Multiscale Visual Comparison of Execution Traces

Jonas Trümper$^{(a)}$ Jürgen Döllner$^{(a)}$ Alexandru Telea$^{(b)}$

***Abstract*—Understanding the execution of programs by means of program traces is a key strategy in software comprehension. An important task in this context is comparing two traces in order to find similarities and differences in terms of executed code, execution order, and execution duration. For large and complex program traces, this is a difficult task due to the cardinality of the trace data. In this paper, we propose a new visualization method based on icicle plots and edge bundles. We address visual scalability by several multiscale visualization metaphors, which help users navigating from the main differences between two traces to intermediate structural-difference levels, and, finally fine-grained function call levels. We show how our approach, implemented in a tool called TRACEDIFF, is applicable in several scenarios for trace difference comprehension on real-world trace datasets.**

***Index Terms*—Trace analysis, Software visualization, Program comprehension.**

I. INTRODUCTION

Software maintenance is an important part of the software engineering lifecycle [8], [24]. Within maintenance, program comprehension accounts for over 40% of the effort [4]. Along static analysis of the structure and dependencies of a system, dynamic analysis of the system execution, or *trace analysis*, is a key element for comprehension. Trace analysis can expose the interaction of software artifacts at run-time, including aspects such as late binding and data-dependent execution paths, which are hard to find via static analysis [38]. Moreover, comparing traces can yield unique insights into certain behavior aspects: While comparing traces from different versions or even from different programs helps finding which effects code changes have on runtime behavior, comparing traces from multiple runs of the *same* program helps finding why different inputs do (or do not) result in different outputs, and why multiple executions of the same functionality result in different outcomes (non-determinism). The latter is important to ensure stability of execution in various deployment configurations, which includes finding why programs exhibit different behavior when running on different machines.

Trace analysis is hard for several reasons. First, the sheer amount of data generated during tracing poses various analysis and (visual) representation challenges [38]. Second, the analysis has to depict many types of information: time-stamps, object identities, static program structure, and the relationships between such entities. If we want to *compare* two traces rather than analyze a single trace, the data volume doubles, so we need scalable and effective ways to show similarities and differences between the two traces.

The authors are with: (a) Hasso-Plattner Institute, University of Potsdam, Germany; (b) Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen, The Netherlands. E-mail: {jonas.truemper@hpi.uni-potsdam.de, juergen.doellner@hpi.uni-potsdam.de, a.c.telea@rug.nl}. This work was funded by the Federal Ministry of Education and Research (BMBF), Germany within the InnoProfile Transfer research group "4DnD-Vis".

In this paper, we present a visualization method for the interactive comparison of large traces from multiple runs of the same program. In the visualization design, we focus on two main goals. First, we address visual *scalability* by a multiscale design that supports exploration from coarse-grained events of interest, such as aggregated execution similarities and differences, to fine-grained events, such as function-level call similarities. For this, we use and extend several visual metaphors: space-filling plots (for the overview), shaded icicle plots and tube bundles (for the intermediate level), and attribute color-mapping and edge bundles (for the fine-grained level). We propose several interaction mechanisms to help specific user tasks at each level-of-detail, such as finding the most (dis)similar execution fragments; explaining these (dis)similarities by highlighting differences such as execution swaps, call time-shifts, and call durations; and finding execution fragments replicated several times between traces.

We describe our visualization by the task-oriented model of Maletic *et al.* [19]: Our *task* is to help users to compare large-scale traces, specifically to (a) detect execution regions that are (dis)similar; (b) explain the (dis)similarities at several levels of detail; and (c) correlate execution (dis)similarities with static program structure. Our *audience* includes software engineers who want to understand execution aspects of large software systems. The visualization *targets* static program structure, trace information (function calls and call durations) from two traces, and similarity relationships between the two traces. We *represent* these data using a space-filling plot (for overviews), icicle plots (for the call structure), and a multiscale bundling metaphor (for the trace-to-trace similarity relationships). Finally, the visualization *medium* consists of a standard screen with two linked views.

II. RELATED WORK

Visual trace exploration has a long history in program comprehension, and can be classified as follows.

Activity views: Execution traces are often visualized by different variants of icicle plots. The horizontal axis maps time, *e.g.*, function call start and end moments [33] or memory block allocation and release moments [21]. The vertical axis maps call stack depth [33] or memory block address ranges [21]. Stacked timelines allow comparing the evolution of several time series, such as repository commit activities [37], to find event correlations. Multivariate visualization, *e.g.*, scatter plots and dimensionality reduction techniques, help finding correlations in high-dimensional datasets, such as multi-metric log files, or between datasets, *e.g.*, profiling data from different execution traces [18], [20]. Peer-to-peer download metrics [36] and execution traces [29] are shown via linked Cartesian 2D plots.

Correlating views: Structure views are used to show the *static* system structure that is mined, *e.g.*, by static program analysis [23]. Activity views are used to show *dynamic* data such as execution or event logs. The linked views technique is frequently used to correlate the two. For example, Cornelissen *et al.* link a radial bundled node-link view (for static function calls), an icicle plot (for static system structure), and a call timeline (for dynamic execution information) by means of selection and brushing to show which subsystems are active in a given execution phase [5]. Similar techniques are used in ISVis [14] (link execution and structure), Jinsight [25], [26] (link execution and text), and Tarantula [15] and Gammatella [16] (link structure and text).

Comparing sequences: Traces, or more generally (ordered) sequences, can be compared by various techniques. For sequences that also have a hierarchical structure, such as program traces, many tree comparison methods exist [9]. TreeJuxtaposer [22] draws the two trees as dendrograms side by side and uses color and interaction to highlight (dis)similar subtrees. Holten *et al.* [13] extend this idea; trees are drawn as icicle plots, and similar leaf nodes are explicitly connected with bundled edges [12]. CodeFlows compares hierarchies of non-uniform depth using shaded tubes [31]. Beck *et al.* use a similar design to compare multiple hierarchies [1].

Although the bundled edge metaphor is visually scalable, it is best suited to show how *entire* subsequences correspond to each other. Finer-grained events, such as permutations within similar subsequences, and also edges linking non-leaf nodes, easily get lost within a bundle due to the inherent edge overlap.

To find similarities, several techniques exist that operate on hierarchical structures or (ordered) entity sequences. Ovation [7] uses a trace-specific similarity model based on manually specified attributes, such as function name, to find repetitions and similar patterns. TreeJuxtaposer finds tree similarities by computing the ratio of two unordered sets describing the tree's nodes. In contrast to these fuzzy matching techniques, De Pauw *et al.* [6] use exact matching to classify repetitions in web service traces. Hamou-Lhadj and Lethbridge [11] use a hashing-based approach to find patterns and remove repetitions from traces. Code clone detection techniques propose similar mechanisms to find (nearly) similar sub-sequences in a hierarchy (see references in [31]). All such techniques can be used, with small adaptations, in our trace comparison context.

Scalability: Visual scalability, a long-standing challenge [25], is addressed by visualizing restricted *ranges* of the execution data [28]; *aggregating* trace data into coarse-scale event graphs [3], [10], [17], [26], [27]; and by visual *subsampling* techniques that combine subpixel-size events directly in screen space [5], [21]. Although effective, all such techniques have their limitations: Range visualization restricts the insight to a predefined subset of an execution; aggregation can produce a too coarse execution representation; and visual subsampling does not show execution structure (call nesting).

Our trace comparison goal combines all the above challenges: Visualize pairs of large hierarchical sequences (hundreds of thousands of calls in two traces), show call duration and stack-depth information for each such item, and show many-to-many similarities between calls located at any hierarchy level.

III. Trace Comparison

We model a trace as a tree $T = \{f\}$ of function calls

$$f = (F, t^s \in \mathbb{R}^+, t^e \in \mathbb{R}^+, p \in T, C = \{c_i \in T\}). \qquad (1)$$

Here, F represents the definition, or identity, of the called function. Depending on the application and data availability, this can be a full syntax tree description of the function declaration or just the fully qualified function name. The values t^s and t^e, where $t^s < t^e$, are the start, respectively end moments of the call. The caller of f is denoted by p. The set C holds the children, or callees, of f, ordered by call times, *i.e.*, $\forall c_i \in C, c_j \in C, i < j | t^e(c_i) < t^s(c_j)$. Further, we denote the call stack rooted at f by $S(f)$.

To compare two traces T_A and T_B, we first design a so-called similarity function $s : T_A \times T_B \to [0,1]$, where for any two calls $f_A \in T_A$ and $f_B \in T_B$, $s(f_A, f_B)$ gives the similarity of the entire call stacks $S(f_A)$ and $S(f_B)$. We compute s using the method originally applied for detecting repeating patterns in a single trace [2], but now using two traces, as follows. Each function definition F is given a unique ID. For each call stack $S(f)$, we compute the set $\Gamma(f)$ containing all IDs of calls in S

$$\Gamma(f) = \{F' | \exists f' \in S(f), F(f') = F'\} \qquad (2)$$

Next, given two calls $f_A \in T_A$ and $f_B \in T_B$, we compute

$$s(f_A, f_B) = \frac{\|\Gamma(f_A) \cap \Gamma(f_B)\|}{\|\Gamma(f_A) \cup \Gamma(f_B)\|} \qquad (3)$$

where $\|\cdot\|$ denotes set size. For details, we refer to [2], [33].

The above process delivers a potentially very large set of pair-wise similarities between call stacks in the two traces, so we reduce this set as follows. In practice, we are interested only in call stacks having a large similarity $s(f_A, f_B) > \tau$. Setting $\tau \in [0.1, 0.3]$ has given good results in our trace comparisons. When such stacks exist, we say that there exists a *match* $k(f_A, f_B)$. For any such k, there exist also several matches $k'(u, v)$, where $u \in S(f_A)$, and $v \in S(f_B)$: two similar call stacks share similar substacks. We call the set of matches k', which explain k, a *group* $G(f_A, f_B)$. We further call k the *root* match of G, and denote k as $R(G) = (G_A, G_B)$.

We next partition all computed matches into a minimal set of groups, as follows. Starting with no group, we traverse one of the trees, *e.g.* T_A, in breadth-first order. For each encountered call $u \in T_A$, we iterate over its matches $k(u, v \in T_B)$. If there is no current group G or u is not contained in G_A or v is not contained in G_B, we create a new empty group and make it current. Otherwise, we add k to G and continue the traversal. We later use groups to create a visual trace comparison (Sec. IV-B).

All in all, the trace comparison described above delivers a set $K = \{k = (f_A \in T_A, f_B \in T_B)\}$ of groups containing matches between sufficiently similar stacks, *i.e.*, $s(f_A, f_B) > \tau$. We next show how we use these groups and the hierarchical trace data to visually analyze trace similarities.

IV. Visualization Design

We use a focus-and-context design that follows the well-known information-seeking mantra "overview, zoom and filter, and details on demand" [30] (see Fig. 1). First, we select a pair of already computed traces (T_A, T_B) that we wish to compare. If match data (Sec. III) is available for this pair, we use it directly, else we compute it on demand and store it for later use. After the trace pair and match data are loaded, we use the *overview* window (described next in Sec. IV-A) to find interesting execution areas, or *focus* areas, in the two traces. These can be areas in a trace where many matches exist with the other trace. Alternatively, we can select specific focus areas in the two traces and compare them – for instance, compare two executions around the same moment. After selecting the focus areas, we can interactively examine the *comparison* window to get insight on the (dis)similarities of the call stacks in focus (see Sec. IV-B).

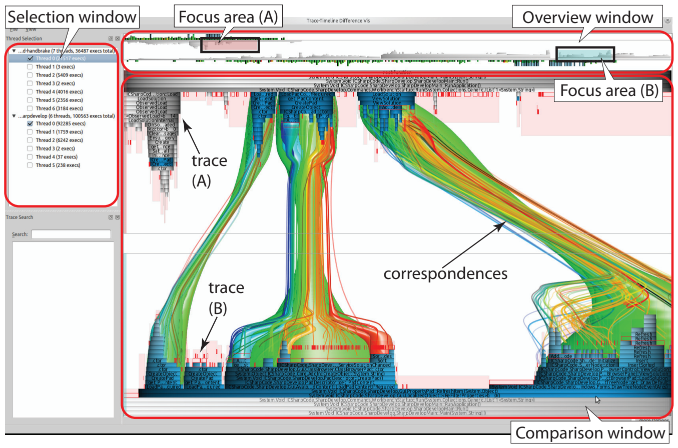

Fig. 1. Trace comparison visualization design.

A. Overview Visualization

The overview visualization shows an aggregated view of both traces (Fig. 2), with the upper part of this view dedicated to T_A and the lower part to T_B. For the full time extent of each trace, we draw two graphs. For T_A (Fig. 2, top trace), the lower graph shows an icicle plot of the call stack. The upper graph shows a bar chart drawn over N equal-sized time intervals, with N set so that each interval maps to 10 screen pixels.

For each such time interval $[t_{start}, t_{end}]$, we draw a bar whose height encodes the sum of similarities s of the matches that the trace has over $[t_{start}, t_{end}]$. For T_B (Fig. 2, bottom trace), we draw the same call stack and bar graphs, but mirrored in the y direction. The inner icicle plots of the two traces form a zoomed-out display similar to [34], which helps finding deep calls or long-duration calls. The bar chart interpretation is simple: High bars show execution areas where there are many strong matches.

For each bar, we also compute the relative start-time δ of the matched calls in the other trace over $[t_{start}, t_{end}]$, *i.e.*

$$\delta = \sum_{f \in T | k(f,g) \in K \ \wedge \ t_{start} \leq t^s(f) \leq t_{end}} |t^s(f) - t^s(g)| \tag{4}$$

and color the bar based on δ using a red-gray-green colormap. Red bars indicate matches from the current trace to the *past* of the other trace; gray bars show matches between the two traces which are *aligned* in time; and green bars show matches from the current trace to the *future* of the other trace. When hovering the mouse over a call f, we set the background of all bars which have matches of calls in $S(f)$ to blue. For example, in Fig. 2, the user sees calls which appear compactly grouped in the focus region of T_A. The blue bars in T_B show that these calls have matches scattered over a large portion of T_B. Very few of these are in T_B's focus. Thus, to better examine these matches, the user can now shift T_B's focus to the left.

The above mechanisms allow users to quickly find several zones of interest in the two traces: Low bars show execution areas which have no, or very low-similarity, matches in the other trace. These areas are likely less interesting for further analysis. High gray bars indicate areas which have well-aligned matches. High red or green bars indicate areas which are executed at different time instants in the other trace, which are arguably the most interesting to analyze. Bar backgrounds help panning the foci of the two traces to find matching calls. Using these cues, users can select the areas of interest for their specific use-cases, and next use the comparison window (Sec. IV-B) to gain finer-grained insight.

B. Match Visualization

Given a match-rich focus area in the trace-pair, the comparison window shows details on these matches, as follows (see Fig. 3). First, we render the two call stacks using two mirrored icicle plots, where the x extent (width) of the elements shows call duration, and the y axis encodes stack depth. We also use cushion shading to emphasize the call stack structure [34]. Finally, we outline found call repetitions (Sec. III) in red (cf. [2]).

Next, we use edge bundles to connect matched calls in the two traces. We start by the hierarchical edge bundling (HEB) design of Holten *et al.* [12], [13]: Match edges are routed along a tree structure computed using the hierarchy represented by the icicle plot nodes, mirrored along the x axis. However, our data differs from Holten's in several important respects:

R1: Our icicle plot nodes have highly different widths (since we encode call duration in width);

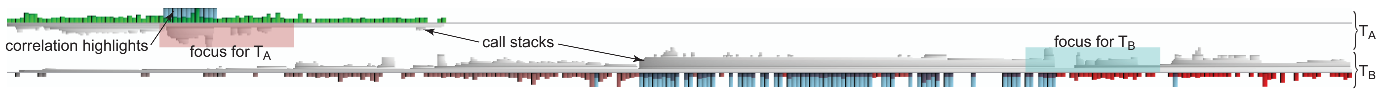

Fig. 2. Overview visualization. The top and bottom parts show the match highlights and call stacks for the two compared traces, respectively.

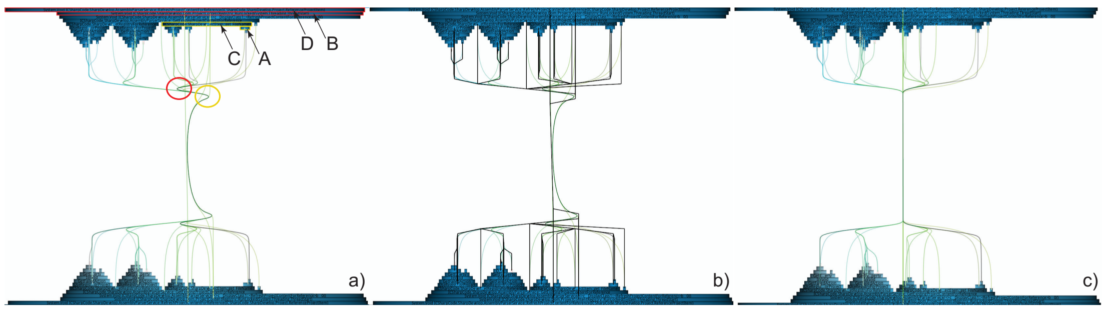

Fig. 3. Original HEB bundling (a) and its control tree (b). Undulation artifacts due to node widths are visible. Our modified bundling (c).

R2: We have to draw matches between non-leaf nodes (which encode execution similarities at coarser scales);

R3: The empty vertical space between the two icicle plots can be quite small (due to the arbitrary depth of the call stacks). Nodes of the upper icicle plot can fall below the lower icicle plot nodes (Fig. 4 b). This never happened in earlier HEB designs;

R4: Many icicle plot nodes have subpixel size (short function calls). For large traces, using such nodes in the HEB algorithm creates cluttered bundles and slow-to-draw images.

Given the above, using the original HEB method creates strong visual artifacts. Several examples follow. Fig. 3 a shows two wave-like structures in the bundles (marked in red and yellow). These appear since the nodes A and C are not centered horizontally within their respective parents B and D (issue **R1**). Figs. 4 a,b show several very sharp bundle bends (red markers) and undulations (green marker) due to the relatively small space between the traces (issue **R3**). For large traces containing hundreds of thousands of calls such problems become only bigger and more frequent.

To solve such issues, we modify the HEB layout, as follows. First, for call stacks containing only nodes narrower than one pixel, we solely render their bounding box (pink rectangles in Figs. 4 a,b). This upper-bounds the number of rendered shapes at a time, which ensures a high frame rate, and also makes the image less cluttered (issue **R4**). Next, we reserve a horizontal band B centered around the mid-line of the match view (see Figs. 4 a,b). This is the area where bundling will take place. For each group G, we build a separate control tree-pair, one tree for $S(G_A)$ and one for $S(G_B)$. As tree control points, we use only the centers of those nodes in $S(G_A)$ and $S(G_B)$ which are broader than 1 pixel *and* also fall outside B, which we next call *key nodes*.

When constructing the control tree for a group G, we also clamp the x coordinate of each node f to the x range of the control-points of all child nodes in $S(f)$ that have matches in G. This shifts the control points horizontally so that the resulting control tree has far less right-left twists. In turn, this reduces the amount of horizontal undulations in the resulting bundles, and thereby removes artifacts of type **R1** and **R2** (compare Fig. 3 c with the original HEB in Fig. 3 a).

Given the above control tree, we next add the non-key nodes (narrower than 1 pixel or falling within B) to the tree, as follows. For each non-key node n, we ascend its call stack until we find a parent p_n which was added to the control tree. Such a parent always exists, as nodes closer to the call-stack root are broad and far away from B. Next, we scan the control tree downwards from p_n and find the node q whose control point is geometrically closest to n, and add n as a child of q in the control tree. Hence, all non-key nodes are added as leaves to the control tree. As such, the coarse-level structure of the control tree and, more importantly, its height are not changed, and matches from non-key nodes are smoothly 'merged' into the bundles of key nodes. Comparing Figs. 4 c,d, which use our modified bundling, with Figs. 4 a,b (original HEB), we see that the undesired sharp bends and undulations have been removed, and the bundle appears centered within the x extents of the matched nodes.

Let us note that most existing applications of HEB feature edges which connect equally-sized nodes, located at the same hierarchy level, and which are laid out regularly, *e.g.* along lines [1], [13] or circles [13]. Our modified bundling relaxes these restrictions, so it can be used in other contexts where the original HEB method delivers suboptimal results.

C. Multiscale Visualization

Although our modified HEB method helps answering questions on the sizes and time offsets of matching execution fragments, several questions remain (see *e.g.* Fig. 5). First, a HEB rendering cannot show *permutations* in matched sequences, since the inherent overdraw caused by tight bundles makes it very hard, if not impossible, to follow individual edges. Such permutations are inherent to our match computation. Second, we recall that matches between shallower call levels are more relevant than deeper-level matches (Sec. III). However, HEB renders all edges identically, so we cannot easily spot more important edges. Finally, HEB represents edges as 1D curves. This makes it hard to see, for such an edge, which are the durations of the matched elements it connects.

We address the above issues by a multiscale HEB

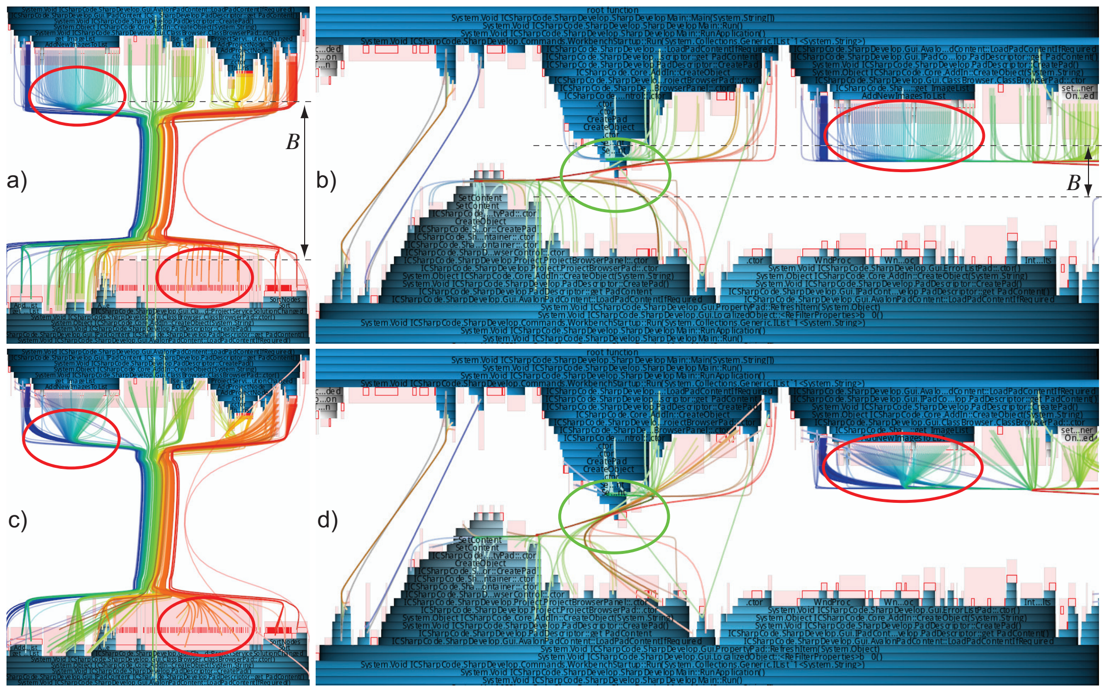

Fig. 4. Original HEB bundling (a,b) showing sharp bends (red markers) and undulations (green marker). Our modified bundling (c,d).

visualization, inspired by image-based edge bundling (IBEB) [32], *i.e.*, we draw edges as shaded 2D tubes instead of 1D curves as in HEB (Fig. 5 c). This is explained next.

Tube layout: Consider two calls f and g that are connected by a match k. Let x_f, y_f, w_f be the x and y coordinates of the center and width of the icicle plot rectangle for f, and x_g, y_g, w_g the similar quantities for g (see Fig. 5 a). Let γ be the modified HEB curve that connects the two centers, computed as described in Sec. IV-B. Let $t:[0,1]$ be an arc-length parameterization for γ. We construct two curves γ^L and γ^R which represent the left, respectively right, curved borders of our tube shape. If $\gamma^L = (\gamma^L_x(t), \gamma^L_y(t))$, we set

$$\gamma^L_x(t) = \gamma_x(t) - \phi(t)\left((1-t)\frac{w_f}{2} - t\frac{w_g}{2}\right)$$
$$\gamma^L_y(t) = \gamma_y(t)$$

where $\phi:[0,1]\to[0,1]$ is a function that models the gradual shrinking, or thinning, of the tube from its ends towards its center. Profiles that generate bundle-like tubes are given by

$$\phi(t) = \lambda + \frac{1-\lambda}{2}(1+\cos 2\pi t). \quad (5)$$

Similarly, we construct the tube's right-border curve γ^R as

$$\gamma^R_x(t) = \gamma_x(t) + \phi(t)\left((1-t)\frac{w_f}{2} - t\frac{w_g}{2}\right)$$
$$\gamma^R_y(t) = \gamma_y(t).$$

Tube shading: We visually emphasize our tube bundles by pseudo-shading using a cushion-like luminance texture, dark at the borders γ^L and γ^R and bright in the center (γ). For this, we define a 1D convex parabolic shading profile ranging from 0 (dark) to 1 (bright), similar to the well-known cushion treemap design [35]. Next, we render our shape by discretizing γ^L and γ^R with 50..100 sample points, and drawing the resulting quads using a 1D texture encoding our shading profile. Fig. 5 c shows the result: The tubes appear like 3D shaded shapes that smoothly connect their corresponding icicle-plot elements. The design of the profile ϕ ensures that the tubes follow the shapes of their corresponding edge bundles – compare *e.g.* Figs. 5 b,c. This allows us to smoothly toggle between line and tube visualizations, or generate visualizations containing both tubes and lines in the same image.

Our tubes are visually quite similar to IBEB bundles. However, important differences exist: First, while IBEB constructs tube-like shapes in order to *simplify* an existing HEB drawing, our tubes represent *one-to-one* our edges, as our aim is to show the time (horizontal) extents of all matched icicle plot nodes. Second, IBEB has a highly involved implementation, which uses edge clustering, image blurring, distance transforms, and skeletons. We only use a few simple curve interpolation and hardware-accelerated 1D texture mapping operations. Consequently, our method renders the same amounts of shaded tubes as IBEB, roughly 10..20 times faster than the latter. This is essential for interactive analysis, as typical trace-pairs can contain thousands of matches.

Tube stacking: To combine our shaded tubes in a final image, we add a z (depth) coordinate γ_z to our curve γ, computed by linearly interpolating the call-stack depths of its endpoints f and g, and next set $\gamma^L_z = \gamma^R_z = \gamma_z$. Rendering our tubes with standard depth (Z) buffering shows higher-level (coarser) matches behind lower-level (finer) ones. Due to Z buffering, we also directly handle matches that connect different stack-depths in the two traces.

Fig. 5 c shows the overall result of our bundled tubes.

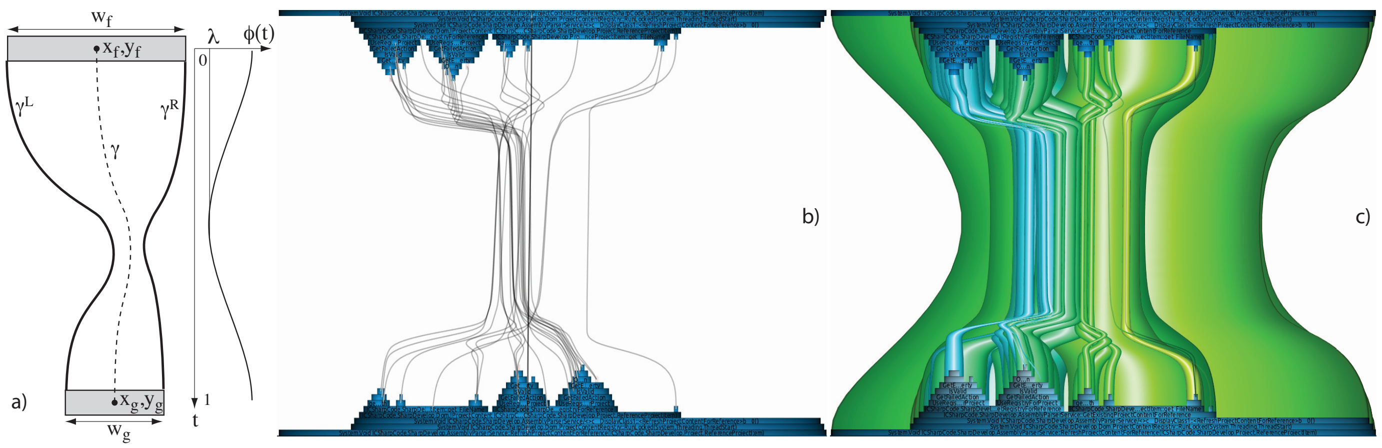

Fig. 5. Tube design (a). Tube bundles (c) add more information on the match start and endpoints and nesting than line bundles (b).

In contrast to line bundles (Fig. 5 b), we now clearly see the time extents of the matched call stacks, encoded as tube thickness, and we can separate coarse matches (thick tubes) from fine ones (thin tubes). Also, crossings are now clearer. The overall result is a multiscale match visualization, where matches of high-level and long call stacks (which are more important) are visually prominent.

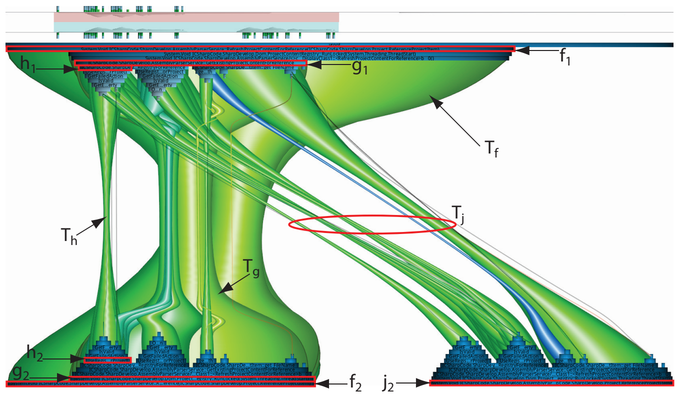

Fig. 6. Finding execution duplicates.

Finding execution replications: Tube bundles also help finding fragments from a trace that are replicated several times into the other trace. Fig. 6 shows this: At a coarse level, the largest visible tube T_f (behind all other tubes) shows a strong similarity between the largest part of the top trace (stack rooted at f_1) and the first part of the bottom trace (stack rooted at f_2) Finer-grained tubes *explain* this similarity: For example, the tube T_g shows that the above stacks are similar because the sub-stack rooted at g_1 (top trace) is similar to the bottom-trace sub-stack rooted at g_2. This similarity is in turn explained by the tube T_h, which shows that the sub-stack rooted at h_1 (top trace) is similar to the one rooted at h_2 (bottom trace). However, we see that the sub-stack rooted at g_1 (top trace) is *also* similar to a second sub-stack rooted at j_2 (bottom trace). This is shown by several diagonal tubes marked as T_j.

D. Attribute Mapping

We enrich our trace visualization by mapping several attribute values that are relevant to questions of interest. The key use-case is to *explain* the computed matches: Given two matched stacks, connected by HEB curves or tubes, we want to know *why* the two stacks are similar and *where* they differ. We address this as follows.

Finding permutations: As outlined earlier, our match computation is insensitive to permutations. This is desirable for discovering stacks that match regardless of call order. However, permutations mean execution-order differences that should be highlighted. Visually detecting small-scale permutations can be hard using the tube metaphor only. Given a match $k = (f, g)$ with call start times $t^s(f)$ and $t^s(g)$ respectively, we address this problem by mapping the difference $|t^s(f) - t^s(g)|$ to the saturation of a base color (red), and use the resulting color for our HEB curves or tubes. Fig. 7 shows the result: Matches with similar starting times show up as gray. Matches with different starting times appear as red. In our example, call A from the beginning of trace T_1 matches B at the end of trace T_2, and call C from the end of T_1 matches three times (D, E, F) at the beginning of T_2.

Finding trace-centric outliers: A generalization of the above use-case is to show whether a call f in a stack T_A occurs at the same relative position (with respect to T_A's root) as its match g in a stack T_B. To show this, we use the partition of the traces into groups (Sec. III). For a group G, rooted at G_A and G_B respectively in the two traces, we first select a viewpoint, *i.e.*, decide if we want to examine matches from the perspective of T_A or T_B. We do this by moving the mouse cursor in the upper half, respectively lower half, of the match view. If we select the viewpoint of T_A, we next color each match $k(f_A, f_B) \in G$ by the value $\frac{t^s(f_B) - t^s(G_B)}{t^e(G_B) - t^s(G_B)}$ using a rainbow (blue-to-red) colormap. The interpretation of this color mapping is as follows (see Fig. 8): Matched calls, which occur at the *same* relative moments in the two traces

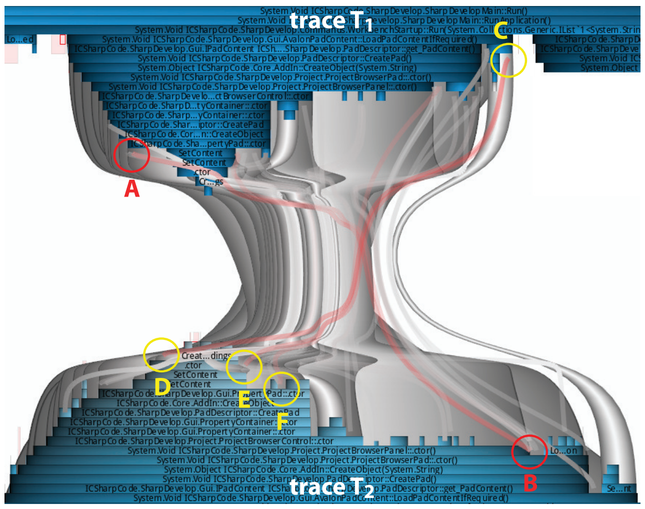

Fig. 7. Finding execution permutations.

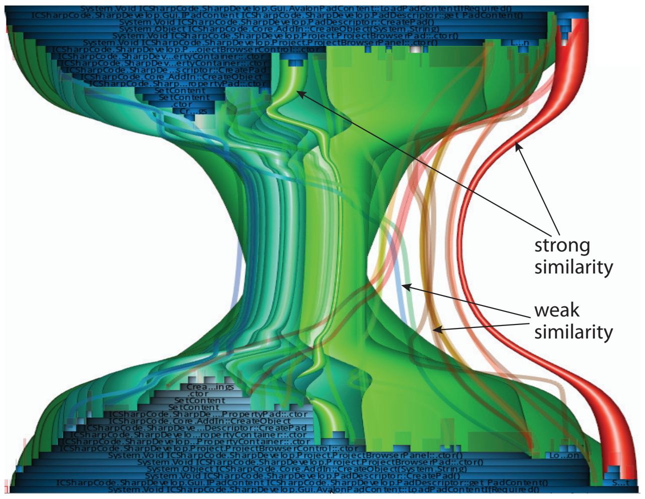

Fig. 9. Depicting match similarity.

(with respect to the start times of the root calls G_A and G_B respectively), have a color that follows the rainbow gradient, *i.e.*, are blue if they occur early, and red if they occur late. Calls $f_A \in T_A$ that occur relatively *later* in T_B or calls $f_B \in T_B$ that occur relatively later in T_A appear as red outliers on a cold (blue..green) background – see insets in Fig. 8. Similarly, calls in one trace, which are matched at *earlier* moments in the other trace, appear as cold outliers on a warm background.

Depicting similarity: A final use-case is to show the similarities s of the detected matches (Sec. III). This allows us to further separate strong (relevant) matches from less relevant ones, *i.e.*, further explain why two stacks are similar or not. For this, we map, for each HEB tube, its similarity s to the tube's transparency or strength of a white specular highlight: For tubes thinner than 16 pixels, we use transparency, since these tubes are too thin to show a specular highlight. For thicker tubes, we use highlights, since these tubes must be opaque so that the nesting effect (Sec. IV-C, Fig. 5) is visible. Fig. 9 shows the result: Tubes with strong specular highlights, like the red tube to the right, stand out in the image, and indicate strong (important) matches. Diffusely shaded or half-transparent tubes attract less attention, which is in line with them being weak (unimportant) matches. For example, we see that all tubes that diagonally cross the image are both thin and half-transparent. This tells that the two compared traces are quite similar, the differences being relatively short-lived call stacks (thin tubes) which are permuted between the two traces (crossing tubes) and which are not strongly similar (transparent tubes). Encoding the similarity in specular highlights and transparency has the advantage that we still can use hue for showing other attributes, as described earlier.

E. Interaction

Both traces in the match view can be interactively zoomed and horizontally panned (see Fig. 10 a and b respectively). Also, clicking on a call stack in a trace automatically aligns it with all matched groups in the other trace. This helps bringing into focus matched sequences in the two traces.

Brushing with the mouse over a call stack restricts color mapping (Sec. IV-D) to matches contained in groups rooted in that stack. All other matches are drawn in gray. This helps focusing the analysis on specific match groups.

The bundling parameters can be adjusted to obtain several effects: Tube thickness $\lambda \in [0,1]$ (Eqn. 5) can be set to create thinner tubes (with less occlusions, Fig. 10 c) or thicker tubes (which better show call nesting, Fig. 10 d). The thickness of the band B (Sec. IV-B) can be set to create shallower bundle control trees (which help following the main bundles, Fig. 10 e) or taller control trees (which help seeing where the tubes connect to the icicle plots, Fig. 10 f).

V. Applications

We describe the usage of TraceDiff for the analysis of a large trace-pair – approx. 150.000 calls and 1.500 function definitions. The two traces were recorded while an instrumented open-source C# IDE (approx. 1 MLOC, 45 contributers, 8 years of development) loads two different solution files, *i.e.*, varying input data. As we are running the same code twice, we expect to see strong overall correlation across the two traces. Figs. 11 a,b show a completely zoomed-out view of the compared traces. As we can see from the overview window, trace 1 (Figs. 11 c,d top) takes roughly a third of the execution time of trace 2 (Figs. 11 c,d bottom). Our questions are: Since these traces have significantly different lengths, do important similarities in the recorded executions yet exist? Where are these similarities, and which parts are different?

In the correlation view (Figs. 11 a,b), we see that icicle plot shapes for the two traces differ a lot. Hence, the two traces encode quite different dynamics in terms of call lengths and stack depth. Further inspection of the overview shows that trace 1 contains matches to trace 2 only within its first two-thirds (blue bars, overview top), while trace 2 contains matches over its full extent (blue bars, overview bottom). This is our first hint that the traces contain similar execution patterns.

Next, we want more insight into these patterns. For this, we

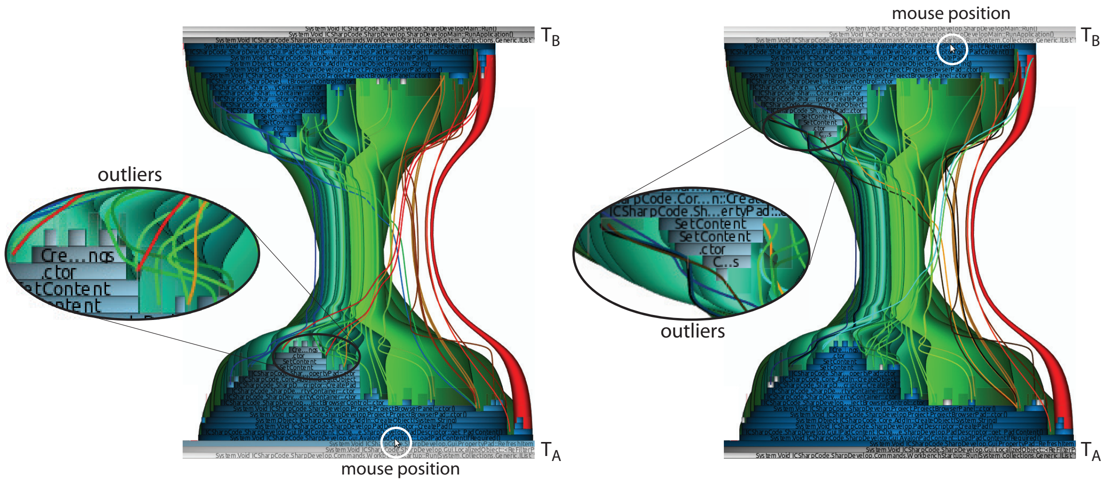

Fig. 8. Finding trace-centric outliers with respect to the bottom trace (left) and the top trace (right)

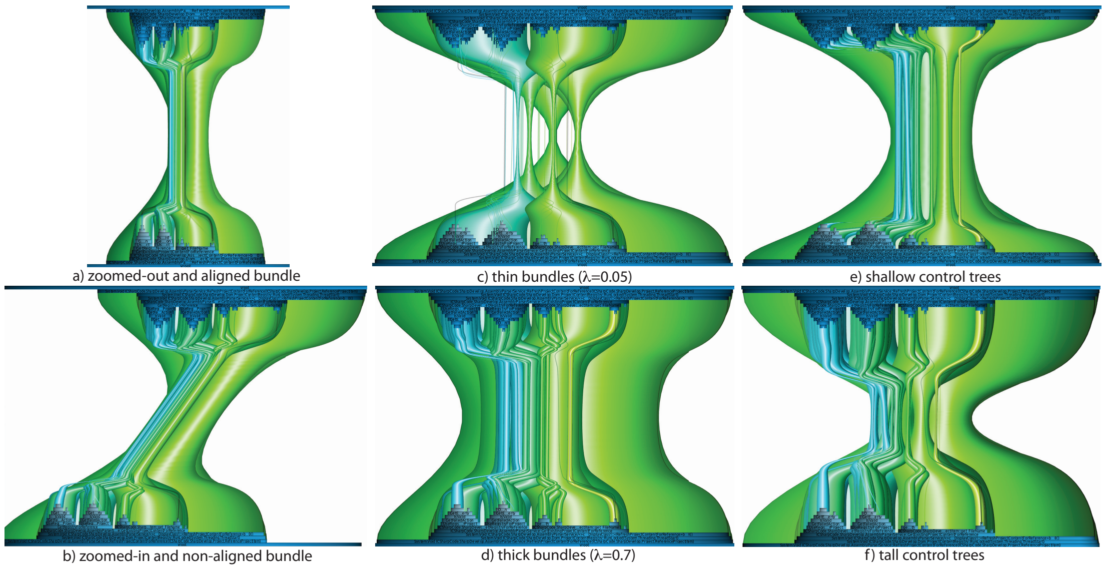

Fig. 10. Bundling parameters. The resulting bundle shapes are stable and readable for various zoom, pan, and tube shape values.

focus on the first two-thirds of trace 1 and on the entire trace 2. When we look at the match view, we see that there exist quite a number of execution similarities. At the coarsest level, we identify five match groups $(A-E)$. Three such groups (C,D,E) account for relatively short-duration sequences. The remaining two groups (A,B) account, together, for over 50% of the execution. Also, we discover that there are no matches between the begin and end phases of the two traces. We next use the permutation colormap (Fig. 11 c) to examine groups A and B, and quickly see several saturated red lines appearing: These are matched execution fragments that occur at different moments in their respective match groups. In group A, we see that the first phase of trace 1 (small bundle A') matches very well the last phase of trace 2 – the executed pattern has been shifted between the two traces. In group B, we find a more complex pattern: the first phase of trace 2 matches a large interval of trace 1 – the red lines in group B are concentrated at the bottom but fan-out at the top. Hence, the first phase of trace 2 has been spread over the whole execution of trace 1.

To learn more about the discovered matches, we now apply the trace-centric color mapping (Fig. 11 d), and move the mouse into trace 2. If we look at the color gradient in group A (trace 2), we spot several outliers (permutations) of the standard blue-to-red colormap (white markers, Fig. 11 d). These are functions that are called relatively earlier (blue lines) or relatively later (red lines) in trace 1. To see where these calls match in trace 1, we can visually follow the line colors from bottom to top. This outlines a second usage of our color mapping: Besides identifying time-shift outliers, colors help in following correspondences between the two traces. In contrast, the color gradient in group B (trace 2) does not show such interruptions of the rainbow pattern, which smoothly goes from blue (left) to red (right). Hence, the matches in group B indicate that the execution order between the two traces is preserved. A second difference between groups A and B is visible: At the right of A, we see a shiny orange tube (T, Fig. 11 d). This indicates that the last part of the sequences described by group A has a very strong similarity. We see no such shiny tubes in group *B*. This shows that the execution of *A* contains much stronger similarities than the execution of *B*. Finally, if we look at the last part of the matched sequences in group *B* (red lines), we see that these lines have a large vertical spread, both in trace 1 and 2 (dotted markers, Fig. 11 d). Hence, the last parts of these matched sequences occur over a short period of time (narrow red bundle) and deep call stack (large vertical spread). The entire analysis described above took around five minutes.

Finally, we note the added value of aggregating small-duration calls (Sec. IV-B): In Fig. 11, such calls are indicated by the relatively large pink rectangles. As we can see, there are several such tall rectangles, which have around 30% of the height of the match view. If we did not perform the call aggregation, there would be very little, if any, vertical space between the two traces in which to draw the bundles, and this would lead to an unreadable match visualization. Our aggregation and subsequent modified HEB layout creates sufficient vertical space for the bundle visualization.

VI. Discussion

Generality: The correspondence visualization, though demonstrated on traces, works for any hierarchical sequence comparison for which match data is available. The matches are not restricted in any way, *i.e.*, they can be many-to-many matches on any level in the hierarchy.

Visual scalability: An enhanced HEB technique eliminates the visual artifacts created by the original HEB. By combining this with a multiscale correspondence visualization, we can encode additional attributes in the correspondences, such as the width of matched elements. All in all, this lets us visually compare hundreds of thousands of calls in two traces at interactive rates.

Ease of use: The interaction techniques allow for easy user input; to explore the underlying trace and match data, users only need to learn how to point, click, zoom and pan. Brushing techniques implicitly translate point actions into selections both on the call stacks and the correspondence visualization.

Flexibility: By these inputs, users can easily adjust the analyzed subset of the data and adjust level of detail. The multiscale correspondence visualization automatically adapts to the selected level of detail. The color mappings address specific questions in the given context of trace comparison.

Limitations: While our visual design is definitely more scalable than those of other techniques, such as HEB or CodeFlows, it will as well create clutter for very large hierarchical sequences and numerous many-to-many matches.

VII. Conclusions

We have presented TraceDiff, a visual tool that proposes several novel interactive visualization techniques for the analysis of the similarity of large execution traces. We address visual scalability and readability by introducing a modified hierarchical edge bundling layout and icicle plot node aggregation. We extend edge bundles to shaded tube bundles in order to visualize the time-extents of execution patterns, and also explain execution matches by multiscale nesting. We use attribute mapping to colors and highlights to further add similarity information and also assist finding execution permutations and time shifts. We demonstrate our techniques on the analysis of a large execution trace.

Further work will address different designs for the tube bundles to encode additional attributes. Also, given the high visual scalability of our approach, we plan to extend its application to the visual comparison of multiple execution traces and of traces from different program versions or different programs.

References

[1] F. Beck, R. Petkov, and S. Diehl. Visually exploring multi-dimensional code couplings. In *Proc. IEEE VISSOFT*, pages 137–145, 2011.

[2] J. Bohnet. *Visualization of Execution Traces and its Application to Software Maintenance*. PhD thesis, Hasso-Plattner-Institute, Univ. Potsdam, Germany, 2010.

[3] A. Chan, R. Holmes, G. Murphy, and A. Ying. Scaling an object-oriented system execution visualizer through sampling. In *Proc. IEEE IWPC*, pages 237–244, 2003.

[4] T. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1999.

[5] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. In *J. Sys. & Software*, volume 81, pages 2252–2268, 2008.

[6] W. De Pauw, S. Krasikov, and J. F. Morar. Execution patterns for visualizing web services. In *Proc. ACM SOFTVIS*, pages 37–45, New York, NY, USA, 2006. ACM.

[7] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proc. USENIX COOTS*, pages 219–234. USENIX, 1998.

[8] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[9] M. Graham and J. Kennedy. A survey of multiple tree visualisation. *Information Visualization*, 9:235–252, 2009.

[10] A. Hamou-Lhadj. *Techniques to simplify the analysis of execution traces for program comprehension*. PhD thesis, Univ. of Ottawa, Canada, 2005.

[11] A. Hamou-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proc. IEEE WODA*, pages 33–36, 2003.

[12] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. In *Proc. INFOVIS*, pages 741–748, 2006.

[13] D. Holten and J. J. van Wijk. Visual comparison of hierarchically organized data. *CGF*, 27(3):759–766, 2008.

[14] D. Jerding and J. Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE TVCG*, 4(3):257–271, 1998.

[15] J. Jones and M. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proc. ASE*, pages 237–243, 2005.

[16] J. A. Jones, A. Orso, and M. Harrold. Gammatella: visualizing program-execution data for deployed software. *Information Visualization*, 3(3):173–188, Sept. 2004.

[17] D. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *IEEE Comp.*, 30(5):63–70, 1997.

[18] D. Leon, A. Podgurski, and L. J. White. Multivariate visualization in observation-based testing. In *Proc. IEEE ICSE*, pages 116–125, 2000.

[19] J. Maletic, A. Marcus, and M. Collard. A task oriented view of software visualization. In *Proc. IEEE VISSOFT*, pages 32–40, 2002.

[20] J. Moc and D. Carr. Understanding distributed systems via execution trace data. In *Proc. IWPC*, pages 60–67, 2001.

[21] S. Moreta and A. Telea. Multiscale visualization of dynamic software logs. In *Proc. EUROVIS*, pages 11–18, 2007.

[22] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Zhou. Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility. *ACM Trans. Graph.*, 22(3):453–462, July 2003.

[23] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[24] D. Parnas. Software aging. In *Proc. IEEE ICSE*, pages 279–287, 1994.

[25] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. ACM OOPSLA*, pages 326–337, 1993.

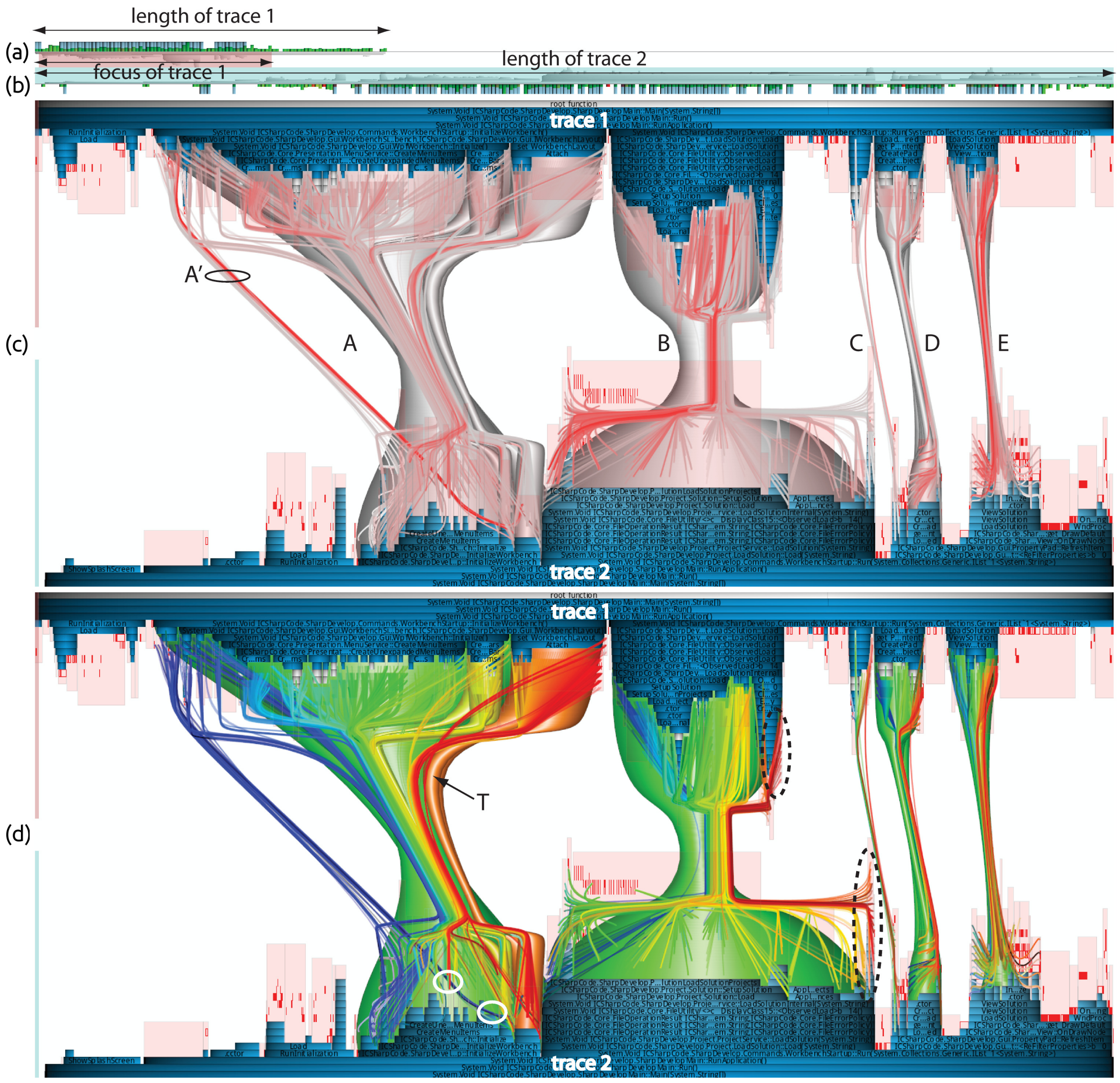

Fig. 11. Analysis for a large trace-pair: a,b) Overviews of traces 1 and 2; c) Finding coarse-level matched sequences and their permutations; d) Finding trace-centric time shifts.

[26] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proc. COOTS*, pages 219–234, 1998.

[27] S. P. Reiss. Visualizing java in action. In *Proc. ACM SOFTVIS*, pages 57–65, 2003.

[28] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. ICSE*, pages 221–230, 2001.

[29] J. Roberts. TraceVis: An execution trace visualization tool. In *Proc. MoDS*, pages 123–130, 2005.

[30] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. IEEE Symp. on Visual Languages*, pages 336–343, 1996.

[31] A. Telea and D. Auber. Code flows: Visualizing structural evolution of source code. *Comp. Graph. Forum*, 27(3):831–838, 2008.

[32] A. Telea and O. Ersoy. Image-based edge bundles: Simplified visualization of large graphs. *Comp. Graph. Forum*, 29(3):65–74, 2010.

[33] J. Trümper, J. Bohnet, and J. Döllner. Understanding Complex Multithreaded Software Systems by Using Trace Visualization. In *Proc. ACM SOFTVIS*, pages 133–142, 2010.

[34] J. Trümper, A. Telea, and J. Döllner. Viewfusion: Correlating structure and activity views for execution traces. In *Proc. TPCG*, pages 45–52. Eurographics, 2012.

[35] J. J. van Wijk and H. van de Wetering. Cushion treemaps: visualization of hierarchical information. In *Proc. INFOVIS*, pages 73–78, 1999.

[36] L. Voinea, A. Telea, and J. J. van Wijk. Ezel: a visual tool for performance assessment of peer-to-peer file-sharing networks. In *Proc. InfoVis*, pages 41–48, 2004.

[37] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: visualization of code evolution. In *Proc. ACM SOFTVIS*, pages 47–56, 2005.

[38] A. Zaidman. *Scalability Solutions for Program Comprehension through Dynamic Analysis*. PhD thesis, Univ. of Antwerp, 2006.