Technical Report

UCAM-CL-TR-859
ISSN 1476-2986

Number 859

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Resourceful: fine-grained resource accounting for explaining service variability

Lucian Carata, Oliver Chick, James Snee,
Ripduman Sohan, Andrew Rice, Andy Hopper

September 2014

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Resourceful: fine-grained resource accounting for explaining service variability

Lucian Carata Oliver Chick James Snee
Ripduman Sohan Andrew Rice Andy Hopper

Computer Laboratory, University of Cambridge, UK
firstname.lastname @cl.cam.ac.uk

Abstract

Increasing server utilization in modern datacenters also increases the likelihood of contention on physical resources and unexpected behavior due to side-effects from interfering applications. Existing resource accounting mechanisms are too coarse-grained for allowing services to track the causes of such variations in their execution. We make the case for measuring resource consumption at system-call level and outline the design of *Resourceful*, a system that offers applications the ability of querying this data at runtime with low overhead, accounting for costs incurred both synchronously and asynchronously after a given call.

1 Introduction

Running multiple services on the same physical host, multiplexed over a pool of shared resources, is a common method for increasing machine utilization. This is typically achieved through operating system process separation, containerization or hypervisor-based virtualization. However, depending on the workloads of the collocated services, the increase in utilization may also increase contention on system resources. This translates into greater variability in system behavior, and lower predictability in terms of performance and failure rates [4].

By design, the control planes managing hardware resources (the hypervisor and/or the OS kernel) are viewed as black boxes by any applications executing on top. This means that there is no direct way for a single process to detect or query whether its execution is affected by resource contention (e.g. cache trashing, IO storms). Furthermore, it is difficult to diagnose and recover from side effects of performance degradation caused by contention (e.g. fewer results returned, loss of precision in timeout-based processing algorithms).

The fundamental step towards enabling applications to measure and react to periods of resource contention during their execution is giving them the ability to query detailed

*in alphabetical order

resource consumption data on the actions they perform. To this end, we present Resourceful, a framework that provides configurable resource utilization measurements at system call granularity to applications interested in monitoring their footprint in the context of overall resource consumption.

For actions performed completely in user space, applications can already be instrumented to monitor their own execution. In contrast, once a system call is made, they no longer have any control or visibility into what happens at kernel level in terms of resource consumption. However, many applications end up spending a significant percentage of their time in the kernel [2].

It is possible to make coarse-grained measurements at this level in terms of *aggregate* resource consumption using existing profiling and monitoring tools. For example, the Linux kernel provides mechanisms such as the `getrusage()`[1] call, `perf`[2], and `ftrace`[3]. Other tools such as `iotop`[4] or `netstat`[5] use information exposed through `/proc`, while DTrace [3] and SystemTap [12] allow the user to write scripts that can gather similar data. However, current methods fall short in the following important dimensions:

(i) **Accounting granularity and aggregation:** The mechanisms above can only obtain system-wide or per-process statistics. Therefore, it is difficult to understand the contribution of a particular process' *activity* towards total resource consumption. For example, if one wishes to diagnose occasional high latency responses from a server, then per-process aggregated data is of limited use. Instead, fine-grained information is needed, such as per system call data aggregated only over the lifetime of the request-response cycle. Existing tools that can track individual function calls, such as `ftrace`, are often limited to debugging scenarios because of the overheads they introduce.

(ii) **Accounting for resources consumed asynchronously:** not all the effects of a given system call occur during its execution. A very simple example here is a process writing data to disk: while the application makes multiple calls to `write()` on a particular file descriptor, there may be no immediate I/O activity on disk because of the kernel's use of a buffer cache. The actual submission to hardware will occur asynchronously, depending on the I/O scheduler, the expiration of a flushing interval, or on an explicit call to `fsync`. Simply recording resource consumption metrics before and after a `write` call will not capture its full cost. In order to fully explain shared resource usage, one needs to take into account asynchronous effects and assign the costs of running them to original causes. There are no existing tools supporting this type of investigation.

(iii) **Online analysis and feedback:** with the exception of `getrusage()`, most kernel-level resource accounting mechanisms available are designed for debugging or offline analysis scenarios – the final output is a log that needs to be processed in order to extract relevant data. The applications themselves never have access to this data while running, and therefore cannot adapt in real time to other concurrent workloads. Instead, applications are given the illusion of exclusive resource ownership.

The difficulty in providing the OS support for dealing with those issues can be attributed to the high overhead that is typically introduced by fine grained measurements

[1] `http://www.gnu.org/software/libc/manual/html_node/Resource-Usage.html`
[2] `https://perf.wiki.kernel.org/index.php/Main_Page`
[3] `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`
[4] `http://guichaz.free.fr/iotop/`
[5] `http://linux.die.net/man/8/netstat`

and to the challenge of tracking causal dependencies between pieces of code executed by the kernel. Resourceful addresses these problems through *selective* kernel probing, informed by static binary and code analysis.

The main contributions of this paper are:

Describing an architecture that allows applications to gather fine-grained (system call level) resource consumption data, broken down per kernel subsystem, with low overhead.

Presenting a method for automatically identifying kernel subsystem boundaries and the minimal number of required instrumentation probe points, using static analysis.

A framework for attributing resource consumption of asynchronous tasks to the calls that triggered their execution.

Our current implementation focuses on gathering resource usage data from the Linux kernel. However, the overall design is general enough to allow for implementation in hypervisors and extension to other codebases.

In this paper, we show that it is possible to run a system tracking resource consumption based on our design with reasonable overhead. Our initial experiments consider accounting for socket accept/send/receive resource consumption in a real application (lighttpd). A full investigation on the accuracy of the recorded data remains as future work.

2 Resourceful: System design

Fine-grained accounting is required in cases where developers need insights into the way applications behave or interact with each other: for example, one might want to understand the cost of a client request, either on a single host or over a distributed system; this implies being able to aggregate data at sub-process granularity (summing over the calls that were made to service that request). On high latency requests, there is a need of diagnosing causes: What is different from the low latency case? Were there unintended interactions between the server and other co-located applications? Where was most of the time spent?

The same fine-grained resource usage data can be used in improving user-level scheduling, or application coordination: applications could take such decisions based on the current state, for example by task-level coalescing of requests [5], or could trade resources similar to auction frameworks [10].

With these use cases in mind, Resourceful is designed to give applications full control over measuring resource consumption of system calls. The framework has three main components: (i) *measurement configuration:* this analyzes the current kernel in order to identify a minimal set of instrumentation probe points and subsystem boundaries (the level at which aggregation takes place), guided by a user-provided configuration; (ii) *a kernel module* responsible for inserting these probes into the kernel and activating them when applications request resource consumption data and (iii) *a user-space library* exposing an API that applications can use to express interest in the resource consumption of particular system calls and to read the results after the required information was gathered on the kernel side.

Each resource accounting result provides detailed metrics grouped by kernel subsystem. For example, the data recorded for one system call would contain total CPU cycles, wall clock time and memory costs, but these values are further broken down for each functional subsystem touched during that call: total CPU cycles spent in the Network subsystem, total cycles spent in VFS and subsystem-specific metrics such as bytes sent/received, number of retransmissions, IO queue size, disk writes. The application can select exactly which of those metrics are recorded and can also perform custom aggregations across multiple system calls.

By reporting resource consumption aggregated by subsystem, Resourceful provides a more detailed view of what happens inside the kernel. For example, given a socket `send()` operation, the application can view the breakdown of latency and answer questions such as: Was most of the time spent in the network stack? Was the packet delayed by the scheduler moving the task on a different core?

2.1 Kernel subsystem identification

The Linux kernel has a modular structure of *subsystems*, such as VFS, logical filesystems, block devices. A complete list of these can be found in the Linux Kernel Map.[6] Resourceful uses this logical structure in identifying suitable probing points for reporting the resource consumption on a per-subsystem basis.

To identify subsystems, we perform inter-procedural static analysis of the currently-running kernel. We start by finding all `call` instructions in the kernel's binary, and determine the function symbols for both the callee and the caller. Each function is then categorized into one of the subsystems in the Linux Kernel Map, predominantly based on its source file location. We track the function calls that are made from within one subsystem to another, and we consider them to be part of the subsystem boundary.

```
global {
  subsystem_whitelist: net_link_layer
}

subsystem net_link_layer {
  boundary:
    probe dev_queue_xmit {
      arg     : skb
      capture: {
        name: net_buf_enq,
        val : &skb->dev->qdisc
      }
    },
    probe qdisk_restart {
      arg     : dev
      capture: {
        name: net_buf_deq,
        val : &dev->qdisc
      }
    }
  metrics: cycles
  map_async: match(net_buf_deq, net_buf_enq)
}
```

Listing 1: Sample configuration file defining a custom subsystem

[6] http://www.makelinux.net/kernel_map/

Instrumentation probes are then inserted around the locations where such boundary functions are called. Concretely, if the `sys_socket` function (from the Network subsystem) calls `kmalloc` (from the Memory subsystem), we need to add probes surrounding the call site of `kmalloc` within `sys_socket`. Even if this means we could potentially be setting numerous probes (a function such as kmalloc is called often), it is the only way to avoid false positives: setting the probes inside `kmalloc` would mean probing more often than necessary: any call to `kmalloc` would be probed, even when it's not on a subsystem boundary (e.g any call to `kmalloc` from within the Memory subsystem). Whilst inserting more probes has a higher startup cost, there is no runtime cost of unused probes and there is no increase in code size from inserting unused probes.

Besides the subsystems identified automatically this way, we allow users to influence the process through configuration files such as the one presented in Listing 1. Here, users can remove subsystems detected automatically or add their own. The example shows how the network link layer could be defined as a separate subsystem, generating probes for when packets are enqueued or dequeued from device buffers.

So far, we have applied our analysis to the Linux kernel. However, this approach can be extended to any codebase where modules are organized using a directory-based structure.

2.2 Kernel accounting infrastructure

Having identified the minimum number of probe points required for measuring per-subsystem resource consumption, a kernel module has the job of inserting those probes at runtime. For the Linux kernel, this can be achieved using kprobes.[7]

When triggered, each kprobe records a snapshot of resource consumption metrics according to the subsystem in which the probed function resides. For gathering actual data, we use existing kernel mechanisms such as `perf_events` for reading CPU Performance Monitoring Unit (PMU) data or structures that contain statistics, like `tcp_info`.

Besides making sure that those probes run and snapshot resource consumption statistics when required, the kernel module also needs to efficiently store this information and make it accessible to user-space applications. We provide an overall schematic of how this works in Figure 1.

On initialization, the kernel module creates two character devices: a data device (`/dev/rscfl`) for resource accounting information and a control device (`/dev/r_ctl`) for communication between user space and the module. When applications linking with the Resourceful library call `init()`, the two devices are mmap-ed into their local address space. This is done for each application thread, and the corresponding `mmap` allocates a new buffer on the kernel side to hold resource accounting data for system calls made from that thread.

On hitting a probe, the module needs to determine the corresponding accounting buffer for writing resource consumption data. An index that links given `pids` to their corresponding buffers is maintained in per-cpu hash tables for this purpose, avoiding locks on the critical path. Resourceful intercepts calls to scheduler functions in order to maintain those indexes up-to-date for each cpu.

[7] `https://www.kernel.org/doc/Documentation/kprobes.txt`

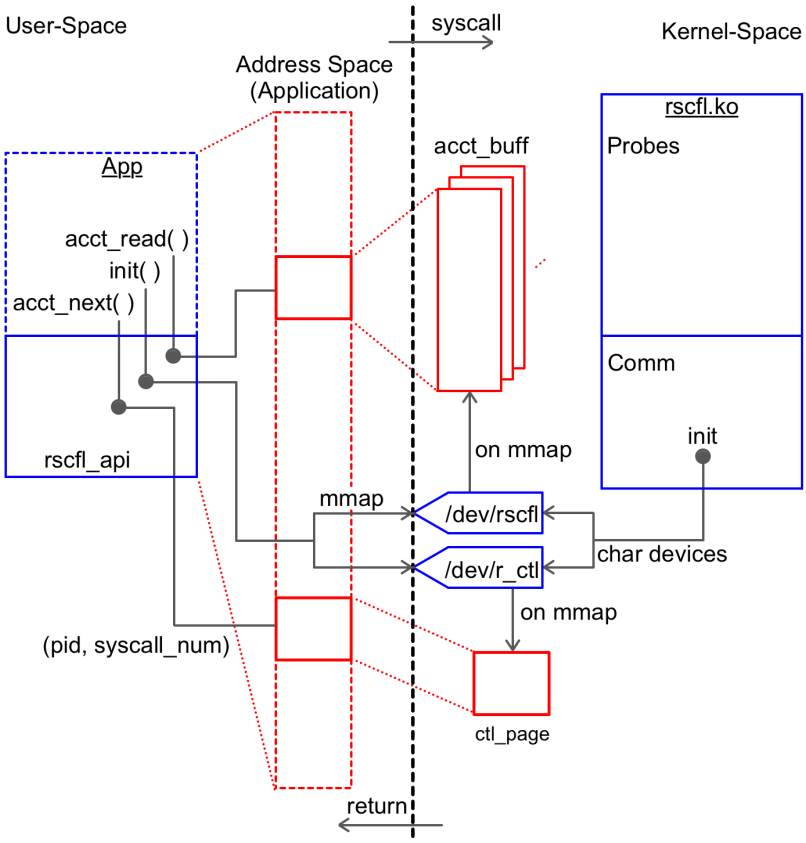

Figure 1: Primary user-space/kernel-space interactions detailed. Resource accounting data is requested by applications for specific system calls and is then available for reading (zero-copy) from kernel buffer regions mmap-ed into the current address space.

Accounting for Asynchronous Resource Consumption

As described in our initial example, the total cost of performing a buffered write should also consider the asynchronous operation committing the data to disk. Unlike existing approaches, Resourceful separately reports the resources consumed asynchronously after making such system calls. Without reporting the amortized, asynchronous cost, current performance monitoring APIs provide an incomplete representation of system resource consumption.

The solution for tracking calls that have caused a given asynchronous task hinges on the observation that a link between two operations can only exist if the kernel code maintains some data structure shared by both. *Cache buffers* in which data is enqueued and later dequeued asynchronously (flushed) and *timers* started by a function for running a given task when reaching zero are just two examples. Tracking the lifetime of those shared structures and any actions performed on them is sufficient for determining the calls to which resource consumption should be attributed.

When multiple system calls are amortized over the same asynchronous operation (e.g. multiple writes being buffered and then flushed to disk once), the resources consumed by the flush will have to be divided amongst all the initial system calls, following an *attribution strategy*. For buffered writes, the simplest strategy is to divide the costs proportional to the size of each write.

Resourceful performs accounting for asynchronous kernel operations by observing that Linux provides a number of abstractions for performing asynchronous work: *timers*, *tasklets*, *workqueues* and *interrupts*. Each of these is characterized by particular shared data structures, and the framework will index their addresses in order to track the operation that triggered the asynchronous execution.

Custom asynchronous accounting can also be set up: Listing 1 shows how network scheduler buffers (the qdisc buffer) can be tracked across enqueue/dequeue operations: when the `dev_queue_xmit` probe is hit (synchronously), Resourceful will store the address of the qdisc buffer. A following `qdisc_restart` probe being hit on an asynchronous path will match on the address of the qdisc buffer and search the index for the correct accounting structure that should be filled.

2.3 User space API

The core of Resourceful user's space API is formed by the three functions presented in Figure 1. `init()` must be called by the application on every thread that needs resource consumption data, and returns a handle used by the other functions in identifying the locations of mmaped buffers.

Before a system call of interest, the developer calls `acct_next()`. This will write an entry to the control device marking the fact that the kernel infrastructure must track resource consumption for the next system call comming from that `pid`. Variants of `acct_next` exist, in case the developer needs accounting for the next n system calls, or just needs to start/stop accounting at specific points in the application. `acct_next` also allows the user to pass in a bit array in order to filter-out uninteresting kernel subsystems from the result. Custom aggregation can be set up by passing an integer token (arbitrarily chosen by the user) to multiple `acct_next` calls. The data for all the calls with a given token value will be aggregated in the same accounting structure.

The third API call, `acct_read`, allows zero-copy reads of accounting data from the kernel. Synchronous costs can be read immediately, and a callback can be set up for running when the asynchronous part of the cost has also been recorded. The returned data structure contains the same bit array used to apply filters, but this now specifies which subsystems the system call has actually touched (in this way, the application knows which elements of the structure contain valid data).

An API extension which we have not yet implemented in our prototype will allow applications to `mmap` resource accounting buffers of other processes. This paves the way for applications that react to concurrent workloads, by throttling or delaying their own operations for example.

3 Initial results

We have developed the kernel subsystem identification and resource accounting prototypes separately, with the purpose of showcasing the feasibility of the design and gathering preliminary overhead data.

At the moment, the kernel module uses SystemTap to manually set kprobes for the network, memory and filesystem kernel sub-systems.

In order to understand the overhead introduced by Resourceful, we have modified *lighttpd* to link with our user-space library and do resource accounting for each socket

accept/send/receive call. At runtime, lighttpd is configured to serve a number of static files for performing end-to-end latency and throughput tests.

To characterize overhead, we first look at the distribution of latencies and compare a vanilla lighttpd binary with the one that runs resource accounting using Resourceful, for 10 000 sequential requests issued using `http_ping`. As observed in Figure 2, the median latency increases by 3.9% when resource accounting is active. The tail latency is also slightly increased, with the 99th percentile growing by 7.66%.

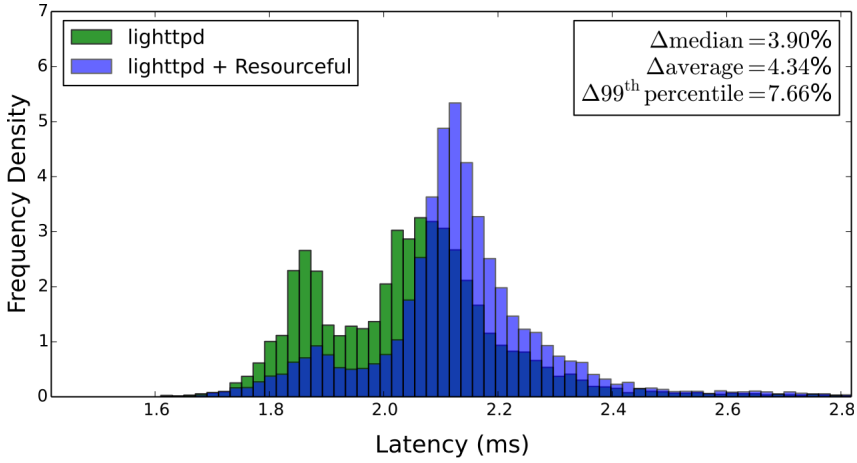

Figure 2: Normalized latency histograms showing changes in latency distribution when enabling Resourceful for lighttpd

Importantly, the overall characteristics of the latency distribution stay the same: we see two local maxima and the overall shape remains similar. This suggests that even with the overhead of measuring resource consumption, one would still observe the latency characteristics of the underlying system.

Table 1 shows the drop in throughput (~31%) averaged over 10 runs when doing 100k sequential requests for small files using `http_load`. This should be seen as an overhead upper-bound, as we have not yet performed any optimizations on the system as a whole. An in-depth evaluation is planned as future work.

lighttpd	Base	+ Resourceful
Fetches / second (avg)	2486.4	1716.9
Bytes / second (avg)	54700.5	37772.0

Table 1: Change in throughput when enabling Resourceful for lighttpd

4 Related work

Many existing tracing and profiling tools only instrument a thin layer of the target system. For example Dapper [11] makes use of instruments in an RPC library and X-Trace [7] modifies only the protocol and network stacks. Whilst these tools are able to provide detailed information about their respective instrumented layers, they treat much of surrounding system as a black-box. The Google-Wide Profiling tool [9] attempts to overcome this narrow view by profiling the entire target system, but instead of being

continuous it is sample-based, meaning it makes a trade-off between sample rate and visibility. Resourceful differs from these by allowing any part of a target system to be continually profiled whilst only incurring a low-overhead.

Other tracing systems such as Magpie [1] require the user to provide a definition of the expected system structure in order for it to correctly correlate events. Resourceful automatically identifies subsystem boundaries, although those can also be manually configured by the user.

The primary method of interacting with Resourceful is through its user-space API. Other systems such as Fay [6] go some way towards providing a programmatic interface to trace and profile data, but are still limited to a post-hoc evaluation. Resourceful's API allows trace data to be retrieved and evaluated in line with the target processes execution. This runtime availability of resource consumption data give a much richer view of how the application and kernel are performing, allowing for more informed decisions to be made.

One useful feature to have when designing a dependable system is being able to accurately predict the future. The system described by Ostrowski et al [8] provides methods for users to carry out what-if analysis on existing trace data. Resourceful not only provides the primitives to build such a system but allows the user to make decisions and take action at run-time when provided with the results of a what-if scenario.

5 Future work and conclusion

A number of challenges remain as future work: (i) evaluating the precision and accuracy of the resulting data, thus characterizing the limits of our system – this would allow Resourceful to be used for setting fine-grained quotas or diagnosing software variability (ii) exploring whether some user-space instrumentation can be automatically added using either compiler-based methods or binary rewriting. (iii) extending Resourceful to the hypervisor level and across hosts.

Measuring resource consumption at a fine-grained level is extremely useful in understanding the behavior of a system and its interactions with the runtime environment. However, the main problem with measurements like these is the *probe effect* they introduce. The main purpose of our developed prototype is showing that it is feasible to obtain fine-grained accounting data without significantly altering the characteristics of the system we measure (which, in turn, means minimizing overhead). Longer term, we envision applications coordinating their resource usage or even trading resources based on given policies, using the same fine-grained accounting mechanisms.

References

[1] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 18–18.

[2] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. An analysis of linux scalability to many cores. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010), USENIX Association.

[3] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), ATEC '04, USENIX Association, pp. 2–2.

[4] Dean, J., and Barroso, L. A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.

[5] Dogar, F., Karagiannis, T., Ballani, H., and Rowstron, A. Decentralized task-aware scheduling for data center networks. Tech. Rep. MSR-TR-2013-96, September 2013.

[6] Erlingsson, Ú., Peinado, M., Peter, S., and Budiu, M. Fay: Extensible distributed tracing from kernels to clusters. In *ACM Symposium on Operating Systems Principles (SOSP)* (October 2011), ACM.

[7] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association, pp. 20–20.

[8] Ostrowski, K., Mann, G., and Sandler, M. Diagnosing latency in multi-tier black-box services. In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS 2011)* (2011).

[9] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., and Hundt, R. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro* (2010), 65–79.

[10] Shi, W., Zhang, L., Wu, C., Li, Z., and Lau, F. C. An online auction framework for dynamic resource provisioning in cloud computing. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2014), SIGMETRICS '14, ACM, pp. 71–83.

[11] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.

[12] V. Prasad, W. Cohen, F. E. M. H. J. K., and Chen., B. Locating system problems using dynamic instrumentation. In *Proc. of the 2005 Ottawa Linux Symposium* (2005), pp. 49–64.