ddtrace: Rich performance monitoring in distributed systems

Benjamin Braun*, Henry Qin*
bjmnbraun@gmail.com, hq6@cs.stanford.edu
*Stanford University

***Abstract*—Through performance monitoring, application developers and maintainers can find software bugs, performance bottlenecks, and source of tail latency in their applications.**

In this paper, we develop a general-purpose performance monitoring library, ddtrace, designed to be easy to integrate with message- and Rpc-based systems which issue child requests as part of handling a single client request, such as the RAMCloud storage system. We demonstrate that ddtrace can analyze Rpc performance across multiple machines.

The strengths of ddtrace are twofold.

- **Low overheads. 130 nanoseconds per interval of code measured, 5x faster than the next fastest trace logging library we could obtain and test.**
- **Scriptability. Rich performance monitors can be written in a widely-used language (currently, C++) using the ddtrace API.**

Furthermore, these scripts are decoupled from the application, so the application is protected from faulty performance monitors and vice versa, and performance monitors can be reused across multiple applications built with ddtrace support.

I. Introduction

In traditional applications, the process of identifying and eliminating performance bottlenecks is relatively well-understood. Various profiling tools are already very mature and widely used [11][3][1].

As the systems community has moved to a large-scale datacenter-oriented model of computing, and applications increasingly require communication with and synchronization across multiple machines in order to service requests, it becomes correspondingly more difficult to understand the sources of latency in these applications. This understanding is particularly important for online services, and the back-end dependencies of online services, which often have to meet strict Service Level Agreements (SLAs). Given that some of these SLA's are focused on tail latencies [6], the ability to collect detailed performance data for all requests becomes particularly relevant.

In an effort to fill this void, researchers developed the X-trace software library[8]. Unfortunately, X-trace has high overheads - we observed an average cost of $19\,\mu$s to the application process simply to log a single trace through X-trace. In the RAMCloud storage system, one of the authors has expended substantial time to understand the latency breakdown of requests as they traverse multiple servers over the course of single client request. Because RAMCloud has read latencies on the order of $5\ \mu$s and write latencies on the order of $14\ \mu$s, a $19\ \mu$s overhead is not tolerable, so this author had to build ad-hoc solutions and manually reassemble the performance data from multiple servers.

In the current work, we design and implement a general purpose library for distributed performance analysis which will be much lighter weight than any previous system, while recording enough metadata to infer causal relationships between requests. More specifically, for any given user request, we can quantify, with cycle-counter level granularity, the time each server spent processing it and the sub-requests that it generated, as well as the number of L2 and L3 cache misses it generated on each of those servers.

II. Existing performance analysis tools

For non-distributed applications, statistical profiling tools such as perf suffice for finding the most expensive parts of the application [3]. However, these tools are insufficient for monitoring distributed applications because they do not provide a unified view of an application's performance across multiple machines.

Performance monitors for distributed applications have been developed [8]. X-trace[8] is a framework for causal tracing of a networked system across the boundaries of heterogeneous systems. When testing X-trace on our own, we observed costs as high as $19\ \mu$s to log a single

event. This high overhead is unacceptable for a performance monitor designed to work in latency-sensitive distributed systems [12][5], where requests can be processed in a matter of microseconds. The X-trace server is also not scalable [14]. Our system borrows some ideas from X-trace; specifically, we tag each top-level user request with a Universally Unique ID (UUID).

Dapper[13] is an internal distributed profiling tool used at Google. The technical report states that a span, the equivalent of a ddtrace interval, can be created and destroyed in as little as 176 ns. Dapper employs an optimization that allows applications to add a lightweight checkpoint annotation mid-interval without creating a new trace record. Dapper also employs sampling to reduce the number of spans actually created, reducing latency and increasing throughput. We plan to implement these optimizations in a future version of ddtrace. Unlike Dapper, ddtrace measures hardware performance counters. Dapper logs every trace to disk using a daemon and these logs are later processed by a central aggregator. ddtrace is more general and allows applications to provide custom data consumers that can perform analysis on data traces as the application produces them, sometimes avoiding the need to log every trace to disk.

SystemTap, a port of the DTrace Solaris OS framework to Linux, is a scriptable trace logger with customizable trace formats [7] [10]. Systemtap can be remotely invoked, allowing these scripts to run on a cluster of computers. In our testing, we observed that the cost to log an event through Systemtap is around $1\ \mu$s.

LTTng is a competitor to Systemtap with fewer features but designed to be higher performance[9]. In our testing, we observed that the cost to log an event through LTTng is around 500 ns. Our system uses SPSC-queues, which is similar to LTTng's multi-buffered strategy for logging records.

Apache flume is a system for constructing record streams that may span multiple computers. Flume is highly configurable, and appears to be used in a number of high profile projects. Unfortunately, we were unable to find any published performance numbers for Flume, and did not have sufficient time to test it with our systems because it is written in Java and our system is written in C++.

III.Goals

We have three objectives for our performance monitoring tools.

1) **General trace analysis with causality.** We want to generate causality diagrams in the spirit of X-trace but with performance data to supplement the causal data.
2) **Minimal impact on application performance.** This goal encompasses both normal-case run-time performance and failure modes of the measurement system. In the normal case, we aim to limit the overhead of performance collection as much as possible. If our monitoring system crashes, restarted, or runs slowly, the application being monitored should not suffer any loss in performance.
3) **Automated tail request identification.** We would like to automatically identify and report requests violating user-specified SLAs.

IV. ddtrace Design Overview

ddtrace combines the scriptability of D-trace with the distributed use case and ID tracking of X-trace. At a high level, the system enables a client application to log the changes in hardware performance counters across arbitrary intervals of code within a single thread, while simultaneously maintaining causality of intervals as a request migrates across threads on a single machine and across the network to remote machines. ddtrace maintains a vector clock containing an application-provided universally unique request ID together with an application-provided unique server ID. This vector clock is logged with every interval, enabling us to collect and order the intervals associated with any particular request.

Usage and API for Target Application

Our user-facing API has two parts. The first part is encapsulated in the Interval class and is intended for use by the target application.

On each server, the client initializes the library by calling the `init` function and providing a unique server ID and the set of hardware counters it is interested in reading. ddtrace currently supports three architectural counters which should

be available through the `rdpmc` instruction with the appropriate configuration on all modern Intel machines [2].

- L2 Cache Misses
- L3 Cache Misses
- Userspace Cycles

After initialization, within each thread, the application code constructs an `Interval` object and calls `start()` and `stop()` to specify the interval of interest. A call to START increments the component of the local vector clock corresponding to the current server ID and reads the value of the user-specified performance counters. A STOP call again reads the performance counters, and then computes the difference in these counters, and logs a record for the interval. Pairs of calls to START and STOP must be disjoint with a single thread. For convenience when making finer-grained measurements, we also support the `checkpoint()` API call, which is equivalent to calling STOP after a previous START, and then immediately calling START to measure the next interval.

Usage and API for the Data Consumer

When data is logged by the API calls made in the previous section, it is sent through a shared memory pipe to a separate process on the same machine as the monitored process which we shall henceforth label the Data Consumer. This Data Consumer is a simple C++ program which the user of our library will write, and it uses the second half of the API to pull data from the other end of the pipe, using a queue abstraction. More specifically, the Data Consumer will initialize an object called a `RecordSource` and repeatedly invoke the `popRecord()` method on it to get the next record.

If the data consumer does not pull data from the pipe sufficiently quickly, the pipe should simply overflow and spill out packets in FIFO order based on when they were received. Under our current implementation, we instead drop packets at the producer end of the pipe until the pipe is emptied by the consumer, but this will be corrected in future work.

Performance Trace Aggregation and Monitoring
The last piece of the puzzle towards attaining our goal of a distributed performance trace with causality requires two additional components, which we name the Intelligent Data Consumer and the Live Monitoring Master, both of which are under development.

The Intelligent Data Consumer is a general-purpose Data Consumer which performs two tasks on receiving a data record.

1) Log the data record to the local disk, for later offline analysis and generation of performance-annotated causal diagrams. It simultaneously buffers the last N records in memory, where N is an application-dependent parameter.
2) Detect whether the latency recorded in the record is above some user-defined SLA. If so, it will send a message to the Live Monitoring Master, which will request the latest buffered records from all instances of the Intelligent Data Consumer, collect the traces associated with each recent user request, and generated the performance-annotated causal diagram for just those those requests.

The second component is a Live Monitoring Master, which will report SLA violations and actively request additional data to assist online diagnosis. In addition to coordinating requests and as described above, it will have a user-facing component so that a system administrator can easily view the causal diagrams for recent SLA violations.

Practical Use in a Distributed Application

In this section, we discuss the code changes necessary to make ddtrace work in RAMCloud [12], a distributed, low-latency key-value store.

The Rpc request and response headers would need to have a `VectorClock` added to them. This VectorClock will be completely empty in the client's initial request. The first server to receive the request will create a globally unique `RequestId`, and initialize the VectorClock with this request ID.

The server then needs to allocate storage for the VectorClock with a lifetime equal to the lifetime of the request and copy the VectorClock out of the request message. It will pass a pointer to this

VectorClock to any Interval objects it initializes. Finally, whenever a server issues a new Rpc as a part of procesing the current Rpc (such as ReplicationRpc), it must copy the current VectorClock into the request header of that child Rpc.

This VectorClock management logic, together with our API calls, are sufficient to make our system fully functional.

V. Implementation

ddtrace ddtrace at its core creates and serves `IntervalRecords`, which have the following specification:

```
struct IntervalRecord {
    uint64_t startCycles;
    uint64_t endCycles;
    VectorClock clock;
    PerfRecord countersDiff;
}
struct VectorClock {
    uint64_t id;
    uint64_t length;
    VectorClockEntry entries
        [MAX_VECTORCLOCK_ENTRIES];
}
struct VectorClockEntry {
    uint16_t serverId;
    uint8_t count;
}
struct PerfRecord {
    uint16_t counterSelector;
    uint64_t counters
        [MAX_COUNTERS_PER_RECORD];
}
```

Records are produced by the application thread and communicated to a data consumer residing on the same machine through a shared memory mapping. This shared memory mapping has an associated file in a subdirectory of `/dev/shm`. A data consumer discovers running application threads by scanning this directory for files, and we also use a shared memory version counter in the directory to prevent unecessary scans over the directory. The consumer detects "dead" shared memory files that are no longer associated with any running thread using the `fuser` command to count processes referencing a file. When serving records, a consumer round robins over all shared memory files it has discovered. This is illustrated in Figure 1.

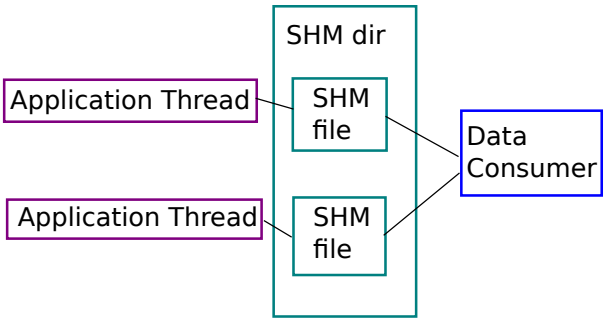

Fig. 1: Diagram of the implementation of ddtrace.

We use the `rdpmc` x86_64 instruction to query hardware performance counters and the `perf_event_open` system call to configure and enable these counters.

The `startCycles` and `endCycles` fields of `IntervalRecord` are populated by reading from the local CPU cycle counter, using the `rdtsc` x86 instruction.

VI.Evaluation

A. Experiments

Target application We designed a synthetic "mock RAMCloud" application that has timing and threading characteristics similar to that observed in RAMCloud write request processing. We did not benchmark the real RAMCloud system because our system currently uses C++11 features which were not available in the version of libc on the RAMCloud test cluster, and we had not yet completed the implementation of our Live Performance Master or our Intelligent Data Consumer. This synthetic application repeatedly processes a simulated write Rpc with single replication whose processing always follows the following pattern. First, a dispatch thread receives the Rpc, and does some work, finally handing off the Rpc to a worker thread. The worker thread then does some work, and then makes a sub-Rpc. The sub-Rpc is assumed to just involve a short amount of straight computation, for simplicity. When the sub-Rpc returns, the dispatch thread sees the return and hands it off to the waiting worker thread. The

worker thread then does some more computation and finally returns the result of the Rpc. We simulate this application using busy loops that spin for set amounts of time to simulate either network latency or computation. We generate an `IntervalRecord` for each interval of code delimited by thread handoffs and network hops. The result is a function that spins for a total of 11μs and logs 4 records. We make this mock RAMCloud application multithreaded by performing this simulation in parallel on multiple threads.

We have implemented two simple consumers: a counter consumer and a disk-logging consumer that. The counter consumer simply counts incoming records but does not process them. We use the counter consumer in experiments to illustrate a consumer operating essentially at the maximum possible poll rate. The disk-logging consumer is currently single threaded, and it buffers records in memory before writing them to disk. We buffer at most 1000 records in memory for this purpose. While writing this buffer to disk, the poll rate of the disk-logging consumer drops.

Experimental setup. We configure ddtrace to include the value of a HW performance counter that measures userspace cycles with each log interval.

All experiments are run on a machine with 2.3GHz Intel Xeon CPUs with 32KB L1I and L1D caches, a 256KB L2 cache, and 15MB L3 cache. The machine is a NUMA architecture with 2 sockets each containing 6 physical cores. The machine uses a Toshiba MG03ACA1 7200 RPM spinning disk.

All numbers shown have standard deviations that are within 5% of the mean.

Microbenchmark We measured the amount of time it takes to call START followed by STOP on a single thread for various queue size limits. To do so, we wrote a simple application that calls START followed by STOP in a tight loop, with a small amount of spin time in between each log entry, while running the counter consumer. Figure 2 shows the distribution achieved after logging 5 million pairs for each queue size. In this experiment, the consumer was able to keep up with the rate at which the application produced records.

Next, we tested how the latency changes for measuring and logging intervals when multiple threads are used. The results are shown in Figure 3.

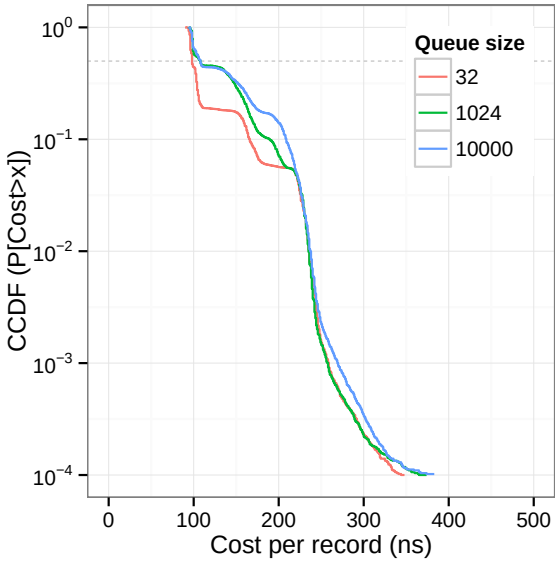

Fig. 2: The cost of calling START followed by STOP can be 60 nanoseconds higher when using a large size for the record buffer queues compared to using a small size. The graph plots the probability that the log operation takes longer than some number of nanoseconds for various sizes of the record buffer queue. Median latency is indicated by the dashed grey line. This additional cost is due to an increase in cache miss rates when the record buffer queue cannot fit in the caches. On our test machine, the queue exceeds the L1 cache in the 1024 queue size example and the L2 cache in the 10000 queue size example.

	Runtime (s)	Overhead
Baseline	1.12	-
ddtrace	1.19	6.3%

TABLE I: Using ddtrace to monitor the performance of a mock RAMCloud application adds a performance overhead of at most 6.3% when under maximum load. See text for details.

Impact on application performance We measure the time for the mock RAMCloud application using 4 threads to complete 100,000 Rpcs on each thread while running the counter consumer. The results of this experiment are shown in Table I.

Finally, when running either of our consumers with the mock RAMCloud application, we observe no dropped records.

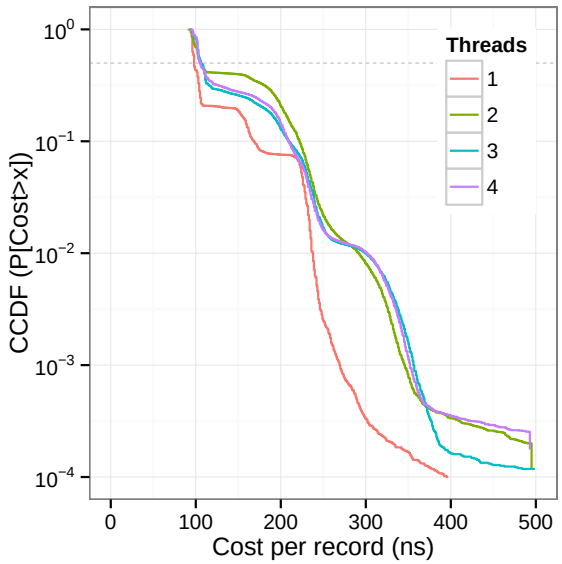

Fig. 3: ddtrace achieves a 99.9% tail latency of less than 375 nanoseconds for starting and then stopping an interval when 4 threads are used. The graph plots the probability that the log operation takes longer than some number of nanoseconds for various numbers of concurrent record producers. Median latency is indicated by the dashed grey line.

VII. Discussion and Conclusion

In this work, we describe and benchmark a general-purpose, low-overhead, distributed performance monitoring system for analyzing the performance of individual user requests as they propagate through multiple servers in a distributed system. We achieve a per-trace latency that is 5x lower than the next fastest publicly available logging library we could obtain and test (LTTng). We can generate causal diagrams depicting the latency breakdown for any arbitrary request in a stream of requests, and annotate such diagram with performance metrics to investigate cache and branch predictor behaviors.

We apply our system to develop a comprehensive understanding of latency in the RAMCloud storage system, as described in Appendix A. We believe that our system will be useful for a comprehensive understanding of other distributed systems such as Hadoop or GFS.

Source code: https://github.com/bjmnbraun/ddtrace.

References

[1] Google perftools. http://code.google.com/p/gperftools/?redir=1. Accessed 2014-12-04.

[2] Intel 64 and ia-32 architectures software developer's manual, volume 3b. http://www.intel.com/. Accessed 2014-12-04.

[3] Linux perf tool documentation. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed 2014-12-04.

[4] Ramcloud test cluster specs. https://ramcloud.atlassian.net/wiki/display/RAM/Cache+Latencies+and+Sizes+on+RAMCloud+Test+Cluster. Accessed 2014-12-10.

[5] Reddis home page. http://redis.io/. Accessed 2014-12-04.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[7] F. C. EIgler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen. Architecture of systemtap: a linux trace/probe tool. 2005.

[8] 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*, pages 20–20. USENIX Association, 2007.

[9] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais. Combined tracing of the kernel and applications with lttng. In *Proceedings of the 2009 linux symposium*, 2009.

[10] R. McDougall, J. Mauro, and B. Gregg. *Solaris (TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall PTR, 2006.

[11] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[12] J. e. e. Ousterhout. The ramcloud storage system. https://ramcloud.atlassian.net/wiki/download/attachments/6848571/RAMCloudPaper.pdf.

[13] 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 research*, 2010.

[14] W. Wang. End-to-end Tracing in HDFS. MSc Thesis, Technical report CMU-CS-11-120, School of Computer Science, Carnegie Mellon University, 2011.

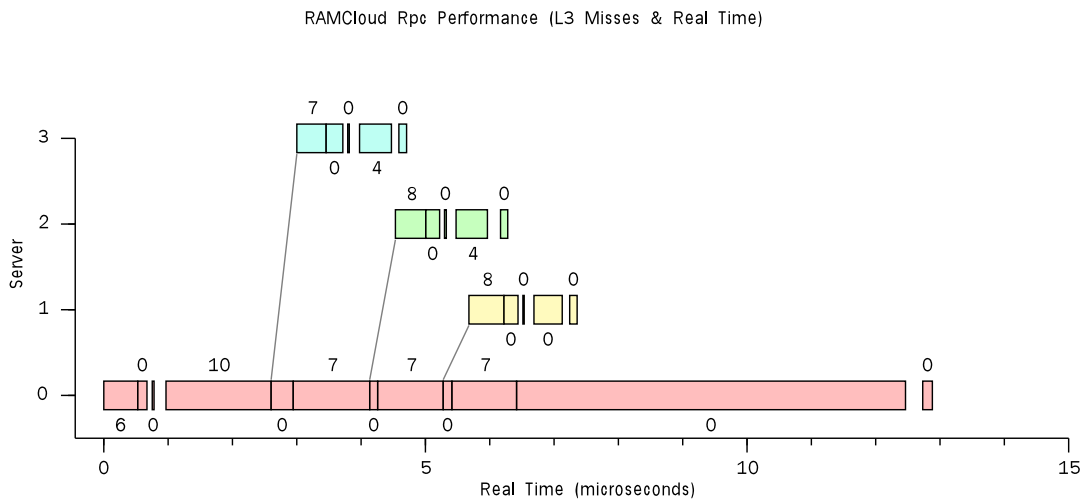

Fig. 4: Anatomy of a write Rpc in RAMCloud with three-way replication. Each box represents an span of code running on a server with the ID given on the vertical axis. The width and horizontal placement of the boxes represent real time measured using the cycle counter on the machine that the interval was measured on. The numbers above and below the boxes correspond to the number of L3 cache misses taken during that span of code. Diagonal lines represent a replication Rpc going to a backup server.

Appendix

A. Application to RAMCloud

After a substantial backporting effort, we made our library compatible with `gcc 4.4.7` and linked RAMCloud [12] against it. We then ran two experiments to understand distribution of L3 cache misses, and to evaluate the overhead of our library on a real system. In the RAMCloud test cluster [4], every machine runs a 2.4 Ghz Intel Xeon processor with 4 physical cores and a Seagate SSD. Each core has a private 32 KB L1 Cache and 256 KB L2 Cache, while all the cores share a 8 MB L3 cache.

1) Performance Experiment Under Normal Configuration

In this experiment, we configured RAMCloud with 3-way replication (one in-memory copy, three on-disk copies), with four RAMCloud masters which are also running backups, one coordinator and one client sending back-to-back write requests. We initialized a table with two million key-value pairs, and then capture the traces using ddtrace while doing $100,000$ overwrites. An overwrite is an update to the value of a key value pair where the key is already assigned to some value.

For this experiment, we defined 11 intervals on the data master and 5 intervals on each backup, for a total of 26 intervals measured. Figure 4 shows the performance of one of the requests in this experiment. The width of each of the blocks corresponds to an interval measured, and the numbers above or below the blocks show the corresponding L3 cache misses.

The intervals correspond to events which are meaningful to the application in question such as polling the NIC, unmarshalling the packet and sending a replication Rpc.

2) Overhead Experiment Using a Single Replica

In the previous experiment, there were 3 replicas with partially overlapping work, making it difficult to evaluate the overhead of our framework accurately.

In order to evaluate our overhead, we repeated the experiment with single replication (one in-memory copy, one on-disk copy), for a total of 16 intervals.

The baseline latency for this write is 12.26μs, and with 16 intervals enabled is 14.33μs. Thus, the latency added for each interval added is 130 ns at the median.