Studying multi-threaded behavior with TSViz

Matheus Nunes[1], Harjeet Lalh[2], Ashaya Sharma[2], Augustine Wong[2], Svetozar Miucin[2], Alexandra Fedorova[2], Ivan Beschastnikh[3]
[1]Computer Science, Universidade Federal de Minas Gerais
[2]Electrical and Computer Engineering, University of British Columbia [3]Computer Science, University of British Columbia

***Abstract*—Modern high-performing systems make extensive use of multiple CPU cores. These multi-threaded systems are complex to design, build, and understand. Debugging performance of these multi-threaded systems is especially challenging. This requires the developer to understand the relative execution of dozens of threads and their inter-dependencies, including data-sharing and synchronization behaviors.**

We describe TSViz, a visualization tool to help developers study and understand the activity of complex multi-threaded systems. TSviz depicts the partial order of concurrent events in a time-space diagram, and simultaneously scales this diagram according to the physical clock timestamps that tag each event. A developer can then interact with the visualization in several ways, for example by searching for events of interest, studying the distribution of critical sections across threads and zooming the diagram in and out. We overview TSviz design and describe our experience with using it to study a high-performance multi-threaded key-value store based on MongoDB.

A video demo of TSViz is online: https://youtu.be/LpuiOZ3PJCk

I. Introduction

Moore's law is dying and modern systems include increasingly more cores to help applications improve their performance. For example, the latest iPhone 7 has 4 CPU cores and 6 GPU cores. Server systems are built with as many as 80 cores. To utilize these cores the software must embrace concurrency. Today this is achieved by instantiating and managing multiple threads and by building systems that require high performance with support for extensive multi-threading. Concurrency, however, improves performance at a cost to complexity. For example, a system can reap higher performance only if the threads are able to do useful work uninterrupted and in parallel. Achieving this state requires the developer to carefully orchestrate thread synchronization.

When a system exhibits poor performance, the developer has a substantial challenge: she must understand the runtime thread behaviors and determine why certain threads are not performing as expected. There are several tools for this, most of which are profilers. Profilers, such as `perf`, are excellent at capturing aggregate properties of the execution: they can inform the developer about the approximate fraction of CPU cycles used by each function, or tally up the contribution of each function to the overall hardware cache misses or operating system page faults, and even associate specific events, such as branch mispredictions, with lines of source code.

However, there is an important kind of information that profilers are fundamentally unable to provide, which makes them limited to helping with a certain class of performance anomalies. Profilers obtain performance information by sampling events during the execution; they do so by periodically interrupting the running program to record callstacks and various hardware or operating system metrics. This information is not sufficient for performance problems that require examining the steps of the execution in the small and in answering questions like: what was the sequence of executed functions during a high latency event? Or, which memory locations were accessed and by which threads? Or, were any threads blocked on locks?

To answer the above questions about complex multi-threaded behavior developers need additional tools to expose individual thread interactions and gather detailed data, and they need tools to effectively study this data.

In this paper we describe a tool called *TSViz*, designed to help developers understand captured multi-threaded behavior in complex systems by presenting an interactive visualization of a captured execution trace. To use TSViz a developer must first trace their system to capture runtime thread activity. For this we rely on DINAMITE [8], an LLVM-based instrumentation tool (Section II), but developers may also use other tools, such as Pin [7], or even custom-generated application logs. A persistent developer can manually grep through execution logs, but *TSViz* offers an easier and more general-purpose means of understanding the captured traces. TSViz visualizes the captured traces with no effort on the part of the developer and allows the developer to immediately start exploring and answering questions about what happened during the execution.

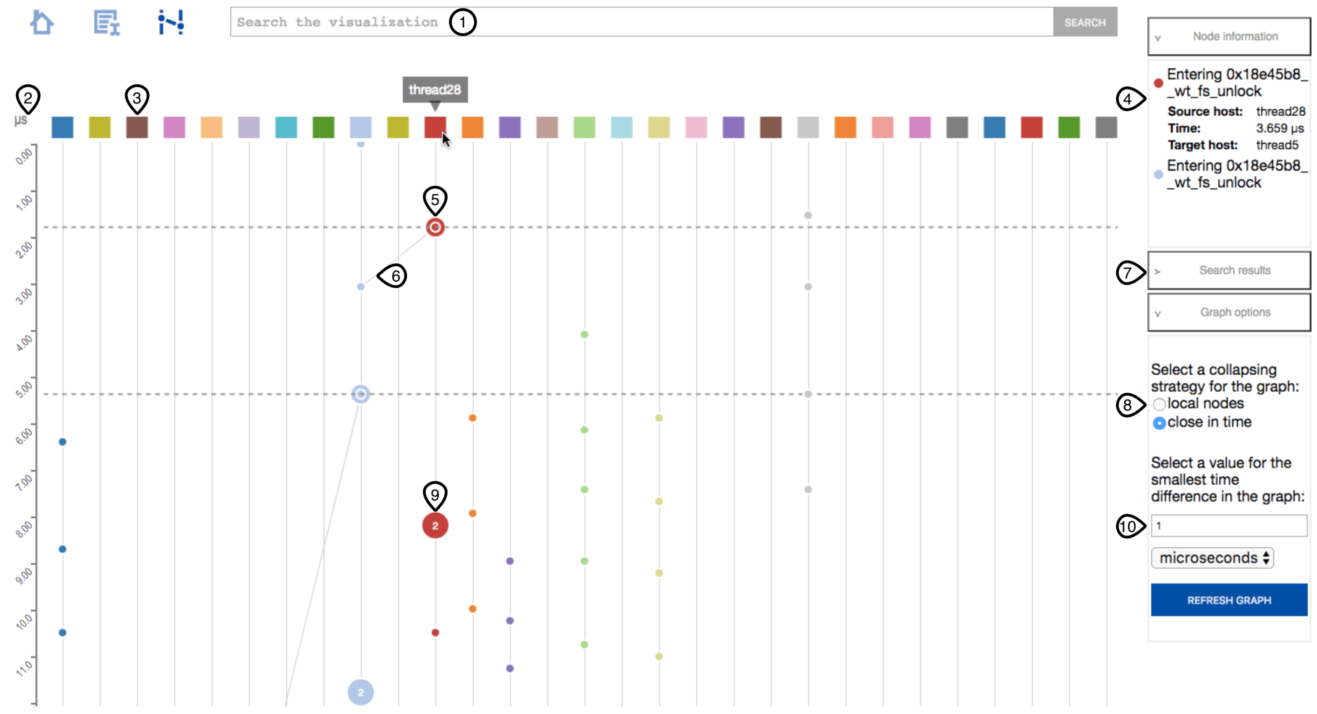
TSViz is an open-source tool that runs in the browser, requires no installation, and is publicly deployed[1]. Next, we overview DINAMITE and then detail TSViz. In Section IV we described our experience in using TSViz with debugging the performance of thread locking behavior in a high-performance multi-threaded key-value store. We then briefly discuss our next steps, overview related work, and conclude.

II. Tracing with DINAMITE

We previously described DINAMITE in [8]. Here we summarize its key features.

DINAMITE is an LLVM-based tool that statically instruments C and C++ programs to automatically record the following events during an execution: function entry/exit

[1]http://bestchai.bitbucket.io/tsviz/



● Entering 0x18e45b8_ _wt_fs_unlock
Source host: thread28
Time: 3.659 μs
Target host: thread5
● Entering 0x18e45b8_ _wt_fs_unlock

Select a collapsing strategy for the graph: local nodes [✓ close in time]

Select a value for the smallest time difference in the graph: 1 microseconds

Fig. 1. TSViz screen with key areas numbered. ① The visualization may be searched for events, for specific graph structures, or for events denoting intervals. ② The ruler denotes the scale for distances between events in the visualization. ③ A box represents a thread; box colors provide a consistent coloring for events associated with a thread. ④ Event details are shown in a panel to the right, and when two events are selected their distance in physical time is detailed. ⑤ An event on a thread timeline is represented as a circle. ⑥ Partial ordering between events is represented as diagonal lines. ⑦ Search results are detailed in the panel to the right. ⑧ A sequence of local events that are close to each other in physical time (or all local events in a sequence) are collapsed into a larger circle, e.g., ⑨, whose label indicates the number of collapsed events. ⑩ The graph can be zoomed in/out, which effects the ruler, the spacing among events, and the collapsing of events.

points, function arguments, memory accesses along with full debugging information (variables names/types, source code locations). Each type of event can be enabled independently. For each event DINAMITE records the timestamp, enabling us to use event traces for performance analysis. Reproducing real performance bugs in production software typically requires running the program with many threads for dozens of seconds; thus execution traces reach multiple gigabytes in size that are unwieldy for human consumption. Yet, the traces contain valuable information that can help the developer debug the most difficult performance problems if presented in the right form. We found TSViz essential for that purpose.

III. Visualizing multi-threaded traces with TSViz

TSViz is designed to run in the browser and does not require the user to install any software. To use the tool the developer must upload a log, possibly generated by DINAMITE, and enter a set of regular expressions to parse this log. There are three required reg-ex capture groups: event physical timestamp, event logical timestamp, and event text. The developer can also define and parse out user-defined fields for each event (e.g., the number of database records processed by the thread up to this point).

Figure 1 illustrates the main TSViz screen that is displayed once the developer uploads the log, the regular expressions, and instruct TSViz to analyze the trace. The figure visualizes the trace used in the locking algorithm case study (Section IV-A). The screen is dominated by a time-space diagram of the trace: individual threads are represented with vertical timelines, one per thread and from left to right. Events are positioned on these timelines *to scale* using the physical clock timestamps associated with each event. The partial ordering between events is displayed as diagonal lines. A developer can interact with the diagram in several ways: searching for events, zooming the diagram in and out, clicking on events to reveal time duration between event pairs or event meta-data, and by transforming the graph by hiding thread timelines or by filtering thread timelines. The tool also includes features to help a developer to pair-wise compare and contrast executions. Due to limited space we only discuss several of these features.

TSViz extends ShiViz [3], a tool for visualizing interactions between nodes in a distributed system. TSViz extends ShiViz in three ways (1) visualization of logical *and physical* time, (2) ability to zoom the diagram based on physical time, (3) ability to search for intervals of activity based on start and end keywords, and an interactive display of interval search distribution results. We briefly discuss each of these differences.

ShiViz was designed for distributed systems in which clocks between nodes are not synchronized. It was limited to the logical time recorded in the log. In a multi-threaded system threads read the same physical clock and TSViz assumes that the logs include both physical and logical timestamps. The physical timestamps are used to position events to scale. This allows the user to judge distances between events, e.g., a cluster of closely positioned events really did occur around the same time, even between different threads. Using physical timestamps alone, however, is not enough for understanding concurrency. They cannot explain whether an event *a* occurred after an event *b* because of a dependency (e.g., *a* caused *b*), or because of scheduling effects which may be transient (e.g., *b* could occur before *a* if the system was re-executed). TSViz therefore visualizes both types of timestamps.

The ShiViz tool does not support zooming: since logical time cannot be used for measurement, it is unclear what zooming semantics to use. TSViz uses physical timestamps, which have a natural notion of scale and zoom: the same number of pixels on the screen can represent a 1μs interval or a 1ms interval in the trace. Further, TSViz collapses thread-local events that are close to each other visually at the current zoom level into clusters. This improves readability and makes the space-time diagram more sparse and abstract as the user zooms out.

Finally, TSViz introduces a new kind of search (while retaining keyword search and structured search in ShiViz). With TSViz a developer can search for intervals that are defined by start events and end events. For example, a developer can find all intervals that start with an *open()* event and end with a *close()* event. Each interval in the result set is associated with a duration and TSViz also generates a histogram of these durations, which are ordered globally, and by thread. A developer can use this histogram to, for example, jump to the longest interval instance.

IV. Case study evaluation

A. *Debugging performance of a new locking algorithm*

A performance engineer was working on a new locking algorithm for a widely used key-value store WiredTiger [2]. At the heart of the algorithm was a well-known ticket lock: each thread wishing to acquire the lock gets a ticket — an integer value signifying the thread's order in the queue of lock waiters. The thread releasing the lock increments the lock's epoch number; a lock waiter whose ticket matches the epoch number becomes the lock holder. Typically lock waiters busy-wait for a lock, repeatedly checking if their turn has come; the new algorithm was designed to limit the number of busy-waiting threads by making some of them block. The thread releasing the lock, in addition to incrementing the epoch number, would also unblock one or more threads whose turn to acquire the lock is approaching.

Once the implementation was complete, the engineer discovered that the lock performed more poorly than expected in some situations. She used TSViz to examine detailed interactions between threads vying for the lock (see Figure 1). First of all, TSViz allowed her to confirm that the logic of the algorithm appeared to be working correctly: threads correctly handed over the lock to one another according to their ticket number, waking up blocked threads as necessary. TSViz also

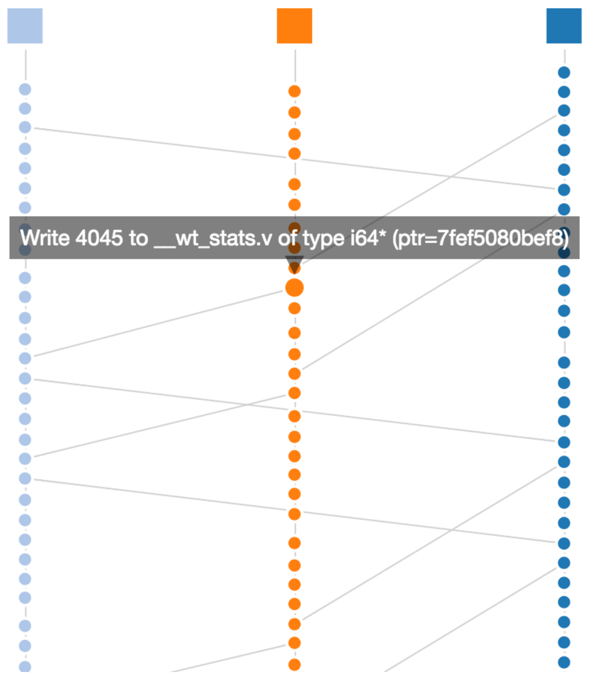


Fig. 2. TSViz showing multiple threads sharing variables. An edge between two events is shown when one thread writes a variable and then another thread reads the same variable. In a read-only lock-free workload we would expect to see few or no edges between threads. The presence of many edges in the TSViz view signals a potential contention on a shared variable.

revealed that sometimes acquiring the lock took much longer than usual. The first suspicion was that the lock holder spent more than the usual amount of time in the critical section. However, TSViz showed that this was not the case. In fact, the lock holder released the lock promptly, as expected. Further examination of the trace with TSViz revealed that the threads who took an unusually long time to acquire the lock were the ones that were blocking, and even though they were woken up by the threads releasing the lock, the wakeup did not occur early enough for them to return from the sleep queue in the OS kernel in time to take the lock as soon as it was released.

Overall, TSViz informed the engineer that the deadline by which a lock waiter must be awakened in order to be ready for its turn to take the lock is a crucial tuning parameter in this algorithm. According to the engineer, there was no way that she would have been able to obtain the same insight from the output of a profiler tool. The profiler might have shown that the program spent a lot of time busy-waiting for a lock, but it would not have provided the insight into the fine-grained intricacies of thread interactions.

As an additional bonus, TSViz led the engineer to discover another unexpected phenomenon. With TSViz the engineer observed long pauses in the execution — functions that did not block on locks or performed I/O took an unusually long time to complete. Further investigation revealed that these pauses were due to a Linux kernel scheduler bug, such as those described in [6].

B. Detecting contentious shared variables

Just like locks, shared variables can become a point of contention in a multi-threaded program and can limit scalability on multicore CPUs. When a thread writes a variable that is cached at other cores, the cache coherency protocol sends messages across the CPU memory hierarchy to invalidate the other cached copies. The more cached copies there are, the longer it takes to read and write the variable, because of these invalidation messages. The cost of accessing shared variables can be high even if they are not protected by locks or accessed via atomic instructions. Unlike locks, shared variables are much more difficult to detect with conventional profilers, because memory accesses are not wrapped in individual functions. We demonstrate how TSViz can be used to efficiently identify shared variables that were a culprit behind a scalability limitation in the WiredTiger key-value store on some benchmarks. Accesses to these shared variables slowed down the execution by 4× on 16 cores and by 20× on 32 cores for a read-only benchmark sequentially traversing the data table (relative to when the problem was fixed).

Originally, it took substantial time and a custom-designed OS kernel module to get to the root cause of this problem: to identify that shared variables were limiting the scaling and to pinpoint the variables that caused the problem. With TSViz (see Figure 2), we can immediately observe significant communication between threads in a workload where this pattern is not expected. In this view, each event is either a function entry/exit or a memory access. An edge is drawn between two events executed by different threads when one thread writes a variable and another thread later reads the same variable. By hovering the mouse over the events connected by the communication edges we can see the shared variable that was accessed. TSVIz reveals that it is the `__wt_stats` statistics structure. As a result of this experience this statistics structure in the WiredTiger product was subsequently redesigned to be per-thread [1], and the scaling bottleneck was eliminated.

V. Discussion and next steps

TSViz is especially suitable for studying micro-interactions among threads. These behaviors are difficult to reconcile with aggregate information provided by profiling tools. We are working on building intermediate tools that are (1) more scalable and visualize behavior at a more macro level, and (2) do not perform aggregation, which makes it impossible to reconstruct the sequence of what actually happened. One way in which we can achieve this kind of abstraction is by detecting patterns among threads and clustering those that behave in a similar manner. Threads in a multi-threaded systems are often clones and have similar performance profiles. By clustering threads together TSViz can visualize larger systems and also cut down on the amount of information that a user needs to initially consider. This strategy has been used to great effect in Ravel [4], [5].

TSViz runs in the browser, which makes it easy to use, but also limits its performance. Runtime traces that are gigabytes in size require substantial processing power; we have observed that TSViz usability degrades substantially after approximately a hundred thousand objects in the browser's DOM (e.g., events in the trace). Smarter and more custom rendering can extend this limit, but the browser ultimately limits the scale at which TSViz is applicable. We are considering alternative approaches, such as visualizing parts of the trace, and browser-based performance improvements, such as web workers and other browser concurrency mechanisms.

The ShiViz tool has the ability to structurally compare pairs of traces and to highlight the differences. We plan to add this capability to TSViz and to take into account the extra physical timestamp information. This information can be used to, for example, highlight events in the trace where the other trace's corresponding event occurred much earlier (i.e., highlight events that were delayed relative to the other trace). This has been previously considered in work by Sambasivan et al. [9].

VI. Conclusion

Although threads are frequently used to improve system performance, threading often introduces correctness and performance issues, which requires the developer to understand a system's runtime behavior. Reasoning about concurrent behavior is difficult, particularly when the behavior is part of a complex multi-threaded system.

We described TSViz, an interactive visualization tool for studying multi-threaded system behavior. TSViz visualizes logical and physical timestamps and includes a variety of advanced capabilities, such as zooming and search. Importantly, the tool runs in a browser, supports a variety of trace formats, and is deployed online: http://bestchai.bitbucket.io/tsviz/.

References

[1] The pull request redesigning the single shared statistics structure to be per-thread. https://github.com/wiredtiger/wiredtiger/pull/2102.

[2] The WiredTiger transactional key-value store. http://www.wiredtiger.com.

[3] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging Distributed Systems. *CACM*, 2016.

[4] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann. Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time. *IEEE TVCG*, 20(12):2349–2358, 2014.

[5] K. E. Isaacs, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P. T. Bremer. Ordering Traces Logically to Identify Lateness in Message Passing Programs. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):829–840, March 2016.

[6] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In *Eurosys*, 2016.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[8] S. Miucin, C. Brady, and A. Fedorova. End-to-end Memory Behavior Profiling with DINAMITE. In *FSE*, 2016.

[9] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *NSDI*, 2011.