Resourceful
Monitoring under the Microscope

LUCIAN CARATA, OLIVER R. A. CHICK, AND RIPDUMAN SOHAN

Lucian Carata is a Research Associate at the Computer Laboratory, University of Cambridge. He has done work in the area of provenance, root-cause analysis, I/O performance, and system measurement. lucian.carata@cl.cam.ac.uk

Oliver R.A. Chick is a passionate hacker of all things, from compilers to lower levels in the software stack. He earned his PhD from University of Cambridge with a thesis on understanding the impact of running complex workloads in virtualized environments. In the process, he devised new methods to achieve low-side-effect tracing (shadow kernels). oliver.chick@cl.cam.ac.uk

Ripduman Sohan is a Senior Researcher in the systems area at the Computer Laboratory, University of Cambridge. He has done work in the area of storage, virtualization, end-host networking, energy-efficient computing, provenance, and instrumentation. ripduman.sohan@cl.cam.ac.uk

Typically, monitoring systems record system-wide and application-level metrics *separately*, with significant time and expertise being invested in understanding how one affects the other when diagnosing complex issues. Resourceful, our open source project, bridges the gap between the two by allowing applications to record the system-level metric changes caused by each of their actions. For example, a Web server could record "the time spent in the TCP stack for servicing a request." We discuss the ideas that support this approach and provide a number of use cases showing how they can be useful in the real world.

The Usual Suspects

"Why is it slow?" (with the dreaded variant, "Why is it *sometimes* slow?") is a question that sysadmins have been asking ever since computer systems grew complex enough to run software. In response, common wisdom suggests deploying monitoring solutions such as Nagios and Munin to understand the status and evolution of production systems. More recently, open-source tools such as Prometheus, Heka, and Bosun have become popular by introducing ideas on tracking multi-dimensional time series that were battle-tested in companies with large computing infrastructures [6, 7]. They provide APIs with which software engineers can instrument their code to expose metrics for the monitoring system. The data ends up in customizable dashboards where it can be queried, used for alerts, or archived.

While there have been significant improvements in the number of available tools and low-overhead introspection mechanisms (perf, SystemTap [5], DTrace [1], eBPF), easily tying together the resources used and code paths touched inside the kernel while an application performs arbitrarily defined activities (such as executing a db query and sending back a response) remains a challenge, one which Resourceful (rscfl) sets up to solve. This is not about "fixing everything without waking sysadmins up," but exploring new design points and tradeoffs in the monitoring/debugging space that will make your life easier.

Key to this is programmability: we should start using tools that provide their results in ways that can be naturally consumed, either by dashboards, complex analysis tools, or by applications themselves, while placing everything they measure *in context*: in the context of what other applications/VMs are doing, competing workloads, and lack of perfect isolation. *No metric should be recorded without tracking the circumstances and effect it has on other metrics within the same time period.*

While at first sight simple, those initial ideas have led us to some less obvious design and implementation choices. By open-sourcing Resourceful, we hope both to start a wider discussion and to show the ability of solving some difficult real-world problems.

Resourceful: The Ideas

At its core, Resourceful allows applications to express interest in the measurement of fine-grained kernel-side metrics in order to understand the side effects of userspace actions when

interacting with the OS: Where was most of the time spent? Was their execution interrupted by the scheduler, and for how long? How did the statistics of the TCP stack (retransmits, bytes sent) change during this time? This takes an application-centric view, like a monitoring API would, but the measurements are about the OS and its resource sharing and multiplexing.

Exposing this data in a monitoring context allows gaining insights about real-time application behavior. Consider the case of a simple Web server: how would you track *per-request* page cache misses, time spent in the TCP stack, time spent doing I/O, or interactions with the servicing of other requests?

Measurements in Context

One of the significant differences between OS-level debugging tools and application-monitoring frameworks is the amount of detail they have about the running application: a monitoring framework may collect custom metrics specific to the application such as "number of client transactions per second," "time taken to run database queries fetching the front page," or "number of 404 errors per minute." This data may be collected together with per-system global metrics such as "TCP traffic," "I/O wait times," and "CPU load" and be displayed on the same dashboard for at-a-glance sanity checking.

However, once problems appear, it becomes *somebody's* (hopefully, somebody else's) task to figure out how things went wrong. How useful are dashboards in figuring out the problem? If "system-wide I/O wait times have increased while the number of transactions per second have dropped," do we have a better idea on where to look for what's causing the issue? Likely so, but only with enough experience and intuition about where the problem might be. That or a lot of trial-and-error. This is the stuff sysadmin "war stories" are made of.

We propose that it would be helpful to bridge the gap between application-specific and system-wide metrics. What if you could collect changes to system-wide metrics *in the context* of an application-specific one? What if you could have a metric of "I/O wait times for each request"? This is what rscfl is implementing through its API: applications declare the boundaries of interesting actions ("the request") and "announce" when they switch from one action to another, while an rscfl kernel module measures their kernel side effects. This can also be framed as a way of understanding what system resources are used by application-specific actions.

Integration as a Monitoring Solution

Although closer in implementation and low-level mechanisms to existing tracing tools, rscfl integrates with applications as a monitoring system would: it provides an API for collection of fine-grained metrics and allows applications to instrument code paths implementing a high-level functionality or activity (i.e., a Web server declaring "this is code for processing a Web request"). The resulting data can be further exported to inherently distributed monitoring systems such as Prometheus and be integrated in its larger monitoring infrastructure. Creating a root-cause diagnosis system like the one discussed by Ostrovski et al. [4] around this is definitely possible, and we have already built a prototype [8].

This position as the middle-man requires thought about programmability and efficiency: rscfl allows applications to access measurement results by sharing a region of memory between them and the kernel, giving direct access to results without extra copying or parsing of data. Do I hear you say, "That poses security issues"? We have looked at protecting the data as well: measurements are accessible as normal data structures within the application's address space, but by default no other applications have access to it.

Targeting the Kernel

The point at which any application interacts with the world outside its own memory address space is through the OS kernel: whether it is performing I/O, being scheduled together with other applications, or dealing with hardware failure, the kernel is the one doing the management. Our experience has been that these kernel interactions are typically some of the hardest to understand: the kernel is usually part of the code base that developers and sysadmins would like to treat as a black box that "just works."

On the other hand, you might be forced to learn about details inside the box once an application is not behaving as expected, and you're trying to find its bottlenecks. We propose solving this disconnect by explicitly exposing the notion of a kernel subsystem when returning measurement data. It seems like the right level of abstraction to talk about the kernel from an application's perspective: "It has spent this much time in the TCP subsystem"; "The file was not cached, so reading from it used the block subsystem."

In terms of actual measurements, the kernel remains the ideal place to understand the side effects of application actions: it is where resources (CPU, memory, disk) are being shared and time-multiplexed among multiple processes. However, adding lots of instrumentation can be costly. In existing probing mechanisms, the time taken to execute some measurement also depends on the total number of probes that are active. This is why rscfl uses a new type of low-overhead probing, called a KAMprobe (Kernel Advanced Measurement probe). Its execution time only depends on the complexity of the code being run inside the probe, with no dependency on how many other probes are active. We've been running kernels with tens of thousands of active measurement points without a significant performance impact.

Virtualization Awareness (Alpha)

Virtualization introduces new challenges in the picture, with multiple containers or VMs isolated to various degrees from each other on the same host, but introducing resource sharing (time, page cache, memory) that applications or OSes are not directly aware of. We have added hypervisor and containerized kernel support in Resourceful in order to be able to track those elements in the context of application actions: with this support, applications are aware (for example) of the time the VM was not scheduled in as part of the time added to the latency of particular actions. This introduces security considerations for cloud environments, but exploring this area for better understanding of workload co-location properties is very important.

Case Studies

Beyond the general configurable framework that allows anybody to extend Resourceful for tracking custom-defined kernel subsystems, we have investigated a number of use cases, generally connected to making our own systems research and problem troubleshooting easier. They are useful as examples of how the ideas presented above come together in a coherent manner.

Advanced Cache Monitoring

In a production system, caches are some of the key elements for maintaining good performance, yet keeping track of their behavior under complex workloads remains painful, with only coarse-grained summaries available at the OS-level. Collecting fine-grained information is unpopular because nobody likes slow caches: any measurement performed on them is by definition on a hot code path, where every cycle spent counts.

What about measuring things in a test environment? That doesn't often work since it's impossible to know and replicate production cache behavior—especially for shared environments like the cloud. Thus, monitoring by getting periodic snapshots of metrics like hit/miss ratios and eviction rates is typically the only realistic option. Still, wouldn't it be nice to be able to dig deeper and drive optimizations by having a map of what files were hit/missed in OS caches during different operations performed by your application?

We thought the same and leveraged Resourceful's low overhead probing mechanisms to define a PAGE_CACHE measurement subsystem. As the name implies, it tracks the OS-level page cache (normally used for file I/O, mmap, or fs metadata). Developers can choose to monitor the full cache or restrict the parts of the cache that are tracked (not interested in mmaps? why pay the overhead?). On the application side, data collection for this subsystem can be enabled through the API. When per-action aggregations are needed, their boundaries will need to be marked by API calls as well (e.g., for a Web server, mark parts of the code servicing a request or switching between them). Table 1 shows a more detailed comparison with other available mechanisms.

The result allows an application to record per-file cache statistics and give a better idea of when I/O latency degradation happens due to cache trashing. Knowing which files have incurred the most misses in the context of a particular action allows you to make informed compromises: does ensuring a particular file is cached make the code path you're interested in faster?

We have used the same functionality to characterize slowdowns caused by workload transitioning from being fully served from the cache to requiring disk accesses. In such cases, bottlenecks can shift (e.g., network-bound operations becoming disk-bound) for just part of your application, making diagnosis hard.

We're currently working to add visibility into evictors (who eliminated the cache entry that caused my process to miss?) and virtualized environments that hide shared caches (containers).

Hidden Work

The Linux kernel is able to run its own long-lived threads (kthreads) that are treated by the scheduler as any other process. They are used by the kernel to deal with long-running work (e.g., writing dirty page cache entries back to disk) or with work that cannot be completed immediately in regions where blocking is not allowed (such as interrupt service routines). In the latter case, the kernel provides a general mechanism that drivers can use to schedule delayed work: work queues.

However, work queues use a thread-pooling model where a number of long-lived kthreads wait for work to be enqueued from various subsystems and take on the execution of callback functions doing the actual work as needed. Due to this multiplexing of work belonging to different kernel subsystems and drivers, and due to the inherent asynchronicity, it is quite challenging to get a high-level understanding of what work is being carried out by a given work queue/kthread at a given time and to determine what high-level userspace action might have required its triggering.

We have defined a custom rscfl subsystem named TRACK_WORKQUEUE to help us in understanding why applications using an nvme device driver we extended were not achieving the expected throughput and latency figures. It has allowed us to monitor the creation and queuing of work inside kernel work queues as I/O requests from a benchmarking framework (fio) were issued.

This targeted investigation (monitoring the calls into the work-queue subsystem from within just a single application as opposed to system-wide) has allowed us to quickly determine that instead of using the inherent nvme parallelism, our modified driver was serializing block device requests through a single-threaded work queue. Having identified the bottleneck, it was an easy fix to increase the number of dedicated workers for that work queue, leading to significantly improved performance.

	Operation Selection	Aggregation	Metrics	Breakdown
rscfl	• selective (on file read, write, mmap) • all	• per-app • per-app action (programmer defined)	• hit/miss ratio • eviction rate • dirty entries • cached size	• per-file • summary
OS	non-selective (all)	system-wide, per-cgroup		summary
Tracing	non-selective (all)	system-wide, per-app	custom	summary

Table 1: Options offered by rscfl when monitoring caches, in comparison to default OS metrics and tracing mechanisms such as SystemTap, DTrace, or eBPF

Latency in Context

As a fully application-facing example, we have modified a non-blocking Web server to use the rscfl API for tracking resources consumed while servicing each client request. The resulting data is both pushed into Prometheus as a time series and used by a monitoring dashboard in order to understand variations in latency on a per-request basis. We've called this "Latency Explorer," a tool that dynamically allows us to compare high-latency requests with low-latency ones and try to determine where the differences appear. This provides more visibility into one of the areas of high interest for understanding any high fan-out architecture, where tail latency matters greatly [2].

In Figure 1, two views of the system are made available: a latency distribution and a per-request resource consumption breakdown based on Resourceful data. Each of the parallel axes in the bottom graph identifies a consumed resource or metric specific to the application activity (here, responding to an HTTP request). A given request is thus represented on the graph as a line linking the corresponding measured values (the dashed line in Figure 1). An idea of visual analysis using this data is to allow the selection of different intervals in the latency histogram while coloring the corresponding requests differently in the resource consumption graph (Figure 2). Further filtering is available on each of the resource axes.

Using Resourceful

Until recently, Resourceful was developed at the University of Cambridge, and while we spoke openly about the tool, its implementation was considered too immature for release to the wider world. Realizing the buzz that Resourceful was building in academic circles, we have been hard at work for the past 18 months and are now in a position where we are open-sourcing Resourceful so it can be used to increase observability in production systems. Our project is available at github.com/lc525/rscfl, and we're accepting both suggestions and contributions. If you have a monitoring problem where you believe the existing tooling is inadequate or might benefit from the ideas presented here, we would welcome your contributions.

Requirements

The core of Resourceful is a system that modifies your running kernel to insert instrumentation. In order to safely apply this instrumentation we require some capabilities that may not be accessible on some systems:

- **Elevated access.** Resourceful can be run on any Linux kernel without requiring a reboot or modification to the kernel as stored on disk. This is made possible by Resourceful scanning the running kernel, determining the parts of the running code that should be measured, and then applying itself to these regions. Doing that typically requires some form of elevated privileges. However, once rscfl is running it can be used by any application.
- **Kernel debug symbols.** Resourceful has an automated analysis that determines boundaries in the kernel that should be measured. To enable us to perform this analysis, Resourceful requires access to the kernel's debug symbols. In most Linux distributions these can be obtained as a separate package that does not modify the kernel that is running (i.e., the debug symbols live in a separate file and do not affect the running kernel).

Installation

At present Resourceful must be built from Source, however we are considering packaging it for some distributions. We maintain and provide a full set of up-to-date instructions on running Resourceful on our GitHub page, but here we outline the sets required at the time of writing.

- Installation requires you have Git, Wget, and Python 2.7 installed. We expect these will be installed on most Linux boxes.
- Beyond that, it should be as simple as running `make` and `make install`.

Modifying Programs to Use Resourceful

Resourceful supplies a C/C++ API with which userspace programs specify where they start and stop processing a given activity. While this does mean that applications need to be modified in order to use Resourceful, the changes in practice are often trivial and can be added to commonly used remote procedure call libraries in an elegant fashion. The context for mea-

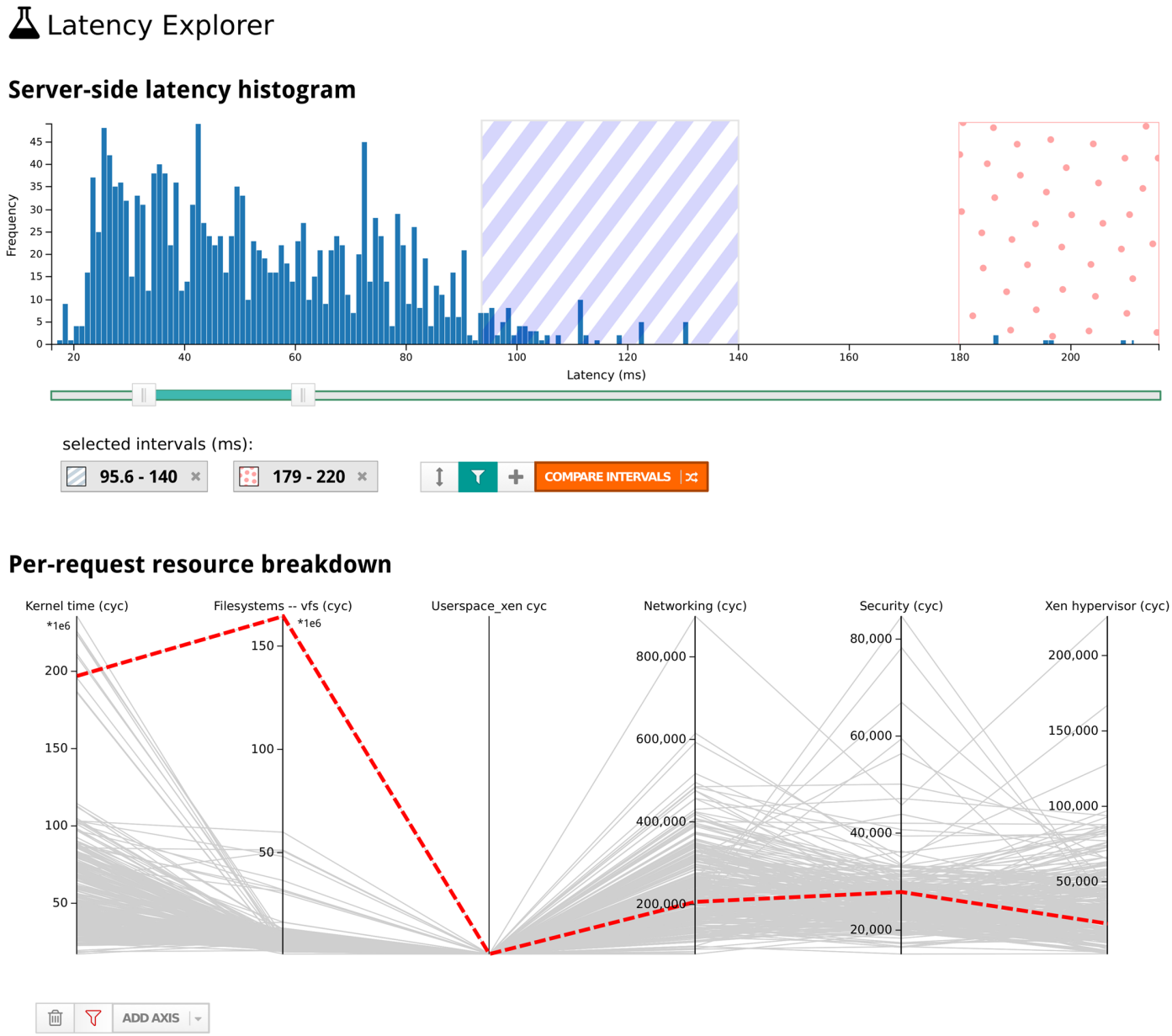

Figure 1: Latency Explorer, a visual analysis tool prototype

surements is being kept by communicating some opaque tags to the kernel. This is not unlike the strategy taken by other systems such as XTrace [3], but we are considering asynchronous behavior in greater detail. When receiving a tag that is the same as one seen before, our kernel module knows that any metric changes should be accounted to the same activity, and it can perform the aggregation directly in kernel space.

The general steps for using the API would be as follows:

1. Initialize Resourceful in your program. This creates a Resourceful "handle," which is much like a traditional file descriptor. It is passed to the other Resourceful functions and contains state about the innards of Resourceful.

```
rhdl_t rhdl = rscfl_init( ) ;
```

2. When your application starts a new activity (i.e., receives a user request), it can request a "token" for it and start accounting the resources it uses:

```
token_t token ;
rscfl_acct(rhdl, token, ACCT_START);
```

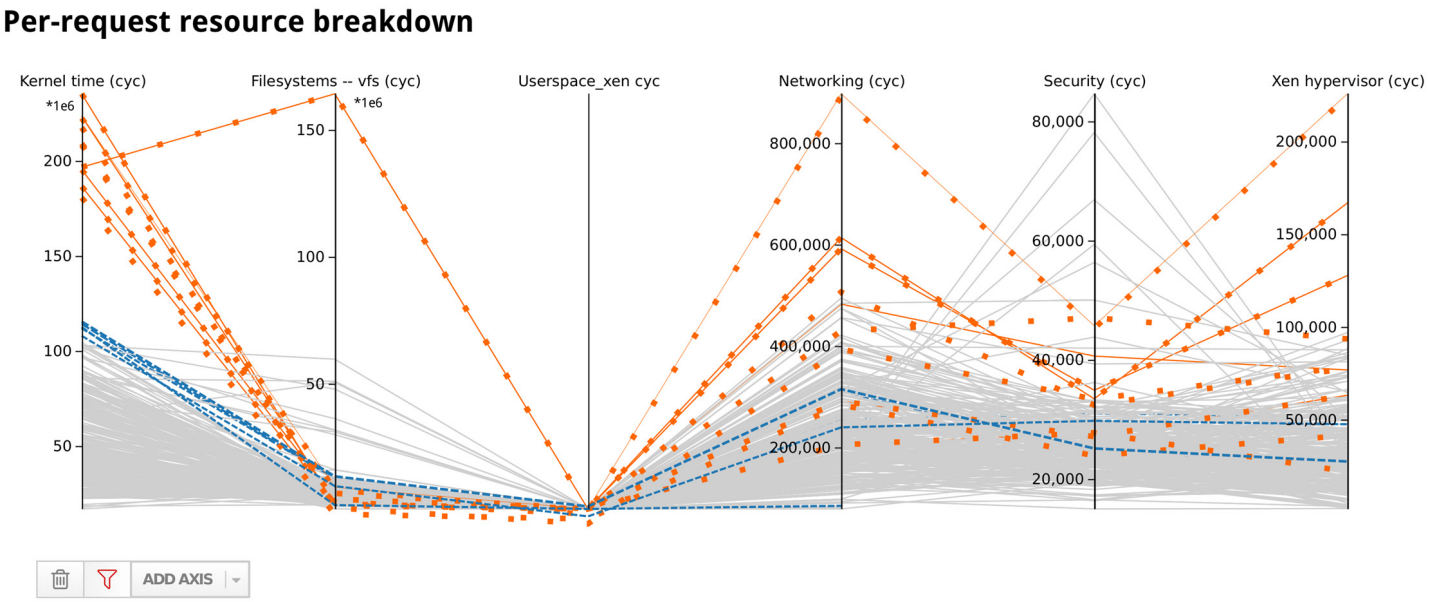

Figure 2: Latency Explorer, interactive filtering for comparing the latencies of requests selected in the Figure 1 histogram: tail latency (dotted) vs latencies between 95.6 and 140 ms (dashed). Each axis can be further filtered, and that in turn updates the histogram (how does the histogram of response times look for requests that spent a lot of time in the Networking layer?).

3. When the activity stops (i.e., the request has been sent), we can stop recording the resources used and read out the values:

```
rscfl_acct(rhdl, token, ACCT_STOP);

// Read the accounting information that we recorded.
rscfl_account_t rscfl_results;
rscfl_read_acct(rhdl, &rscfl_results);
```

`rscfl_results` is a structure from which you can read the kernel resources used in the processing of your request. This is a broken down per-kernel subsystem. For this example, we have measured a default list of performance measurement counters, however Resourceful also has APIs that can be used to measure specific resources. Resourceful also contains some magic higher-order functions that let you perform advanced aggregation of resources used across many requests (map-fold-filter).

4. In modern systems, processing often takes place in asynchronous event loops. This means the application activity might complete in stages. If this happens you can tell Resourceful to apply the resources used to a new activity by switching token:

```
rscfl_switch_token(rscfl_hdl, new_token);
```

The API also provides features for storing arbitrary application-specific metrics together with the kernel-recorded measurements, which is extremely useful when performing a detailed analysis.

Upcoming Features, Conclusion

Resourceful's API is currently available for C and C++ only, but we hope to add wrappers for other popular languages soon. In particular, this presents a good opportunity for instrumenting runtimes that provide green threads. Those can be tricky to monitor at present, and by instrumenting at the runtime level we would also limit the amount of required changes to application code. Other planned features target the extension of our visibility into virtualized environments, and we already have promising research results in that area.

We're not aiming to produce just another tool for debugging/monitoring applications. Instead, we're hoping to restart a discussion on what is needed to advance this area in ways that are helpful to practitioners. Download from github.com/lc525/rscfl and let us know what you think.

References

[1] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proceedings of the USENIX Annual Technical Conference (ATC '04)*, pp. 2–2: https://www.usenix.org/legacy/event/usenix04/tech/general/full_papers/cantrill/cantrill_html/.

[2] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM,* vol. 56, no. 2 (February 2013), pp. 74–80.

[3] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A Pervasive Network Tracing Framework," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI '07)*, pp. 20–20: https://www.usenix.org/legacy/events/nsdi07/tech/full_papers/fonseca/fonseca.pdf.

[4] K. Ostrowski, G. Mann, and M. Sandler, "Diagnosing Latency in Multi-Tier Black-Box Services," in *Proceedings of the 5th Workshop on Large Scale Distributed Systems and Middleware (LADIS '11)*: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37477.pdf.

[5] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating System Problems Using Dynamic Instrumentation," in *Proceedings of the 2005 Ottawa Linux Symposium (OLS '05)*, pp. 49–64: https://www.kernel.org/doc/ols/2005/ols2005v2-pages-57-72.pdf.

[6] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers," *IEEE Micro,* vol. 30, no. 4 (July/August 2010), pp. 65–79.

[7] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google Technical Report, 2010: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36356.pdf.

[8] J. Snee, L. Carata, O. R. A. Chick, R. Sohan, R. M. Faragher, A. Rice, and A. Hopper, "Soroban: Attributing Latency in Virtualized Environments," in *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '15)*: https://www.usenix.org/system/files/conference/hotcloud15/hotcloud15-snee.pdf.