A Way to Group and Visualize Function Call Instances In a Trace Through Pattern Matching with “Shapes”

Pasindu Muthukuda

Abstract

In this paper, we consider the problem of grouping Function Call Instances in a program execution trace, and using these groups to create a high level, low resolution visualization of the trace. We first define a criteria, called the Ideal Grouping Criteria, that makes precise how two Function Call Instances can be considered similar in a way that agrees with human intuition. Then, we develop the mathematics that leads to an algorithm for grouping Function Call Instances. Finally, we visualize these groupings found in a trace consisting of 100M function entrance/exit events on one screen.

1 Introduction

Performance optimization is something that all commercial software systems must do constantly. But performance optimization is notoriously difficult due to the intricate knowledge that is required of a system in order to both diagnose and fix performance issues. Naturally, many tools have been made in the past to understand where performance issues might lie, and what might be causing them. But it has been found that, for at least systems level software like key-values stores, these performance tools are not enough [2]. The main issue is that although it is possible to observe the execution of a program in great detail [4], it is difficult to present such a massive amount of information to a human in a meaningful way. Profilers get around this by aggregating and sampling metrics, and presenting these metrics. Tools that don’t aggregate and sample data are forced to reduce the time window for which to present the data. But identifying key timewindows is far from an easy task, and in fact corresponds closely with actually diagnosing the performance bug.

What is needed is a tool that facilitates free exploration of the whole trace. One way to do this is to have a high level overview of the trace that a developer can use to identify key timewindows. In this paper, we attempt to solve this problem by identifying a small set of structures that summarize the whole trace (“patterns”), and visualizing all occurrences of these structures on one screen. A quote on quote from [2] indicates that developers would find this useful: “‘I dream of being able to do some automated pattern matching on the data in order to isolate interesting time periods.’”

2 Related Works

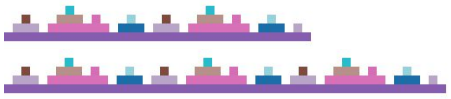
We collect our trace data using an instrumentation tool called DINAMITE [3]. The DINAMITE toolkit consists of both the instrumentation tool, as well as a set of tools used to analyze the trace. TimeSquared [5] is a tool that is used to visualize function calls over time. However, TimeSquared still runs into difficulties trying to display all function calls over large intervals of time; the visualization occupies many pages of space, making it hard for a human to interpret. This is what our algorithm and visualization tool aims to fix.

3 Approach

We address this problem by first defining the notion of a “Function Call Instance” (“Instance” for short) and a “Shape”. A Function Call Instance is simply a function call, along with the sequence of Function Call Instances to happen within. A Shape is a structure reminiscent of a Function Call Instance, but is actually a pattern than many Function Call Instances can match to. But our definition of Shapes is still highly structured, and as a result, we typically end up with a large number of Shapes for a single trace. Fortunately, we are able to define a distances measure over Shapes, allowing us to group Shapes together by similarity into Clusters, leading to a low number of Clusters. These clusters are the “patterns” that we mentioned in the introduction. Finally, we visualize the occurrences of these Clusters on a timeline, summarizing the whole trace on one screen.

4 Qualitative Analysis of Function Call Instances

In order to group Function Call Instances, we first need to identify features that make two Instances look similar. Let’s look at some examples that occur in a run of the WiredTiger evict-multi-stress workload (the trace we use throughout this paper):

a) 

b) 

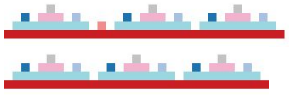
c) 

Figure 4.1

Here, each color represents a function. A function on top of another function is an invocation of the top function from the bottom function.

We align pairs of similar Function Call Instances to illustrate what similar Instances might look like. In Figure 4.1a, the lower Instance repeats calling the (lavender, dark pink, blue) Function Call Instances one more time before exiting. In Figure 4.1b, the lower Instance calls brown a few more times before exiting pink. And Figure 4.1c, there is an inconspicuous call to peach in the top Instance that is not present in the Instance.

To a human, Instances within the pairs look more similar than Instances across pairs. Why is that? Pair a) and b) shows that the number of times something repeats isn’t a factor. When a program executes, it is very common to see repeated sequences of sub Function Call Instances. This is because of loops in the program. Sometimes, the number of iterations of a loop containing a sub Function Call Instance is resolved during runtime, resulting in many similar Instances that only differ by number of repetitions of the sub Instance within. Although distinguishing between these different Function Call

Instances can have some value, to a human, these are basically the same, and thus we choose to consider such repetitions not significant when measuring the difference between two Instances. This exemplifies one important criterion of similarity: our measure of the difference between two Function Call Instances should be resilient to a differing number of exact repetitions of something that occurs in both Instances.

Let's look at what pair c) tells us about similarity. The only difference between the Instances in c) is that there is a call to peach that the top Instance makes that the bottom one doesn't. We want to acknowledge this insignificant, but visible, difference in how we evaluate similarity, because if we don't, we run the risk of a slippery slope where we don't acknowledge differences between Instances that *are* significantly. But in our measure of difference, we want the difference to be proportional to the visible differences between two Instances, meaning that for pair c), the difference should be small. Thus, we have the second important criterion of similarity: our difference measure should reflect the number of visible differences found between two Function Call Instance proportionally.

We summaries our criteria below:

1. Our difference measure should be resilient to a differing number of repetitions of something.
2. Our difference measure should grow proportionally to the number of differences between a Function Call Instance and a perturbed version of it.

We call this the **Ideal Grouping Criteria**; a criteria that is necessary and sufficient for a definition of a difference measure to agree with human intuition. We can now embark on rigorously defining a difference measure that satisfies the criteria above.

5 The Mathematics of Shapes

For a given trace, let F be the set of functions that appear in that trace. We can precisely define the set of all Function Call Instances as follows:

Definition 5.1: Function Call Instances
Let F be a set, and define $I_0 = \{\}$, $I_{i+1} = \{[f, K] : f \in F, K \text{ is an ordered set of } I_i\}$. By induction, we observe $I_i \subset I_{i+1}$. We define the set of all Function Call Instances to be $I = \cup_{i \in \{0 \ldots \infty\}} I_i$. If $k \in I$, then k has the form $[f, K]$, and f is called the "base function", K is called the "child Function Call Instances". The "depth" of k is the smallest i such that $k \in I_i$. We can say an Instance k is "smaller" than an Instance l if it's depth is smaller.

A class for a Function Call Instance might look like the following:

Listing 5.1:

```
class Instance:
    int functionId;
    List<Instance> children;

    Instance(functionId, children):
        this.functionId = functionId;
```

```
        this.children = children;

    int calculateDepth():
        if (children.length is 0):
            return 1;
        else:
            return 1 + (max depth of children);
```

Thus, a Function Call Instance is just a tree where nodes hold a *functionId*. The depth is just the depth of the tree. Remember that an execution trace can be converted into a Function Call Instance; this was the purpose of the definition.

The criteria in the last section motivates us to define the notion of a Shape. A Shape is a structure that partitions the space of Function Call Instances with the goal of having similar Function Call Instances falling within the same partition, and distinct ones falling in different partitions. Our definition of a Shape is as follows:

Definition 5.2: Shapes
Let F be a set, and define $S_0 = \{null\}$, and $S_{i+1} = \{null\} \cup \{[f, C] : f \in F, C \subset S_i \text{ where } null \in C\}$. By induction, we observe $S_i \subset S_{i+1}$. We define the set of all Shapes to be $S = \cup_{i \in \{0 \ldots \infty\}} S_i$. If $s \in S$, then s has the form $[f, C]$, and f is called the "base function", C is called the "child Shapes". We also refer to C as the "contents" of the Shape. The "depth" of a shape s is the smallest i such that $s \in S_i$. We can say a Shape s is "smaller" than a Shape t if its depth is smaller.

The *null* element is a special element, and its use will become evident later. For now, just think of it as a means of preventing C from ever being empty.

A class for a Shape might look like the following:

Listing 5.2:

```
class Shape:
    int functionId;
    Set<Shape> children;

    Shape(functionId, children):
        assert(children.contains(null));
        this.functionId = functionId;
        this.children = children;

    int calculateDepth():
        return 1 + (max depth of children, where the depth of null is 0);
```

Thus, a Shape is just a tree where nodes hold a *functionId*, children are unique (deeply), and each node has *null* as a child. The depth is just the depth of the tree.

We can now map Function Call Instances to Shapes, and define two Function Call Instances to be in the same partitions if they both map to the same Shape. We define this mapping φ inductively as follows.

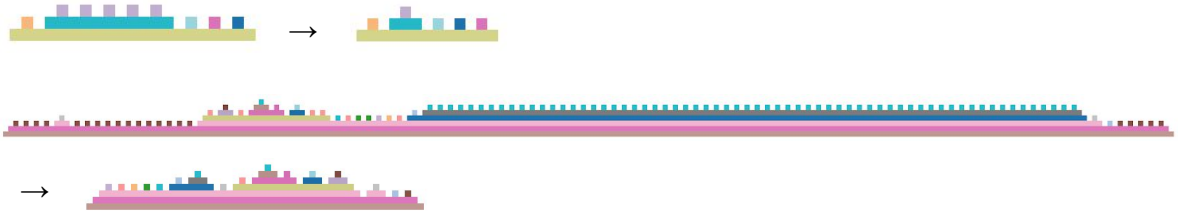
Definition 5.3: Mapping Function Call Instances into Shapes
Define φ over I_0 to be the vacuous mapping (since I_0 is empty). Suppose φ is defined for I_n. We extend φ to I_{n+1} by mapping $[f, K] \in I_{n+1} - I_n$ to $[f, \{null\} \cup \{\varphi(\beta) : \beta \in K\}]$. This defines φ inductively for all I. We say that $\alpha \in I$ "reduces" or "condenses" to $s \in S$ if $\varphi(\alpha) = s$, and we call s the Shape of α.

This mapping can be implemented in code as follows:

Listing 5.3:

```
function mapInstanceToShape(instance):
    Set<Shape> shapeChildren = [null]; // since children must always contain null
    for (childInstance in instance.children):
        shapeChildren.add(mapInstanceToShape(childInstance));
    return new Shape(instance.functionId, shapeChildren);
```

Essentially, φ is kind of a reduction of a Function Call Instance α to a Shape s, where we reduce the child Function Call Instances of α via φ, and condense these reduced children by taking their set. Taking their set removes information about repetition and order. Thus, a Shape is indifferent to ordering and repetition of child function calls when partitioning the space of Function Call Instances. We show examples of this type of reduction below:



If we were to define similarity of Function Call Instances based on if they mapped to the same Shape, this would evidently satisfies criterion 1 of the Ideal Grouping Criteria, but not criterion 2. To satisfy criterion 2, we need a numerical difference measure between Function Call Instances. The way we approach this is by first defining a difference measure d among Shapes. Then, by considering the difference between two Function Call Instances to be that of their corresponding Shapes, we show that this definition satisfies criterion 2.

The difference between two Shapes can be thought of as a distance. We can imagine Shapes being embedded into some high dimensional manifold, where the difference measure is some sort of distance between Shapes. In mathematics, the notion of distance is made precise by the concept of metric spaces. A metric space is a set S with a function $d : S \times S \to \mathbb{R}$ such that:

1. $d(x, y) \geq 0$, with $d(x, y) = 0$ if and only if $x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, z) \leq d(x, y) + d(y, z)$

The most common example of a metric space is $\mathbb{R}^n$, where $d(x, y) = \| x - y \|$, where $\| x - y \|$ is the Euclidean distance. The properties of a metric seem sensible to have for our difference measure between Shapes, since it reflects our intuition of what difference is. Thus, we focus on defining a reasonable metric among the set of Shapes.

We begin defining our metric by first defining a metric over functions. For the set of functions F, the simplest metric is to define $r(f, g) = 1$ for $f, g \in F$ where $f \neq g$, and $r(f, f) = 0$ for $f \in F$. The symmetry in this definition is warranted because in our analysis, all functions are treated equally.

Next, we note that the contents of a Shape s is a set of Shapes. These child Shapes are smaller; they have lower depth. Thus, we might be tempted to define the Shape metric inductively on depth. To do this, we need a metric over sets of smaller Shapes. For this, we introduce the Hausdorff metric, which is a general metric over non-empty subsets of an existing metric space:

Definition 5.4: Hausdorff Distance
Let S be a finite metric space with metric d, and consider the set M of non-empty subsets of S. Let A and B be elements in M, and define the r: $M \times M \rightarrow \mathbb{R}$ to be $d^h(A, B) = \max\{\max_{\alpha \in A}\{\min_{\beta \in B}\{d(\alpha, \beta)\}\}, \max_{\beta \in B}\{\min_{\alpha \in A}\{d(\alpha, \beta)\}\}\}$. d^h is called the Hausdorff distance over M, and it is a metric, turning M into a metric space. We also define the half-Hausdorff distance, denoted d^{hh} from A to B as $\max_{\alpha \in A}\{\min_{\beta \in B}\{d(\alpha, \beta)\}\}$. Thus, $d^h(A, B) = \max\{d^{hh}(A, B), d^{hh}(B, A)\}$. Furthermore, define $\gamma_B(\alpha) = \{\beta \in B : d(\alpha, \beta) = \min_{\beta \in B}\{d(\alpha, \beta)\}\}$. That is, $\gamma_B(\alpha)$ is the set of all element in B that is closest to α (there can be multiple such elements).

A proof that the Hausdorff distance is indeed a metric can be found here [1]. Importantly, the sets in M must be **non-empty** subsets of S. Intuitively, the Hausdorff distance between two sets is the largest distance away that an element is from the whole of the other set. Consider what it means to say the Hausdorff distance between $A \subset S$ and $B \subset S$ is 1. This mean for all elements in $A \cup B$, there is an element in the opposite set closer than or equal to 1 away. The Hausdorff distance is an upper bound for how far an element can be from the other set.

There are a couple more elementary theorems that we should know about metric spaces before defining the Shape metric:

Theorem 5.1
Let M and N be metric spaces with metrics d and r.

a) A subset $N \subset M$ is also a metric space with metric d.
b) The Cartesian Product $M \times N$ is a metric space with the function $q((m_1, n_1), (m_2, n_2)) = d(m_1, m_2) + r(n_1, n_2)$ being a metric.

Armed with the function metric space, denoted r, together with the Hausdorff distance, we can now define our Shape metric. Suppose we have Shapes $[f, C]$ and $[g, D]$, in S_{n+1}. If we have a metric d defined for S_n (which contains C and D), we can define $d([f, C], [g, D]) = r(f, g) + d^h(C, D)$, where d^h is the Hausdorff distance over subsets of Shapes. Here, we see the need for the *null* element in the definition of a Shape. The *null* element guarantees that C and D are non-empty, which is necessary for the Hausdorff distance to exist.

This approach raises another problem: in order to define *d* for S_{n+1} inductively, we must also define *d* between *null* and non-*null* shapes in S_{n+1}. For this, we add a fictitious function *-1* to *F*, and treat it as the base function for *null*. Whenever we try taking the distance between *null* and a non-*null* shape, we replace *null* with [*-1*, {*null*}], effectively giving *null* a base function, and take the distance between the Shapes as we normally would. This results in an attempt to take the distance between *null* and a smaller Shape. Eventually, we will be taking the distance between *null* and *null*, which we define as 0.

How do we add *-1* to *F*? We would like $\{-1\} \cup F$ to still be a metric. One way of doing this is to define $r(-1, -1) = 0$, and $r(-1, f) = ½$ for $f \in F$. We can easily verify $\{-1\} \cup F$ is a metric, denote this set as F_{-1}, and denote this extended metric as still *r*. We define $r(-1, f)$ as ½ instead of 1 because we don't want to consider a missing Shape to be as different as a completely different Shape.

Below is an algorithm implementing this process:

Listing 5.4:

```
function distance(shape1, shape2):
    if (shape1 is null && shape2 is null):
        return 0;

    if (shape1 is null): shape1 = new Shape(-1, {null});
    if (shape2 is null): shape2 = new Shape(-1, {null});

    baseFunction1 = shape1.functionId;
    baseFunction2 = shape2.functionId;
    children1 = shape1.children;
    children2 = shape2.children;

    return functionDistance(baseFunction1, baseFunction2)
         + hausdorffDistance(children1, children2);

function functionDistance(baseFunction1, baseFunction2):
    if (baseFunction1 is baseFunction2):
        // this includes the case when both are -1
        return 0;
    else if (baseFunction1 is -1 || baseFunction2 is -1):
        return 0.5;
    else:
        return 1;

function hausdorffDistance(children1, children2):
    const furthestTo2From1 =
        max_{child1 ∈ children1}{min_{child2 ∈ children2} {distance(child1, child2)}}};
    const furthestTo1From2 =
        max_{child2 ∈ children2}{min_{child1 ∈ children1} {distance(child2, child1)}}};

    return max(furthestTo2From1, furthestTo1From2);
```

We can formalize the complex process above, and prove that it forms a sound metric.

Definition 5.5
Consider the sets S_i from Definition 5.2. Recall $S_0 = \{null\}$, and define $d_0 : S_0 \times S_0 \to \mathbb{R}$ by setting $d_0(null, null) = 0$. We want to define functions $d_i : S_i \times S_i \to \mathbb{R}$ for $i \in \{0 \ldots n\}$ such that:

1. d_i is a metric over S_i ,
2. $d_i([f, C], null) = r(f, -1) + d_{i-1}^{\,h}(C, \{null\})$, and
3. $d_i([f, C], [g, D]) = r(f, g) + d_{i-1}^{\,h}(C, D)$.

First, we see d_0 trivially has these 3 properties. Define M_n the set of all subsets of S_n that contains *null*, and consider it as a metric space under the Hausdorff metric $d_n^{\,h}$. Consider the product metric space $G = F_{-1} \times M_n$. Consider the sub-metric space of G with all elements $[-1, C]$ removed except for $[-1, \{null\}]$. Replacing $[-1, \{null\}]$ with just *null*, we see that G becomes S_{n+1}. Denote G's metric as d_{n+1}. From its definition, we see d_{n+1} satisfies properties 1 to 3. Thus, we can inductively define d_i for all $i \in \{0 \ldots \infty\}$.

In the above definition, we defined a metric d_i for S_i for all i. We now show that d_{i+1} agrees with d_i for all i.

Theorem 5.2
For all d_i , $i \in \{1 \ldots \infty\}$, in Definition 5.5, we have $d_{i-1}(s, t) = d_i(s, t)$ for $s, t \in S_{i-1}$.

Proof:
We use induction on i. The theorem is trivially true for $i = 1$, since $S_{i-1} = S_0 = \{null\}$, and $d_0(null, null) = 0 = d_1(null, null)$ (since d_0 and d_1 are metrics by Definition 5.5, property 1). Suppose the theorem is true for $i = n$. Then we have $d_{n-1}(s, t) = d_n(s, t)$ for $s, t \in S_{n-1}$, and thus by the Definition 5.5:

1. d_n is a metric
2. $d_n([f, C], null) = r(f, -1) + d_n^{\,h}(C, \{null\})$, and
3. $d_n([f, C], [g, D]) = r(f, g) + d_n^{\,h}(C, D)$.

For all elements in S_n. We just replaced $d_{n-1}^{\,h}$ on the right hand side with $d_n^{\,h}$. Then, from Definition 5.5, with $[f, C], [g, D] \in S_n$, we have:

1. d_{n+1} is a metric.
2. $d_{n+1}([f, C], null) = r(f, -1) + d_n^{\,h}(C, \{null\}) = d_n([f, C], null)$
3. $d_{n+1}([f, C], [g, D]) = r(f, g) + d_n^{\,h}(C, D) = d_n([f, C], [g, D])$

This shows that the theorem is true $i = n + 1$. □

Finally, we can now define the metric over Shapes.

Theorem 5.3
For $s, t \in S$, define $d(s, t) = d_n(s, t)$ where n is any $n \geq \max\{\text{depth}(s), \text{depth}(t)\}$. This definition of $d(s, t)$ is well defined, and thus defines a metric over S.

The proof follows directly from Theorem 5.2.

6 Satisfaction of the Ideal Grouping Criteria

The motivation for defining Shapes and the Shape metric was to satisfy the criterion 2: the difference measure between two Function Call Instances should grow proportionally to the number of differences. We can define the difference measure for Function Call Instances as follows:

Definition 6.1
Let k and l be Function Call Instances, and define their difference to be $d(S(k), S(l))$, where $S(k)$ and $S(l)$ are the Shapes of k and l. We denote this difference measure between Function Call Instances as δ.

Note that this isn't a metric over I, since k and l can be distinct, but still map to the same Shape, resulting in a difference of zero.

The difference measure δ can be implemented in code as follows:

Listing 6.1:

```
function difference(instance1, instance2):
    Shape shape1 = mapInstanceToShape(instance1);
    Shape shape2 = mapInstanceToShape(instance2);
    return distance(shape1, shape2);
```

We have the following important theorem, which makes δ satisfy criterion 2:

Theorem 6.1
Let k be a Function Call Instance. Let k' be the same as k, except with one of its descendent Function Call Instances exchanged with another that has a difference of p from the original. Then the difference between k' and k is at most p.

Proof:
We use induction on the depth of k. This theorem is vacuously true for depth(k) = 1, since k has no descendents (and thus there is no pair k and k' as required by the theorem). Suppose it is true for depth n, and consider a k and k' with depth(k) = n + 1. Let b be the child of k containing the descendent to be exchanged, and let b' be the child of k' that b changes to. Call the shapes that b and b' reduce to c and c'. We have $d(c, c') \leq p$ by the induction hypothesis. Let $s = [f, C]$ be the Shape of k, and $s' = [f, C']$ be the Shape of k' Then s' compares to s in one of 3 ways, and we prove the theorem with a case analysis:

1. $C' \subset C$, c disappears. This happens when b is the only child of k that reduces to c. Since $d(c, c') \leq p$, we have the half Hausdorff distance $d^{hh}(C', C) = 0$, and $d^{hh}(C, C') \leq p$, and so $\delta(k, k') \leq p$.
2. $C \subset C'$, a new child c' appears. This happens when there are other children of k besides b that reduce to c, and there isn't already a child of k that reduces to c'. Since $d(c, c') \leq p$, we have the half Hausdorff distance $d^{hh}(C, C') = 0$, and $d^{hh}(C', C) \leq p$, and so $\delta(k, k') \leq p$.
3. c disappears and a new child c' appears. This happens when b is the only child of k that reduces to c, and there isn't already a child of k that reduces to c'. Since $d(c, c') \leq p$, we have the half Hausdorff distance $d^{hh}(C', C) \leq p$, and $d^{hh}(C, C') \leq p$, and so $\delta(k, k') \leq p$. □

With this Theorem, we now have a difference measure δ in the set of Function Call Instances I that have the following two properties:

1. For $k \in I$, if k' is the same as k except some descendent Function Call Instance of k is repeated a different number of times, then $\delta(k, k') = 0$.
2. For $k \in I$, if k' is the same as k except some descendent Function Call Instance of k is replaced by another with a difference of p, then $\delta(k, k') \leq p$.

Thus, we have a difference measure between Function Call Instances with the Ideal Grouping Criteria, as we set out to find.

7 An Alternative Interpretation of Shapes

Although we have satisfied our goal of defining δ, it is still difficult to interpret what the visual differences actually are between two Function Call Instances with a numerical difference of p. This is mainly because d, the metric over Shapes, is hard to visualize. Fortunately, d has another interpretation that we haven't explored yet, one that is easier to visualize. This section can be skipped without loss of continuity, but is here for the interested reader.

First, we make some definitions.

Definition 7.1
We say that two Function Call Instance k and k' "match" if they have the same number of children, and, denoting the child lists as $(b_1 \dots b_n)$ and $(b_1' \dots b_n')$, b_i and b_i' also match.

If we imagine two matching Function Call Instances as in Figure 4.1, we see their outlines would coincide, even though the functions within are different. This is why we call them "matching".

Definition 7.2
Let k and k' be matching Function Call Instances. Writing $k = [f, (b_1 \dots b_n)]$ and $k' = [f', (b_1' \dots b_n')]$, we define the "stack difference" between k and k' as $sd(k, k') = r(f, f') + \max\{\{0\} \cup \{sd(b_i, b_i') \mid i \in \{1 \dots n\}\}\}$, where r is the metric over functions.

Definition 7.3
Consider I to be the set of all Function Call Instances with the set of functions being $F_{-1} = \{-1\} \cup F$. Let s and s' be two Shapes (in the set S of Shapes with functions F). We define a "matching Instance pair of (s, s')" to be a pair of Function Call Instances (k, k') such that:

1. If s and s' are both *null*, $(k, k') = ([-1, ()], [-1, ()])$.
2. If s and s' are not both *null*, replacing *null* with $[-1, \{null\}]$ if one of them is *null,* writing $s = [f, C]$ and $s = [f', C']$, defining 2 sequences with elements in C and C' as $c_{seq} = (c_1 \dots c_n)$ and $c_{seq}' = (c_1' \dots c_n')$ where each Shape in C appears in c_{seq} and each Shape in C' appears in c_{seq}', finding matching Instance pairs (b_i, b_i') of (c_i, c_i'), we can write $k = [f, (b_1 \dots b_n)]$ and $k' = [f', (b_1' \dots b_n')]$.

Of course, a matching Instance pair of (s, s') also matches in the sense of Definition 7.1.We call finding a matching Instance pair for s and s' as doing a "matching Instance pair expansion". Also note

that this definition defines a procedure for calculating matching Instance pairs recursively for arbitrary (s, s'), provided we have a rule for lining up the children of 2 non-*null* Shapes into sequences $c_{seq} = (c_1 \ldots c_n)$ and $c_{seq}' = (c_1' \ldots c_n')$ where each Shape in C appears in c_{seq}, and each Shape in C' appears in c_{seq}'.

Finding a matching Instance pair (k, k') for Shapes s and s' is almost like an inverse operation of reduction. Indeed, it is a right inverse of reduction if we extend the definition of reduction as follows:

Definition 7.4
Consider Function Call Instances with functions in F_{-1}. We can extend the existing definition of reduction so that any Function Call Instance with base function -1 reduces immediately to *null*. Reduction here still results in a Shape that only uses functions in F. All Function Call Instances with functions in F still reduce to S in the same way as the old definition of reduction.

Thus, if (k, k') are matching Instance pairs of s and s', then k reduces to s, and k' reduces to s'.

Now, we show how we can form matching Instance pairs of s and s' such that the stack difference between them is equal to $d(s, s')$.

Theorem 7.1
Recall γ from Definition 5.4. Letting $s = [f, C]$ and $s = [f', C']$, consider the recursive procedure for constructing matching pairs outlined in Definition 7.3

1. If we choose $c_{seq} = (c_1 \ldots c_n)$ and $c_{seq}' = (c_1' \ldots c_n')$ such that either $c_i \in \gamma_C(c_i')$ or $c_i' \in \gamma_{C'}(c_i)$, and where for all $c \in C$, there is an i such that $c = c_i$ and $c_i' \in \gamma_{C'}(c_i)$, and for all $c' \in C'$, there is an i such that $c' = c_i'$ and $c_i \in \gamma_C(c_i')$, then any matching pair generated this way has a stack difference of $d(s, s')$. We call such a matching pair a "Hausdorff matching Instance pair".
2. For any pair of Shapes, a Hausdorff matching Instance pair exists.

Proof:
We prove both of these statements by induction. For s and s' in S_0 (both are *null*), the above procedure generates the pair $k = [-1, ()]$ and $k' = [-1, ()]$ whose stack difference is 0, proving the base case. Suppose the theorem is true for S_j. Consider $s = [f, C]$ and $s' = [f', C']$ in S_{j+1}, taking depth(s) = $j + 1$ without loss of generality, and taking *null* as $[-1, \{null\}]$ if s' is *null*, the stack difference for a Hausdorff matching Instance pair $k = [f, (b_1 \ldots b_n)]$ and $k' = [f', (b_1' \ldots b_n')]$ is $sd(k, k') = r(f, f') + \max\{_{i \in \{1 \ldots n\}}\{sd(b_i, b_i')\}\}$ by Definition 7.2. But by the induction hypothesis, $sd(b_i, b_i')$ is just $d(c_i, c_i')$, and since all $c \in C$ shows up in c_{seq} and all $c' \in C'$ show up in c_{seq}', the last term is just the Hausdorff distance of C and C'. Thus $sd(k, k') = d(s, s')$.

To show existence, we can create a Hausdorff matching Instance pair as follows: when constructing c_{seq} and c_{seq}', first place all $c \in C$ into the left side of c_{seq} with corresponding child c' being an element of $\gamma_{C'}(c)$. Then place all $c' \in C'$ into the right side of c_{seq}' with corresponding child c being in $\gamma_C(c')$. This satisfies the definition of a Hausdorff matching Instance pair. □

The above Theorem demystifies the Shape metric a little bit. The distance between Shapes can be interpreted as the stack difference between a Hausdorff matching Instance pairs. The Function Call Instances making up the matching pair reduce (under Definition 7.4) back to the original Shapes.

Finally, we can prove that a Hausdorff matching Instance pair has the minimum stack difference for any matching pair, completing the relationship between the Shape metric and the stack difference.

Theorem 7.2
Given Shapes *s* and *s'*, the lowest stack difference for any matching Instance pair is achieved by a Hausdorff matching Instance pair.

Proof:
We again prove by induction. For *s* and *s'* in S_0 (both are *null*), the only matching Instance pair is k = [*-1*, ()] and k' = [*-1*, ()], and thus the theorem is true for this case. Now suppose the theorem is true for S_j. Consider s = [*f*, *C*] and s' = [*f'*, *C'*] in S_{j+1}, taking depth(s) = j + 1 without loss of generality, and taking *null* as [*-1*, {*null*}] if *s'* is *null*. To minimize the stack difference for a matching pair k = [*f*, (b_1 … b_n)] and k' = [*f'*, (b_1' … b_n')], for each c_i, we want to pair it with c_i' such that b_i and b_i' have a minimum stack difference. But by the inductive hypothesis, we could just set c_i' to be an element in $\gamma_{C'}(c_i')$. This is just the requirement that k and k' are a Hausdorff matching Instance pairs. Since we know a Hausdorff matching Instance pair always exists, we are done. □

8 Analysis of Shapes in a Real Execution Trace

Now that we have a firm mathematical theory of Shapes, let's turn our attention to applying it to a real trace. We experiment on a run of the WiredTiger evict-multi-stress workload. In particular, we look at threads 1, 7, 9, 10, and 11. Threads above 9 are all worker threads, and thus have very similar behavior, so we only choose threads 9, 10, and 11 for analysis. Below thread 9, the only threads that have significant activity are 1 and 7, so we include those in our analysis. This run of evict-multi-stress ran for 80 seconds, and produced 46M, 20M, 11M, 11M, and 11M function entrance and exit events for threads 1, 7, 9, 10, and 11 respectively. There are 274 different functions that were called, not including mutex functions. From now on, when we refer to the trace, we are referring to just these 5 threads.

Below, we have a table describing the number of Shapes present in the trace. The # All Shapes is all of the shapes found in the current thread and the threads prior. This number is different from just summing up # Shapes in Thread, because Shapes can be shared across threads. Sharing of Shapes is clearly evidenced in Threads 9 to 11, which share almost all of their Shapes. In this table, we are discounting the *null* Shape, and Shapes of the form [*f*, {*null*}].

Table 1:

Thread	# Shapes in Thread	# Additional Shapes	# All Shapes
1	513	513	513
7	745	694	1207

9	101	69	1276
10	104	4	1280
11	104	4	1284

Below is a diverse sample of Shapes from across the threads. In our Shape visualizations, we don't show the *null* element, making them look very similar to the visualizations of the Function Call Instances found in Figure 4.1.

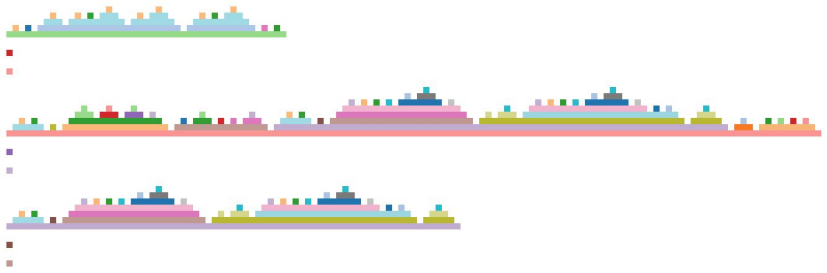
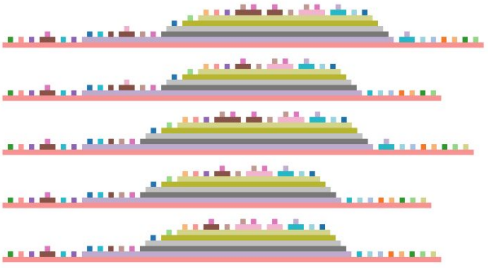


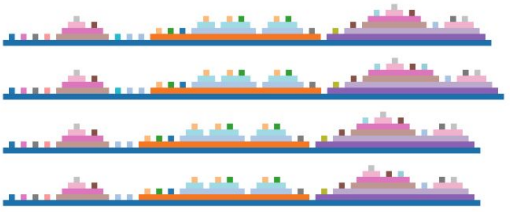
Figure 2.

Although Figure 2 seems to indicates that all Shapes look distinct from one another, we observed that this is not the case. Typically, most Shapes have at least one other Shape that looks very similar. And some Shapes even have many Shapes, on the order of dozens, that look very similar to it. Figure 3 shows just some of these groups of Shapes.

a)



b)



c)

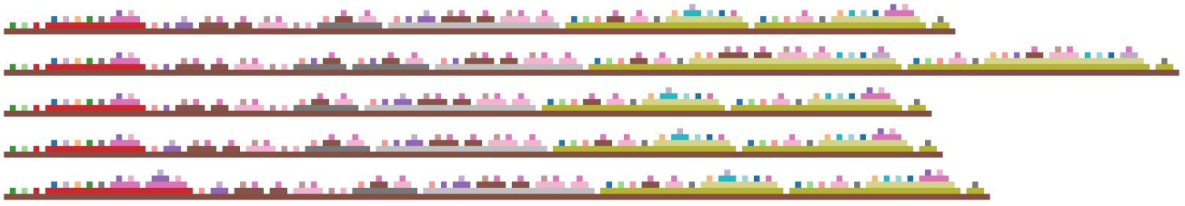


Figure 3.

In fact, we find that the distance between these similar shapes is mostly less than 1.5. If we were to group Shapes according to their similarity, how many such groups would we end up with? We create a clustering algorithm that clusters Shapes together based on their distance to do exactly this.

8.1 Clustering

Because *d* is a metric over Shapes, we can use any general metric space clustering algorithm to cluster Shapes. This is one of the advantages of having strong mathematical properties in the objects we study. But for our purposes, we observed that we can just use a very simple clustering algorithm and end up with a fairly satisfactory set of clusters. That is, all elements in a cluster look very similar, and elements across clusters generally look very different.

We define a cluster to be a set of Shapes that 1) all share the same base function, and 2) has $d(s, s') \leq$ *cluster_threshold*, where *s* and *s'* are Shapes and *cluster_threshold* is some predefined constant. In our algorithm, we set *cluster_threshold* to be 1.5. Together, we call these two properties the "cluster property". Our clustering algorithm simply iterates through all of the Shapes, creating the clusters as we go. For the current Shape *s*, if adding it to an existing cluster doesn't break the cluster property, then we add *s* to it. If no such cluster exists, we create a new cluster consisting of only *s*.

Unfortunately, this simple algorithm lacks some desirable properties, like invariance to the order in which the Shapes are iterated over. However, the practical results of this algorithm justifies its use.

After running the clustering algorithm, the number of clusters we find is far less than the number of Shapes reported in Table 1. That is, although the number of *Shapes* in the trace is high, the number of clusters found with our clustering algorithm is low. Table 2 shows the count of clusters per thread (# Clusters in Thread), and clusters across threads (# All Clusters). We are discounting the clusters that contain Shapes of the form [*f*, {*null*}] in order to ignore cluster containing trivial Shapes.

Table 2:

Thread	# Clusters in Thread	# Additional Clusters	# All Clusters
1	63	63	63
7	18	13	76
9	18	13	89
10	18	0	89
11	14	0	89

8.2 Timeline Visualization

We now have a compelling way to group Function Call Instances. We can map Function Call Instances to Clusters by mapping it to the Cluster that contains its Shape. Two Function Call Instances are in the same group if they map to the same Cluster. Since the difference measure between Function Call Instances δ satisfies the Ideal Grouping Criteria, which we identified as being necessary and sufficient for a difference measure to agree with human intuition, and since all Function Call Instances in a given group differ by at most *cluster_threshold* (= 1.5 in our algorithm), our grouping of Instances therefore agrees with human intuition about what is similar and what is different. As an added bonus, we find that the number of Clusters is quite low. This makes it possible to visualize the Clusters on a timeline.

To visualize Clusters on a timeline, recall that the Function Call Instances contained in the Cluster are actual occurrences of function calls in the trace. There can be many occurrences of a single Function Call Instance at different time intervals. Each of these occurrences is an occurence of the Cluster. Thus, we can show the occurrences of the Cluster as time intervals on a timeline.

When creating our Clusters, we actually use a slightly modified version of the clustering algorithm that was described in section 8.1. In this modified version, we make sure we aren't adding a Shape into a Cluster that is a descendent or an ancestor of an existing Shape in the Cluster. This ensure that all occurrences of a Cluster are disjoint. In addition, we do some extra processing to group Clusters together so Clusters in a group don't overlap. This way, we can display the occurrences of all Clusters in a group on a single ribbon in different colors. This allows us to display 80 Clusters from a single thread in just 16 ribbons (about 5 Clusters per ribbon). Minimizing the number of ribbons is important, since a key contributions of our work is to be able to visualize the activity of many threads on one screen. Pseudocode for the Clustering algorithm can be found in Appendix A.

We show the timeline visualization below for threads 1, 7, 9, 10, and 11. Each have about 15 ribbons, and each ribbon shows about 5 Clusters if 5 difference colors. Note that the Clusters are shared between threads because at no point did we make a distinction between an occurence of a Function Call Instance in one thread versus another thread.

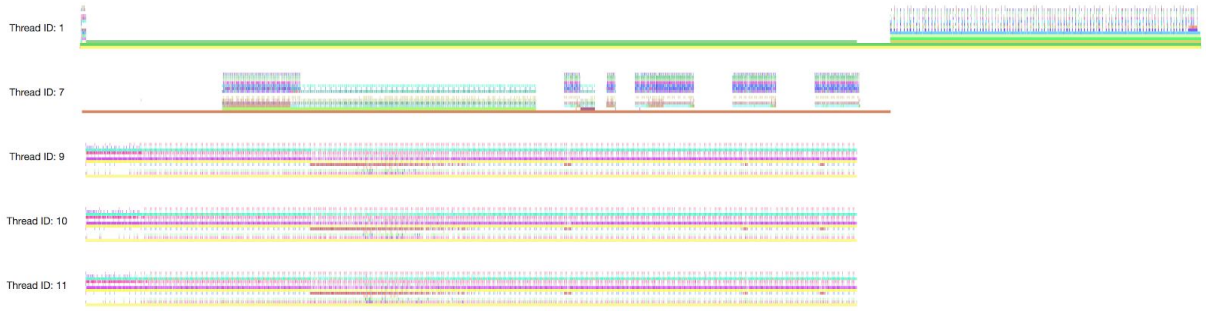
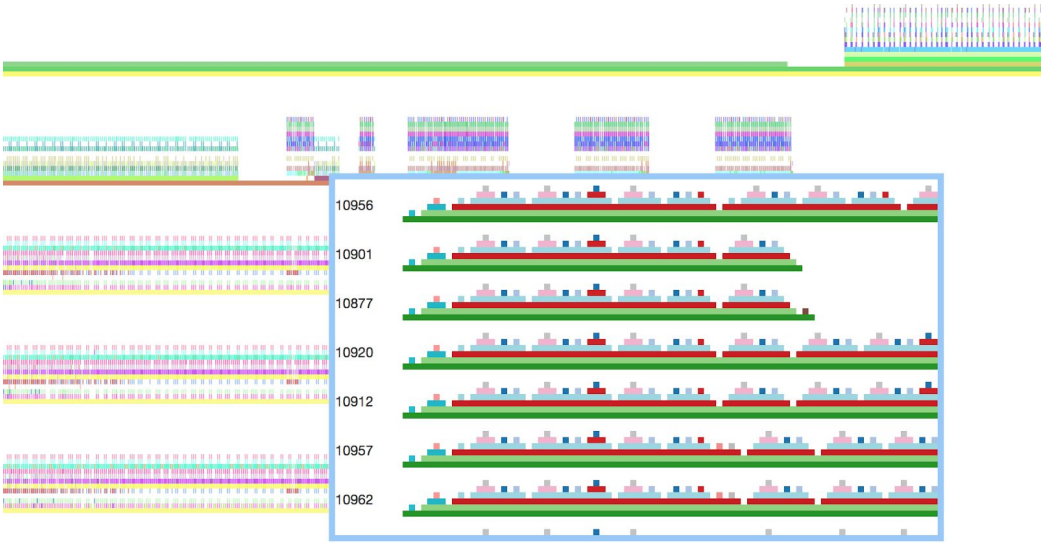


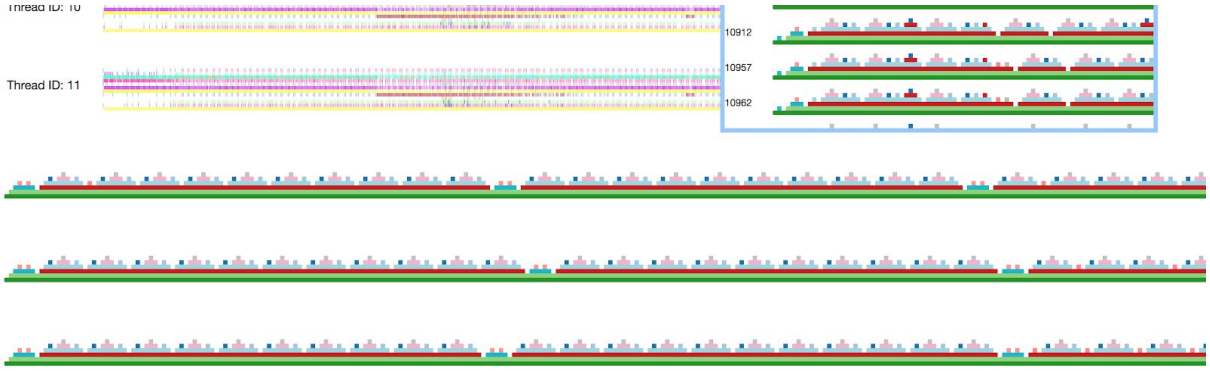
Figure 4.

Here, we can clearly see the similar behavior of threads 9 to 11. We include in our visualization a hover info box that shows all of the Shapes that make up the Cluster (Figure 5a). Clicking on an occurrence of a Cluster shows the occurrence of the Function Call Instance at that point in time. (Figure 5b). Finally we can configure our tool to instead display the occurrence of the Function Call Instance using TimeSquared [5], which shows the occurrence in time domain (Figure 5c).

a)



b)



c)

Figure 5.

9 Advantages of our Algorithm and Future Works

Our algorithm for grouping Function Call Instances is justified rigorously, and the empirical results for the trace we analyzed are pretty compelling. Another advantage we haven't touched on is the speed of our algorithm. In a simple Java implementation, finding the Shapes in the trace and the intervals in which they occur only took a few minutes. Computing the distance metric and clustering the Shapes took only seconds. But the most compelling speed characteristic of our algorithm is that it's highly parallelizable. Each thread in the trace can be processed independently. In fact, a single

thread in the trace can be partitioned into arbitrarily small partitions, and each partition can be processed independently. Therefore, our algorithm can process arbitrarily large traces in arbitrarily short periods of time, given enough computing power.

In this work, we only briefly touch on the visualization aspect of our Clusters. We display the Clusters in a simple manner, with fairly primitive querying capability (infobox, clicking). Our visualization tool contains no statistics about Clusters, Shapes, and Function Call Instances. In [2], it was observed that developer spend a lot of time chasing latency spikes and throughput drops. Our tools don't provide facilities to address this need. As a future work, we could detect latency spikes and throughput drops and we draw a layover on the timeline showing which clusters they occured in.

10 Conclusion

In this paper, we classified Function Call Instances in a trace via the notion of Shapes, the Shape metric, and Clusters. This grouping strategy was rigorously proven to have desirable and intuitive properties, and the empirical results were compelling. We then displayed our Clusters on a timeline visualization, providing an interface for a human to see the execution of a whole trace 1) without being overwhelmed by the amount of information, and 2) without losing an intuitive understanding of the elements in the visualization (Clusters, Shapes, and Function Call Instances).

References

[1] Proof of the Hausdorff Metric: https://www.math.upenn.edu/~brweber/Lectures/USTC2011/Lecture%205%20-%20Gromov-Hausdorff%20distance.pdf
[2] Alexandra Fedorova, Craig Mustard, Ivan Beschastnikh, Julia Rubin, Augustine Wong, Svetozar Miucin, and Louis Ye. 2018. Performance comprehension at WiredTiger. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). ACM, New York, NY, USA, 83-94. DOI: https://doi.org/10.1145/3236024.3236081
[3] The DINAMITE Toolkit. https://dinamite-toolkit.github.io.
[4] Intel Processor Trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing
[5] TimeSquared. https://github.com/dinamite-toolkit/timesquared

Appendix A

We present pseudocode for the clustering and visualization algorithm:

Listing A.1:

```
class Cluster:
    int functionId;
    Set<Shape> shapes;
```

```
    Cluster(functionId, shapes):
        assert(not shapes.isEmpty());
        this.functionId = functionId;
        this.shapes = shapes;

    int calculateDepth():
        return 1 + (max depth of shapes);

function createClusters(shapes):
    Sort shapes by the shape's depth in ascending order;
    List<Cluster> clusters = [];
    for (shape in shapes):
        shapeAdded = false;
        for (cluster in clusters):
            if ((shape.functionId is cluster.functionId)
             && (distance of shape to all shapes in cluster is less than or equal to
                 cluster_threshold (= 1.5))
             && (shape isn't a descendent or ancestor of any shape in cluster)):
                cluster.shapes.add(shape);
                shapeAdded = true;

        if (not shapeAdded):
            clusters.add(new Cluster(shape.functionId, {shape});

     return clusters;

function createClusterGroups(clusters):
    // We first sort the clusters so that clusters with the deepest shapes appears
    // first. Thus, the deepest clusters are grouped first.
    Sort clusters by the cluster's depth in descending order;
    List<List<Clusters>> clusterGroups = [];
    for (cluster in clusters):
        clusterAdded = false;
        for (clusterGroup in clusterGroups):
            if (none of the shapes in cluster are descendents of any of the shapes
                in any of the clusters in clusterGroup, and vise versa):
                clusterGroup.add(cluster);
                clusterAdded = true;

        if (not clusterAdded):
            clusterGroups.add([cluster]); // create a new cluster group

function visualizeShapes(shapes):
    clusters = createClusters(shapes);
    clusterGroups = createClusterGroups(clusters);
    // by sorting the clusterGroups, the shallowest clusters are drawn in the
    // top most ribbon
    Sort the clusterGroups by the depth of the deepest cluster in a group, in
        ascending ordering;
    for (clusterGroup in clusterGroups):
        Draw all clusters in the clusterGroup in one ribbon;
```
