ThreadViewer: Visualizing and Comparing Thread Behavior in a Program Execution

CPSC 547 Project Proposal

Augustine Wong augustine.wong@alumni.ubc.ca

Domain, Task, Dataset

Debugging the performance of computer programs is a challenging and time consuming software development task. A recent survey of 308 engineers revealed that 92% of them encountered performance issues in the preceding year, with each issue taking on average 80 hours to fix [1]. The typical method engineers employ to do performance debugging is to use performance tools which record detailed execution traces of program execution. Engineers then analyze these traces to determine the root causes of performance bugs.

My university research group created a suite of performance tools called DINAMITE which can collect execution traces of single host multi-threaded programs [2]. The execution traces created by DINAMITE are in table format, with each item in the tables representing an "event" which occurred in the computer programs. We define an event as being a point in time when a thread either entered or exited a function. Each item therefore contains the following attributes:

Tid – Because DINAMITE can instrument multi-threaded programs, each item has a categorical tid attribute which identifies the thread which entered or exited a function.

Func – The name of the function which the thread identified by the tid attribute either entered or exited. The func attribute is categorical.

Dir – Indicates whether the thread identified by the tid attribute was entering into the function or exiting the function. The dir attribute is categorical.

Time – The time stamp which indicates when the event took place. The time attribute is quantitative. By subtracting the time when a thread entered a function from the time when a thread exited the same function, we obtain the duration of that particular function call instance.

Below is a snapshot of a DINAMITE execution trace which captured the activity of an application thread in Wired Tiger, a key-value store used by MongoDB. Note that items with dir equal to 0 represent the thread entering a function while items with dir equal to 1 represent the thread exiting a function.

tid	func	dir	time
14	__wt_btcur_search	0	409702545846302
14	config_check	0	409702545851774
14	__config_next	0	409702545852101
14	__config_next	1	409702545852335
14	config_check	1	409702545852582
14	__open_session	0	409702545852854
14	__wt_spin_lock	0	409702545853079
14	__wt_spin_lock	1	409702546027625
14	config_check	0	409702546055068
14	__config_next	0	409702546055369
14	__config_next	1	409702546055608
14	config_check	1	409702546055883
14	__wt_session_reset_cursors	0	409702546056232
14	__wt_session_reset_cursors	1	409702546056473
14	__config_getraw	0	409702546056726
14	__config_next	0	409702546056987

Figure 1: Snapshot of a DINAMITE execution trace

The items in the execution trace are ordered by the time attribute. Notice that there are sections in the execution trace where multiple items with dir equal to 0 are grouped together; these groupings indicate that the thread called functions which in turn called other functions. Therefore, from analyzing DINAMITE execution traces, we can discover a thread's "call stack" with respect to time.

Apart from being able to create execution traces, DINAMITE includes tools to analyze and visualize the traces. One such tool called TimeSquared is used to visualize thread callstacks over time. An example of what a TimeSquared visualization looks like is shown in Figure 2 below.

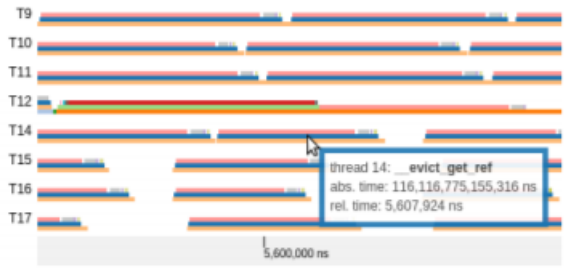

Figure 2: Callstack visualization with TimeSquared

To create the TimeSquared visualizations, we transform our execution traces from quantitatively ordered sequences of function entry/exit events to quantitatively ordered sequences of function calls. Each item representing a function call has the attributes tid, func, entry time, duration, and callstack level. We derive the durations of function calls by subtracting entry times from the corresponding exit times. We also examine the entry and exit times to derive the callstack levels of the function calls.

TimeSquared represents function calls with line marks. The entry time of a function call is represented by the horizontal spatial positioning channel and the duration is represented by the horizontal length channel. The color channel is used to encode the func attribute. Function calls made by the same

thread are grouped into their own rows. The callstack level of each function call is encoded by the vertical spatial positioning within the rows.

The strengths of TimeSquared are that we can easily see:

- The activities of all threads at once
- Dependencies between functions; child functions are shown directly above their parents
- The entry time and duration of each function call

However, TimeSquare's method of visualizing traces does not scale to large, real-world traces. For example, a DINAMITE execution trace capturing over 1 minute of WiredTiger activity contained approximately 200 million events (or approximately 100 million function calls) spread across 28 threads. The same execution trace also captured over 100 thousand different function types. TimeSquared visualizations with millions of function calls would be unfeasibly large; additionally, using the color channel to categorize functions is no longer realistic.

To overcome this scalability issue, we use a "Search, Show Context, Expand on Demand" approach when visualizing traces with Time Squared. Instead of trying to analyze entire traces, we focus on examining portions of the traces which contain function "outliers". We consider outliers to be function calls with excessive duration times.

To find these outliers, we derive 2 new attributes from the quantitatively ordered sequences of function calls we use to create the TimeSquared visualizations:

Average duration – The average duration of all calls to the function defined by the func categorical attribute. For instance, if the func attribute for the dataset item is "foo", then the average duration attribute for the dataset item will equal to the average duration for all foo function calls.

Standard deviation – The standard deviation of all calls to the function defined by the func categorical attribute.

After deriving these attributes, we filter out any dataset items with durations that are not two standard deviations above average; the items which remain represent function outliers. From the entry time and duration attributes, we know the time ranges of each outlier. We also know from the TID attribute which threads executed the outliers. We can therefore located the function outliers in the traces.

But even if we filter execution traces of uninteresting events, two problems still remain. Firstly, function outliers themselves may still be too large to visualize with TimeSquared. One of our WiredTiger DINAMITE traces captured an outlier function lasting 60 seconds; during those 60 seconds, the application thread executed over 11 million function entry/exit events. Secondly, we cannot conveniently use TimeSquared to compare events between two distant, discrete time ranges within the same program execution. For instance, we would have difficulty using TimeSquared to compare a portion of a trace which contains function outliers to another portion of the same trace which contains function calls of typical durations.

Consequently, we propose to make ThreadViewer, a new visualization tool which supports the following tasks:

- Summarize large portions of an execution trace to save on screen real estate. We can still use the filtering technique described above to limit the number of events we need to summarize. But even with filtering, we may still need to summarize millions of events.
- Compare events between two different portions of the same trace so that we can determine, for example, why one function call is an outlier while another call to the same function has a typical duration.
- Discover which parts of an execution trace are worth examining further with TimeSquared. Despite having some deficiencies, TimeSquared is still a good tool for analyzing execution traces. ThreadViewer will help us identify more precisely the parts of an execution trace contain the performance bugs. Then we can use TimeSquared to investigate those performance bugs. By identifying more precisely the parts of the execution trace we are interested in examining, we minimize the number of items TimeSquared needs to display.

To make our infovis problem more tractable, we can reduce the number of attributes we need to visually encode. By getting rid of the time attribute, we transform our execution traces into ordinal sequences of function entries and exits; such sequences would require less screen real estate than Time Squared to display because we no longer need to use the horizontal spatial position and horizontal length channels to represent entry time and duration respectively. Although knowing the durations of each function call is very important for performance debugging, there are several reasons for why it is still valuable to examine ordinal sequences of execution events. Firstly, we may simply want to analyze if a thread is doing anything abnormal like going to sleep frequently. Secondly, since we no longer need to use the horizontal spatial positioning channel and the horizontal length channel to encode function entry times and durations, we can easily align different parts of an execution trace beside each other for comparison. And thirdly, ThreadViewer is intended to work in harmony with TimeSquared which already encodes the time attribute.

Although getting rid of the time attribute reduces the scope of our infovis problem, we still need to summarize our traces to save on screen real estate. One technique we can employ to summarize our traces is to use data compression algorithms. Data compression algorithms excel at finding repetitive patterns within sequences of data items. Fortunately, threads have tendencies to repeatedly execute the same sequences of functions over a substantial periods of time; an example of this repetition is a thread constantly evicting memory pages. Thus, we can employ data compression algorithms to highlight repetitions in program behavior and to collapse execution traces into compressed forms.

SEQUITUR is one example of a data compression algorithm. The SEQUITUR algorithm was developed by Nevill-Manning and Witten to compress sequences of discrete symbols. Below is a demonstration of how SEQUITUR compresses a string of characters:

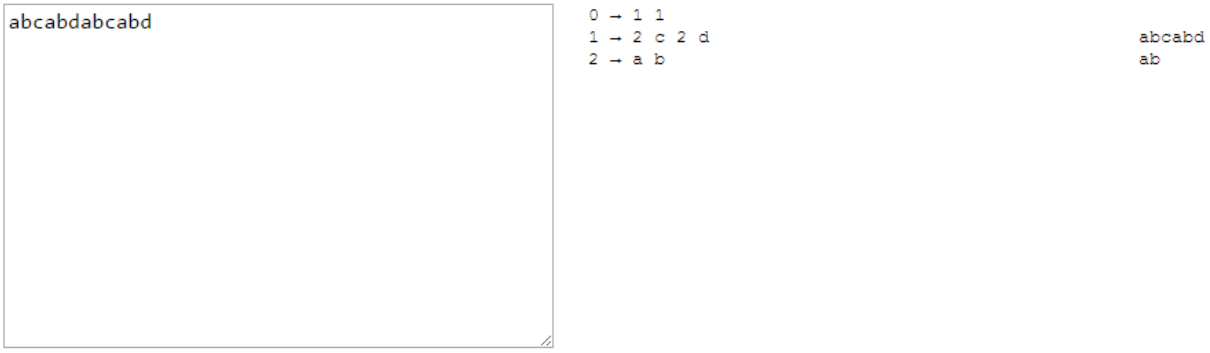

Figure 3: SEQUITUR compressing a string of characters

SEQUITUR produces context-free grammar. In Figure 3, we can see that the context-free grammar generated by SEQUITUR has:

- Terminals {a, b, c, d}
- 3 non-terminals {0, 1, 2}
- Non-terminal 0 as the starting point for the language
- 3 production rules which map the non-terminals to sequences of terminals and non-terminals

We can represent context-free grammar as a parse-tree. Figure 4 below shows the parse-tree of the grammar derived in Figure 3. The nodes representing the terminals are colored yellow.

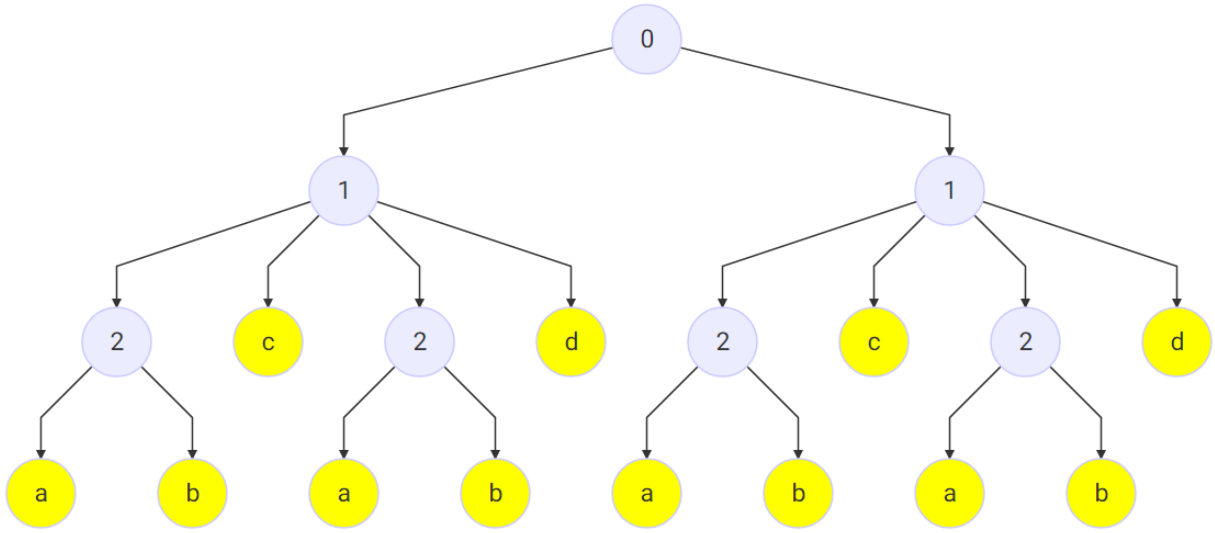

Figure 4: Parse tree representation of context free grammar

Walkinshaw et al posited that an execution trace converted to an ordinal sequence of discrete symbols can be compressed using SEQUITUR [3]. Figure 5 shows how Walkinshaw et al used SEQUITUR to compress a toy execution trace:

Original trace:

```
load, displayDir, selectFile, renderChar, render-
Char, renderChar, renderChar, insertChar, render-
Char, insertChar, renderChar, insertChar, render-
Char, select, copy, paste, renderChar, renderChar,
select, copy, paste, renderChar, renderChar, ren-
derChar, saveFile, displayDir, selectFile, load,
displayDir, selectFile, renderChar, renderChar,
renderChar, renderChar, renderChar, renderChar,
renderChar, renderChar, renderChar
```

Mappings:

```
load=a, displayDir=b, selectFile=c, renderChar=d,
insertChar=e, select=f, copy=g, paste=h, save-
File=i
```

Recoded trace:

```
abcddddededefghddfghddddibcabcddddddddd
```

Production rules
0 → 1 2 2 2 3 3 4 i 5 1 4 4
1 → a 5 4 4
2 → e d
3 → f g h 4
4 → d d
5 → b c

Uncoded production rules
0 → 1 2 2 2 3 3 4 saveFile 5 1 4 4
1 → load 5 4 4
2 → insertChar renderChar
3 → select copy paste 4
4 → renderChar renderChar
5 → displayDir selectFile

Figure 5: SEQUTIUR compressing a toy execution

The “original trace” in Figure 5 is an ordinal sequence of function entry events, presumably executed by a single thread. Each item in this ordinal sequence is mapped to a discrete symbol and passed into SEQUITUR. Thus, the terminals in Figure 5 are the individual function entry events recorded in the trace. Production rule 0 represents the complete trace and the other rules represent “phases” and “subphases” occurring within the trace.

With the SEQUITUR algorithm, we now have a method of summarizing the behavior of each thread captured in our execution traces. However, one drawback to using the SEQUITUR algorithm is that it generates $\log_2 N$ production rules if it encounters N continuous repetitions of a group of symbols. For example, suppose a thread attempted to acquire a lock 64 times. The thread’s execution trace would then consist of 64 repetitions of “Entering lock function, Exiting lock function”:

The SEQUITUR algorithm compressing such a trace will create the following production rules:

0 -> 1, 1
1 -> 2, 2
2 -> 3, 3
3 -> 4, 4
4 -> 5, 5
5 -> 6, 6
6 -> Entering lock function, Exiting lock function

SEQUITUR will produce 6 rules, not including rule 0. Yet the majority of these rules are meaningless for understanding how the thread is behaving; we do not care, for instance, that rule 3 consists of two consecutive instances of rule 4. Ideally, we want to see N continuous repetitions of a group of events represented by one production rule. Therefore, we will need to do some post-processing of the production rules to eliminate meaningless rules.

Expertise

I am a computer hardware engineer and have some basic knowledge about multi-threaded programming. I also acquired some JavaScript and D3.js [4] programming experience from helping my university research group to develop the visual encoding used in TimeSquared.

To develop ThreadViewer, I will use the professors leading my research group as my domain experts. These professors have used TimeSquared to find performance issues in WiredTiger. Therefore, I can consult with them to define the domain tasks and to determine how ThreadViewer can best compliment TimeSquared.

Proposed Infovis Solution

I intend to create two views: the summary view and the comparison view.

- **Summary view.** The purpose of the summary view is to provide a high-level representation of a thread's execution. As mentioned previously, we will use the SEQUITUR algorithm to summarize the execution.
  Suppose we have a thread which executed the following events:

  1) Entered function foo
  2) Exited function foo
  3) Entered function bar
  4) Exited function bar
  5) Attempted to acquire a lock 64 times
  6) Entered function foo
  7) Exited function foo
  8) Entered function bar
  9) Exited function bar

  After removing the meaningless production rules, we will end up with the following summary of the execution:

  0 -> 1, 2, 1

  1 -> Entering function foo, Exiting function foo, Entering function bar, Exiting function bar

  2 -> Entered and exited lock function 64 times

Figure 6 below shows one possibility for how the visualization tool could show this execution:

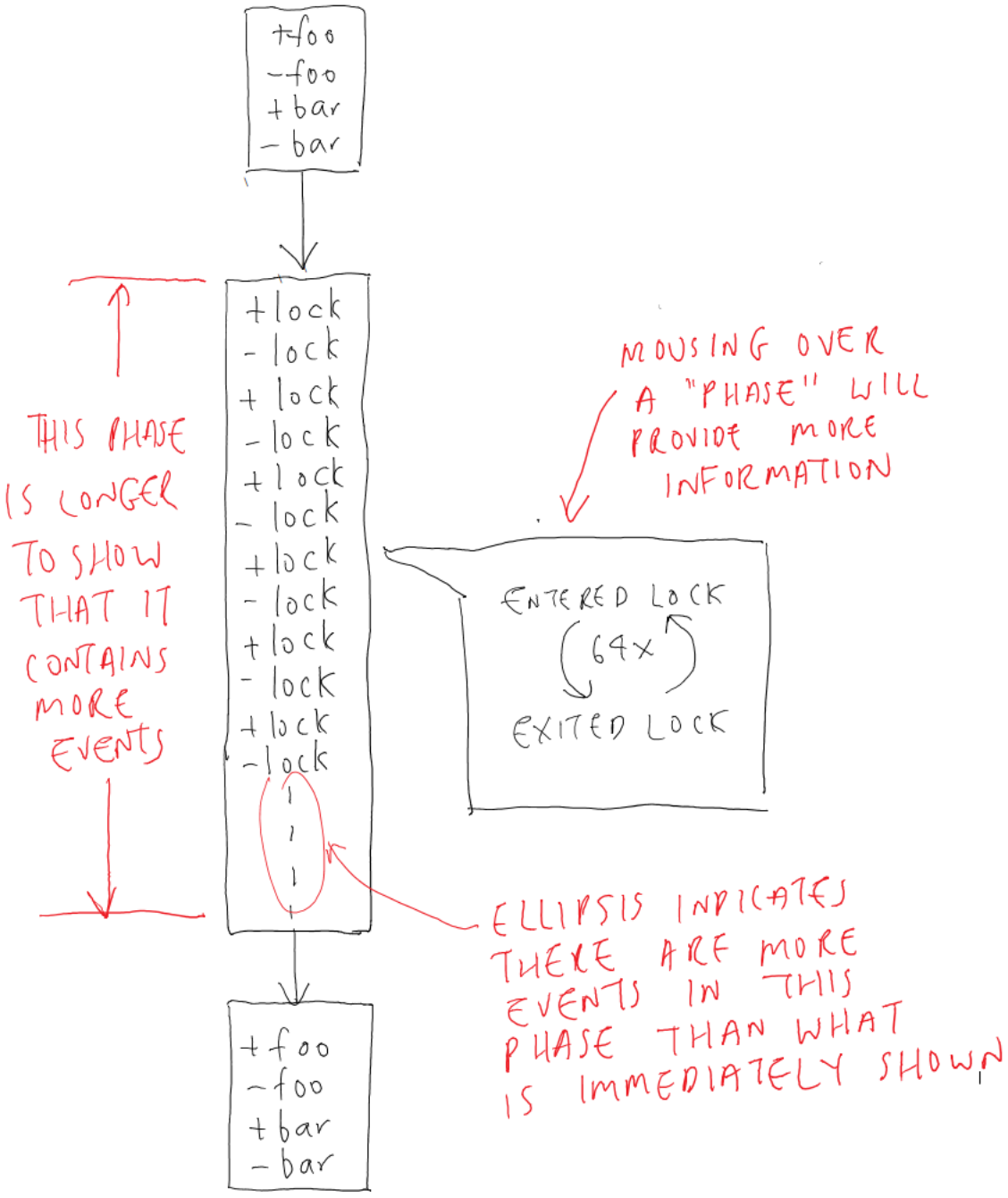

Figure 6: Summary view

Because production rule 0 represents the complete trace, the summary view will essentially be displaying a representation of rule 0. In our example above, production rule 0 consists of three rule instances: 2 instances of rule 1 and 1 instance of rule 2. As suggested by Walkinshaw et al, we will refer to these rule instances as execution “phases”.

Execution phases are represented in the summary view as box marks. Rather than labelling each execution phase by its production rule number, the visualization tool will label each phase

by the function events it contains. To save on horizontal space, each label representing a function entry will begin with a + symbol while each label representing a function exit will begin with a - symbol. By using the function events as labels, users will immediately be able to acquire a general sense of what the thread is doing within each phase.

The vertical spatial positioning channel will be used to encode the order in which the phases are instantiated in production rule 0. The vertical length channel will be used to encode the number of execution events within each phase. In Figure 6, the phase representing the moment when the thread repeatedly attempted to acquire the lock is much longer than the other two phases. However, to save on screen real estate, the vertical length of each phase will not be directly proportional to the number of execution events it contains. Instead, some method of logarithmic scaling will determine the vertical length of each phase.

Occasionally, a phase will not be long enough to label it with all of the execution events it contains. An ellipsis at the bottom of each phase will indicate to users that a phase contains more events than are shown in its labels. If users wish to know all of the function events within a phase, they can mouse over the phase and receive more detailed information.

Kindly note that the visual encoding ideas for the summary view are extremely preliminary. One of the project milestones will be to finalize the visual encoding techniques used by the summary view.

- **Comparison view.** The purpose of the comparison view is to help users compare the differences between two execution traces or even two different parts of the same trace. To compare execution behavior, we would like to adopt the analysis techniques used by scientists when comparing two DNA strands. Like our execution traces, DNA strands are ordinal sequences of items. Therefore, we should be able to compare our execution traces in the same way scientists compare DNA.
  Scientists use the concept of "conservation" to determine how similar multiple DNA strands are. Figure 7 shows how the UCSC Genome Browser [5] uses conservation to compare different animal genes to a human gene.

Figure 7: Conservation between various animal genes and a human gene

The gene of each animal is encoded with a line mark. The thick portions of the line encode the parts of the animal gene which match with the human gene. We can see the gene of a rhesus monkey has high conservation with the human gene. However, the zebrafish which is separated from humans by hundreds of millions of years of human evolution, have very low conservation.

The UCSC Genome Browser uses conservation to provide a coarse grained view of how similar different genes are. If we wish to have a fine grained view of the similarities, we can use the

Needleman-Wunsch algorithm. Scientists use this algorithm to align two DNA strands together, as demonstrated below in Figure 8 below:

```
Sequences     Best Alignments
---------     ----------------------
GCATGCU       GCATG-CU      GCA-TGCU      GCAT-GCU
GATTACA       G-ATTACA      G-ATTACA      G-ATTACA
```

Figure 8: Aligning nucleotide sequences

Scenario of Use

Jim is a computer engineer who wants to improve the performance of WiredTiger. He instruments WiredTiger with DINAMITE and studies the resulting execution traces. He notices that thread 23 took 52 ms to execute the function __cursor_row_search while thread 27 took only 3 ms to execute the same function. He filters out all events from the execution traces which were not executed by thread 23 or 27 and did not take place during the two __cursor_row_search calls. Then he loads the filtered traces into ThreadViewer.

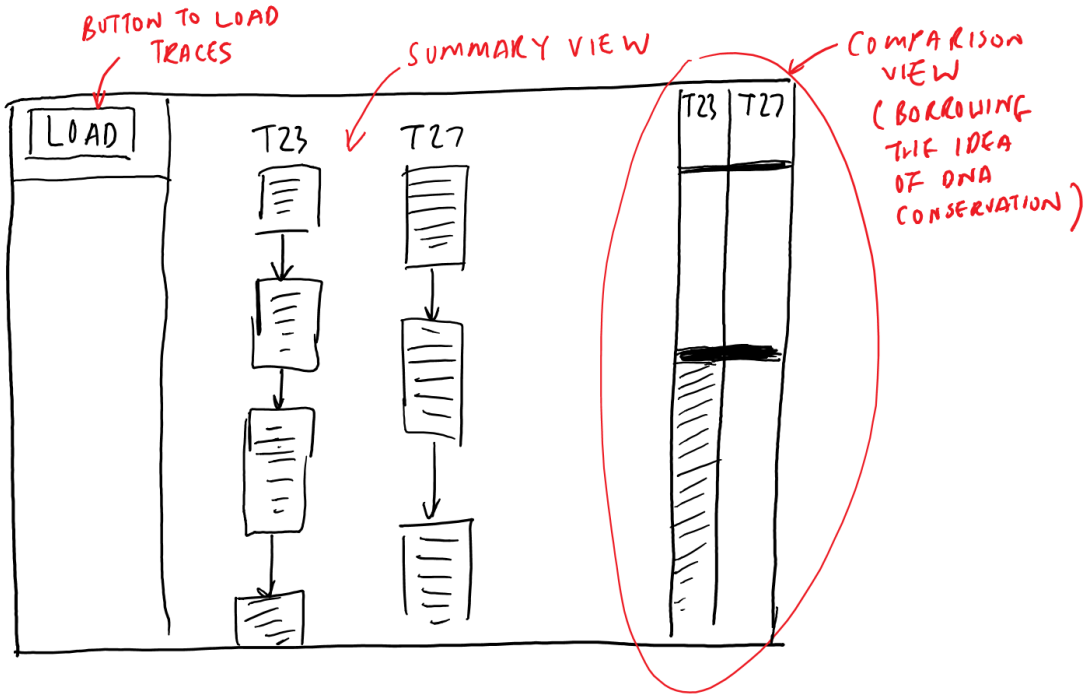

Figure 9: Comparing and viewing thread 23 and thread 27

Jim begins his investigation by using the compare view to compare the behavior between the two threads (Figure 9). He sees that the threads behaved differently when executing the same function. Jim decides to switch to the summary view and examine how thread 23 behaved when performing __cursor_row_search. He sees that thread 23 was building an internal memory page after reading a page from disk. He then uses the summary view to examine how thread 27 behaved when executing

__cursor_row_search. He sees that thread 27 was building a leaf memory page after reading a page from disk, not an internal memory page.

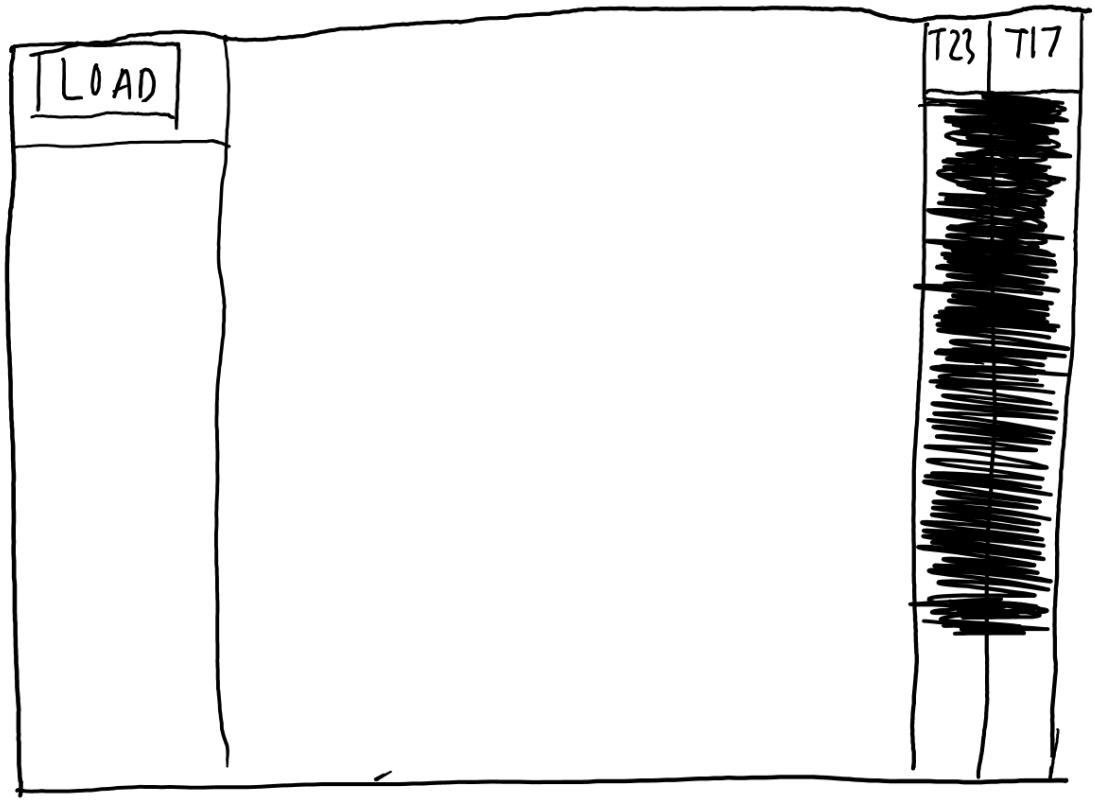

Figure 10: Comparing the behavior of threads 23 and 17

Jim goes through his execution logs once more and notices that thread 17 also took a very long time to execute __cursor_row_search. He decides to use the comparison view to compare the behavior of threads 23 and 17 during their respective outlier calls to __cursor_row_search (Figure 10).
He sees t
hat the two threads behaved almost identically.
Jim now knows that the duration of a call to __cursor_row_search depends on what type of memory page is being built after the page is read from disk.

Proposed Implementation Approach

We created TimeSquared using JavaScript and D3.js. Because we intend for ThreadViewer and TimeSquared to be used together, we will also create ThreadViewer using JavaScript and D3.js. The long term goal beyond the scope of this project will be to merge the ThreadViewer and TimeSquared code together to form one truly cohesive performance debugging tool. ThreadViewer will run on a web browser and be hosted on a bitbucket account.

Project Milestones

Mar 17 – Finalize my visual encoding ideas for the summary view and comparison view.
Mar 21 – First peer project review.

Mar 31 – Interim writeup due. By this time, I plan to be well on my way towards finishing a basic implementation of the summary view.
April 4 – Second peer project review. By this time, I plan to have completed a basic implementation of the summary view.
April 17 – Complete the summary view and a basic implementation of the comparison view. Prepare for the final presentation and begin writing the final paper.
April 25 – Final Presentation. By this time, I plan to have completed the comparison view.
April 28 – Final paper due

Previous Work

Visualization of execution traces is a well-researched field. Most approaches for serial trace visualization involve assigning one of the axes the time variable while the other axis is used to represent different processes, classes, instructions, or methods [6]. TimeSquared also uses one of the axes to represent time and the other axis to represent the behavior of different threads. However, tools which adopt this visualization approach can only visualize fractions of the total program execution duration. Many visualization tools provide an overview of the execution trace and allow users to zoom in on parts of the trace for more details. Extravis is a tool which uses a "massive sequence view" to show an overview of an execution and a "circular view" to show the interactions between the components of a program [7]. Synctrace is another tool which draws a serial timeline overview of a selected thread and shows the call stacks of different threads in the context view [8]. However, this approach of "Overview First, Zoom and Filter, Details on Demand" is not scalable for large traces.
Rather than simply attempting to visualize entire traces, researchers are developing techniques to "localize" root causes of performance issues in computer systems. For instance, Sambasivan et al focused on comparing two request-flow traces to highlight changes in system behavior [9]; by highlight these changes localizes the source of performance issues and guides developer effort.

References

1) A Survey on Performance Tuning By Plumbr, https://plumbr.eu/blog/performance-blog/java-performance-tuning-survey-results-part-i.
2) Svetozar Miucin, Conor Brady and Alexandra Fedorova, **End-to-end Memory Behavior Profiling with DINAMITE,** in 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), 2016.
3) Walkinshaw, Neil, Sheeva Afshan, and Phil McMinn, **Using Compression Algorithms to Support the Comprehension of Program Traces**, in *Proceedings of the Eighth International Workshop on Dynamic Analysis*. ACM, 2010.
4) D3 – Data Driven Documents, https://d3js.org/.
5) UCSC Genome Browser, https://genome.ucsc.edu/
6) Katherine E. Isaacs, Alfredo Giménez, Todd Gamblin, and Abhinav Bhatele, **State of the Art of Performance Visualization**, in EuroVis, 2014
7) Holten, Danny, Bas Cornelissen, and Jarke J. Van Wijk, **Trace Visualization Using Hierarchical Edge Bundles and Massive Sequence Views**, in IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2007.

8) Karran, Benjamin, Jonas Trumper, and Jurgen Dollner, **Synctrace: Visual Thread-interplay Analysis**, in IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2013.
9) Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, Gregory R. Ganger, **Diagnosing Performance Changes by Comparing Request Flows**, in NSDI, 2011.