.org/10.1145/63238.63239.
34. Denning PJ. Is Computer Science Science? *ACM Communications* Apr 2005; **48**(4):27–31, doi:10.1145/1053291.1053309. URL http://doi.acm.org/10.1145/1053291.1053309.
35. Evaluate Collaboratory. Evaluate Collaboratory Technical Reports. http://evaluate.inf.usi.ch/technical-reports/. Accessed Aug. 1st, 2015.
36. Georges A, Buytaert D, Eeckhout L. Statistically Rigorous Java Performance Evaluation. *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, ACM: New York, NY, USA, 2007; 57–76, doi:10.1145/1297027.1297033. URL http://doi.acm.org/10.1145/1297027.1297033.
37. Curtsinger C, Berger E. Stabilizer: Enabling statistically rigorous performance evaluation. *Technical Report*, University of Massachusetts 2012.
38. Curtsinger C, Berger ED. STABILIZER: Statistically Sound Performance Evaluation. *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, ACM: New York, NY, USA, 2013; 219–228, doi:10.1145/2451116.2451141. URL http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/2451116.2451141.
39. Yasugi M, Matsuda Y, Ugawa T. A proper performance evaluation system that summarizes code placement effects. *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ACM, 2013; 41–48.
40. de Oliveira AB, Petkovich JC, Fischmeister S. How much does memory layout impact performance? a wide study. *Intl. Workshop Reproducible Research Methodologies*, 2014; 23–28. URL http://www.occamportal.org/images/reproduce/papers/reproduce14_paper_06.pdf.
41. Campbell R, Gupta I, Heath M, Ko SY, Kozuch M, Kunze M, Kwan T, Lai K, Lee HY, Lyons M, *et al.*. Open CirrusTMcloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research. *Proceedings of The 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, USENIX Association:

Berkeley, CA, USA, 2009. URL http://dl.acm.org/citation.cfm?id=1855533.1855534.
42. Peterson L, Bavier A, Fiuczynski ME, Muir S. Experiences Building PlanetLab. *Proceedings of The 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, USENIX Association: Berkeley, CA, USA, 2006; 351–366. URL http://dl.acm.org/citation.cfm?id=1298455.1298489.
43. Paterson L, Roscoe T. The Design Principles of PlanetLab. *Operating Systems Review* January 2006; **40**(1):11–16.
44. Park, K, Pai V, Peterson, L. CoDeploy - A Scalable Deployment Service for PlanetLab. http://codeen.cs.princeton.edu/codeploy/ visited 2015-07-10.
45. PlanetLab. PlanetLab Bibliography. http://www.planet-lab.org/biblio visited 2012-09-28.
46. PlanetLab. HelloWorldTutorial PlanetLab. http://svn.planet-lab.org/wiki/HelloWorldTutorial visited 2015-07-10.
47. Jaffe E, Bickson D, Kirkpatrick S. Everlab: A Production Platform for Research in Network Experimentation and Computation. *Proceedings of the 21th Large Installation System Administration Conference*, 2007; 203–213.
48. The University of Utah. CloudLab. https://www.cloudlab.us/. Accessed Jul. 18th, 2015.
49. Tannenbaum T, Wright D, Miller K, Livny M. Condor – a distributed job scheduler. *Beowulf Cluster Computing with Linux*, Sterling T (ed.). MIT Press, 2001.
50. Bruce Childer. OCCAM. http://www.occamportal.org/. Accessed Jul. 18th, 2015.
51. Linaro Limited. Deploying a KVM (x86_64) Device. https://validation.linaro.org/static/docs/kvm-deploy.html. Accessed Mar. 27th. 2014.
52. Linaro Limited. Writing a LAVA test definition. https://validation.linaro.org/static/docs/writing-tests.html. Accessed Mar. 27th. 2014.
53. Huang M, Bavier A, Peterson L. PlanetFlow: Maintaining Accountability for Network Services. *SIGOPS Oper. Syst. Rev.* Jan 2006; **40**(1):89–94, doi:10.1145/1113361.1113376. URL http://doi.acm.org/10.1145/1113361.1113376.
54. Apache Software Foundation. Hadoop. http://hadoop.apache.org/. Accessed Sep. 17th, 2012.
55. Amazon Web Services LLC. Amazon Web Services. http://aws.amazon.com/. Accessed Sep. 17th, 2012.

APPENDIX

ApacheBench	http://httpd.apache.org/docs/2.2/programs/ab.html
Emacs	http://www.gnu.org/software/emacs/
CLAPACK	http://www.netlib.org/clapack/
CoreMark	https://www.eembc.org/coremark/
GFortran	https://gcc.gnu.org/wiki/GFortran
Himeno	http://accc.riken.jp/2444.htm
Iozone	http://www.iozone.org/
JavaLinpack	http://www.netlib.org/benchmark/linpackjava/
GCC	https://gcc.gnu.org/
clang	http://clang.llvm.org/
JFlex	http://jflex.de/
LAME	http://lame.sourceforge.net/
Libgcrypt	http://www.gnu.org/software/libgcrypt/
libMicro	https://java.net/projects/libmicro
MiBench	http://wwweb.eecs.umich.edu/mibench/
NUMA-bench	http://code.compeng.uni-frankfurt.de/projects/numabench/
PgBench	http://www.postgresql.org/docs/devel/static/pgbench.html
pipebench	http://www.habets.pp.se/synscan/programs.php?prog=pipebench
Postmark	http://www.filesystems.org/docs/auto-pilot/Postmark.html
SPECCPU2006	https://www.spec.org/cpu2006/
SPECjvm2008	https://www.spec.org/jvm2008/
SQlite	https://www.sqlite.org/
X264	http://www.videolan.org/developers/x264.html

Table VI. Software and Benchmarks Tested with DataMill

DOI: 10.1002/spe

CPU architecture	Number of CPU cores	CPU model	CPU frequency (MHz)	RAM size (KB)	RAM type
Intel(R) Atom(TM) CPU E3845	4	x86_64	1909	1949348	DDR3
Intel(R) Atom(TM) CPU E3845	4	x86_64	1909	1949348	DDR3
Intel(R) Atom(TM) CPU E3845	4	x86_64	1909	1949348	DDR3
Intel(R) Atom(TM) CPU E3845	4	x86_64	1909	1949348	DDR3
AM37x ARM Cortex-A8	1	armv7l	797	500932	LPDDR
Intel(R) Pentium(R) M processor	1	i686	1694	901048	SDRAM
Intel(R) Pentium(R) 4 CPU	2	i686	2993	892688	DDR
Intel(R) Pentium(R) 4 CPU	2	i686	3200	900992	DDR
Intel(R) Pentium(R) 4 CPU	1	i686	2400	900968	DDR
Intel(R) Pentium(R) 4 CPU	1	i686	1595	253236	SDRAM
Quad-Core AMD Opteron(tm) Processor 8378	16	x86_64	2411	32963948	DDR2
Intel(R) Pentium(R) 4 CPU	2	i686	2998	892800	DDR
Intel(R) Pentium(R) 4 CPU	2	i686	2993	892800	SDRAM
Intel(R) Pentium(R) 4 CPU	2	i686	3200	900992	DDR
Intel(R) Pentium(R) 4 CPU	2	i686	2793	900964	SDRAM
Intel(R) Pentium(R) 4 CPU	1	i686	1614	901048	DRAM
Intel(R) Pentium(R) 4 CPU	2	i686	3198	900992	DDR
Intel(R) Pentium(R) D CPU	2	x86_64	2993	2054624	DDR
Intel(R) Pentium(R) 4 CPU	2	i686	3200	900992	DRAM
Intel(R) Xeon(R) CPU E5-2699 v3	36	x86_64	2301	16337272	Other
Intel(R) Xeon(R) CPU 5160	2	x86_64	3000	2826628	DDR2
Intel(R) Core(TM) i5-4300U CPU	4	x86_64	1900	3706100	DDR3
Intel(R) Core(TM) i5-4300U CPU	4	x86_64	1900	3706108	DDR3
Intel(R) Core(TM) i5-4300U CPU	4	x86_64	1900	3706100	DDR3
Intel(R) Core(TM) i5-4300U CPU	4	x86_64	1900	3706108	DDR3
Intel(R) Xeon(R) CPU E5-2690 v2	20	x86_64	3001	8148976	DDR3
Intel(R) Xeon(R) CPU E5-2640 v2	16	x86_64	2001	8149428	DDR3
Intel(R) Core(TM) i7-2600K CPU	8	x86_64	3401	8094688	DDR3
ARMv7 Processor rev 10 (v7l)	4	armv7l	791	896568	DDR3
ARMv7 Processor rev 10 (v7l)	4	armv7l	791	896568	DDR3
ARMv7 Processor rev 10 (v7l)	4	armv7l	1988	896568	DDR3
ARMv7 Processor rev 10 (v7l)	4	armv7l	1988	896568	DDR3
Intel(R) Core(TM) i5-2500 CPU	4	x86_64	3301	8150896	DDR3
AMD G-T56N Processor	2	x86_64	1650	1647368	DDR3
AMD G-T56N Processor	2	x86_64	1650	1647368	DDR3
Freescale i.MX 6 ARM Cortex-A9 Quad core	4	armv7l	1024	1025472	DDR3
Freescale i.MX 6 ARM Cortex-A9 Quad core	4	armv7l	1024	1025472	DDR3
Freescale i.MX 6 ARM Cortex-A9 Quad core	4	armv7l	1024	1025472	DDR3

Table VII. Hardware Platforms Currently Active in DataMill

DOI: 10.1002/spe