Abstract—Bloat, and particularly object churn, is a common performance problem in framework-intensive applications. Object churn consists of an excessive use of temporary objects. Identifying and understanding sources of churn is a difficult and labor-intensive task, despite recent advances in automated analysis techniques. We present an interactive visualization approach designed to help developers quickly and intuitively explore the behavior of their application with respect to object churn. We have implemented this technique in Vasco, a new flexible and scalable visualization platform. Vasco is designed to minimize the cognitive effort required for the visualization task. We demonstrate the effectiveness of our approach by applying it to three framework-intensive applications and identifying previously unreported churn in a commercial system.

I. INTRODUCTION

Modern programmers often assemble complex systems from multiple layers of middleware, frameworks and libraries. For instance, web applications typically consist of a relatively small amount of business logic on top of a large amount of complex middleware. This massive reuse and generality enable the rapid development of highly complex systems. However, this comes at a cost: such framework-intensive systems often require an inordinate amount of resources to accomplish seemingly simple tasks. For example, they can suffer from poor memory utilization, create hundreds of objects as the byproduct of simple computations, or spend a large amount of their time on unnecessary computations. Recently, this software bloat has prompted significant academic and industrial research (e.g., [1], [2], [3], [4], [5], [6]).

Bloat-related problems in a framework-intensive application often go unnoticed until it operates under its real load, after being deployed. Causes of bloat in such systems are typically very difficult to diagnose for two main reasons: (i) these problems are seldom located in a single method or even class, but are instead typically spread across large a region of the execution, and (ii) developers themselves are often unaware of the inner workings of frameworks and libraries used by their applications, and thus cannot fully evaluate the impact of their design decisions on the performance of their applications.

In previous work [7], we have explored object churn, a common performance problem in framework-intensive applications that consists of an excessive use of temporary objects. Temporaries can impose a significant overhead during the execution, not only because of the increased pressure on the garbage collector, but mostly due to increased initialization costs. It is common for applications to spend a considerable amount of time building temporary data structures only to discard them shortly thereafter.

Modern just-in-time (JIT) compilers can stack-allocate short-lived objects based on the results of an escape analysis. However, temporaries typically escape their allocating functions, and thus thwart such optimizations. More advanced JIT optimizations such as Jolt [8] are more effective than previous generic optimizations, but still miss many opportunities. Moreover, while Jolt can eliminate the cost of allocating and deallocating temporaries, the initialization cost of the temporaries remains.

More recently, automated analyses have been proposed to help developers pinpoint expensive copy operations between objects [9], locate inefficiently used containers [6] or low-utility temporary data structures [5], or identify temporary objects created within loops [1]. All of these analyses target stylized sources of churn; they can therefore yield a significant number of false negatives. Other sources of churn that do not follow such patterns are also missed by these approaches. Moreover, even for the problems they report, it is difficult for a developer to decide on a suitable course of action. First, the automated techniques can return false positives because some of the identified problems can be too difficult or even impossible to fix by a developer (e.g., when the problem is entirely contained within third-party code). More importantly, while automated techniques can often locate symptoms of object churn, they cannot easily identify the real source of the problem. Determining the root cause for these temporaries is a difficult task that involves following the flow of temporaries across many method and framework boundaries.

It is thus necessary to develop new tools and techniques to help developers to understand the behavior of their applications with respect to object churn. Such techniques would allow programmers to quickly investigate reports produced by other tools, but more importantly to identify sources of churn not found by existing automatic techniques. This paper, therefore, proposes a new visual, interactive approach...
to help developers to quickly and easily identify sources of churn in their applications.

Our visualization approach is designed to allow exploration of the program behavior in terms of temporary objects in a simple and intuitive way, all while ensuring that it remains scalable. Because churn often crosses method and even framework boundaries, our approach makes it possible to track temporary objects from their creation to their use. Also, since churn often results from multiple methods within a given region collaborating to build complex temporary data structures, our approach represents program regions explicitly using the Sunburst visual metaphor [10], a popular visual representation of tree structures. This allows users to quickly identify regions that exhibit suspicious behavior. Finally, the tool provides different views on the data, and at different granularity levels, in order to facilitate the exploration process without overwhelming the user with a large amount of information. Transitions between these different views are made as seamless and natural as possible in order to lower the required cognitive effort. We use natural abstractions that are easily perceived (e.g., the size and color of a visual entity) to draw the attention of the user to important parts of the system under consideration. Our prototype implementation, Vasco, also supports a number of secondary features that are necessary to support the exploration process that users typically follow.

Contributions. The key contribution of this paper is a visualization technique specifically designed to explore and identify sources of object churn in framework-intensive applications. This technique relies on the Sunburst metaphor to overlay blended escape analysis results on execution trace data. The prototype implementation supports an interactive exploration process through carefully designed mechanisms grounded in preattentive perception principles [11]. These principles help to minimize the cognitive effort required to explore the behavior of real-world applications. Initial empirical studies on real, framework-intensive applications demonstrate the usefulness of our approach by identifying previously unreported sources of churn.

II. BACKGROUND

Our visualization approach simultaneously displays two kinds of information: a representation of the interprocedural control-flow in the application, as well as information regarding temporary objects. This information is obtained from existing analysis tools based on concrete execution traces. In this section, we describe how the information used as input to our visualization is computed.

A. Gathering Execution Traces

Our approach extracts control-flow information from execution traces. Conceptually, a dynamic trace consists of a tree where each node represents a single invocation of a method, and edges represent caller-callee relationships between invocations. Call trees are typically too large to be visualized directly. We therefore use Calling Context Trees (CCTs) [12], a more concise, although less precise, representation of calls. CCTs represent invocations of a method resulting from the same sequence of calls (i.e., the same call stack at the time of invocation) using a single node. Note that while the original CCT definition allows for cycles in the presence of recursive calls, we represent recursive invocations explicitly in our CCTs.

We extract CCTs from traces collected using the Jinsight tool [13]. For each CCT node, we additionally record the types of allocated objects and the number of allocations for each type. Since a single CCT node can represent many concrete invocations, the allocations are aggregated over all invocations for a given node.

B. Escape Analysis

Automatically identifying temporary objects requires computing bounds on their dynamic lifetimes. For this purpose, we use escape analysis. Escape analysis computes bounds on the dynamic scopes of static objects (e.g., allocation sites). For example, it can be used to determine that an object cannot be reachable after the execution of a given method.

Our escape analysis implementation is based on the algorithm of Choi et al. [14]. This analysis subsumes reference (points-to) analysis, which determines for every reference in the program the set of (static) objects it can point to. It additionally computes an escape state for each static object at every method in the program. Each object can either globally escape (e.g., through a static field), escape locally through the arguments or return value of a method, or be captured (i.e., become unreachable after the execution of a given method). Objects can have different escape states at different points in the program, but eventually they either escape globally or become captured. Captured objects are potential temporaries because their lifetimes are constrained by the method(s) that capture them. We say that such objects are local to the region that captures them, i.e. the region of execution defined by the capturing method and all of its callees. For each node in the CCT, the escape analysis algorithm generates a connection graph, a representation of the relationships between objects reachable at this node, along with their respective escape states.

Escape analysis requires an existing call graph to explore the program. We use a blended analysis [7] variant of the analysis: rather than computing the call graph from the analyzed code, we use the dynamic CCT to avoid analyzing unexecuted code. This reduces the amount of analyzed code, and greatly improves scalability compared to a traditional whole-program analysis. Note that our analysis results are sound for a specific execution.

The results from the blended escape analysis are finally postprocessed using the Churni tool (as in [15]) to render
them more easily amenable to a user’s inspection. Churn first maps dynamic instances from the dynamic CCT to the equivalent static objects in the blended analysis results. The mapped instances are then propagated along the edges in the CCT. Static objects with no associated dynamic instances can then be removed from the final, reduced connection graphs. The output from Churni is then used to augment the original CCTs with information regarding capturing nodes for each dynamic object. This new CCT is packaged along with the generated reduced connection graphs and used as input to our visualization tool.

III. A Visual Environment for Churn Analysis

In order to help users identify sources of object churn in framework-intensive applications, we have developed Vasco, an interactive visualization tool that uses as input dynamic CCTs as well as the reduced connection graphs generated by Elude [16]. Vasco allows users to explore the behavior of their programs as intuitively as possible while ensuring the scalability of the approach. In this section, we discuss the key components of our visualization approach.

A. Design Objectives

We have identified three guiding principles for the design of our visualization tool: scalability, low cognitive effort, and smooth transitions and interactions.

Scalable abstractions. The large size and complexity of framework-intensive applications are a challenge for visualization tools. For instance, they commonly invoke hundreds of thousands of methods to accomplish a single task. The visual metaphors must therefore use the available space as efficiently as possible while remaining intuitive and easily understandable by users. Furthermore, it is often desirable to reduce the size of the data being visualized through abstraction. While abstractions can be greatly beneficial, proper care must be taken to achieve a balance between scalability and precision.

Intuitive abstractions. An efficient visualization approach should not impose a large cognitive effort on the users. Therefore, it is necessary to use an intuitive metaphor (e.g., one that employs familiar shapes). Visual attributes used by the metaphor should also clearly and intuitively represent their associated data. For example, shape, size and color are easy to compare between different entities, and should be favored in the design. We base our design decisions in this area on the work by Healey and Enns [11] on the preattentive process, which describes the way humans can quickly categorize images into regions and properties. The number of visual attributes used is also important. Too many attributes will overwhelm users and hinder their ability to identify entities. Too few attributes will not provide the necessary information to accomplish the task at hand and thus greatly limit the usefulness of the approach.

Smooth transitions and interactions. Visual transitions and interactions with the tool can be very disruptive and cause users to lose their focus. It is critical to minimize their impact. For example, information that is often required should be placed as close as possible to the main view without cluttering it. Data that is consulted less frequently should be placed further away. Placing information where it is easily accessible can reduce or eliminate the need for users to shift their focus away from the visualization during interactions. Determining which information is relevant to the visualization task is therefore an important consideration. Furthermore, in most cases, a visualization tool should ensure that the user can understand the context associated with the current view. For instance, previous interactions should leave behind visual artifacts in order to free users from having to remember them. Also, these artifacts can serve as visual anchors while transitioning from one state to another.

B. Representing CCTs Visually

Stasko et al. proposed the Sunburst visual metaphor [10] to represent tree structures. Sunburst uses a radial layout consisting of a series of concentric circles to represent the levels in the tree. The root of the tree is placed at the center of the visualization, and the leaves on the outer circles. Each node in the tree is represented as an arc in the Sunburst, i.e., a portion of one of the concentric circles. Each arc is defined by its level from the center and the portion of the full circle that it spans. The angle spanned by an arc typically depends on properties of the data being visualized. The children of a given tree node are placed in the same sector as their parent, and collectively span most of the same number of degrees as their parent.

Sunburst makes the parent-child relationship between nodes intuitive while not requiring edges to be represented explicitly. It also offers a strong visual representation for the program region induced by a CCT node. This metaphor thus achieves an excellent utilization of the available screen real estate without sacrificing ease of understanding.

C. Metrics

The Sunburst metaphor can represent properties of each node using two visual attributes of each arc: its color and the angle it spans. Each of these attributes is mapped to a metric computed for the node. We allow the user to interactively select which metric map to each of the visual attributes.

We have defined four metrics that are useful for identifying object churn:

Allocated types: the number of distinct types allocated by a set of invocations. This metric measures the complexity of temporary data structures.

1The remainder of the discussion will use the term node to refer to a CCT node and arc to denote its corresponding visual representation in the Sunburst.

1
Allocated objects: the total number of object instances allocated by a set of invocations. This metric measures the dynamic importance of object creation sites in the program. Captured objects: the total number of object instances captured by a set of invocations. This metric measures the importance of capturing points in the program in terms of the number of temporaries they explain. Churn: the number of object instances allocated by a set of invocations that are eventually captured. This metric is used to identify the object creation sites that are more likely to be associated with object churn.

Each metric can be computed on the set of invocations corresponding to a single CCT node or its induced region. Each CCT node correspondingly has two values for each metric: an individual value and a region value. Each value is mapped to a distinct visual attribute of the arc in the Sunburst. The individual value is represented using the color of the arc, while the region value is mapped to the angle spanned by the arc.

The full range of values for the metric mapped to the color attribute is visually represented using a color gradient. By default, this gradient ranges from blue to red for the lowest value to highest values. The user can also select other colors for the gradient. When looking for nodes that allocate a large number of objects, for example, the user may want to quickly locate nodes responsible for at least a certain number of objects. The user can therefore manually adjust the threshold used as the maximum value.

The angle spanned by an arc is assigned according to the ratio of its metric value to its parent’s value. This mapping ensures that the angle spanned by a node is at least as large as the angle spanned collectively by its children, since region metric values are cumulative. Note that in cases where the parent node has a non-zero individual metric value, the number of degrees available to its children is less than the number of degrees spanned by the parent node.

To improve the scalability of the approach and reduce the clutter due to unimportant child arcs, we also define a threshold for the number of degrees necessary for an arc to be represented in the visualization. Arcs smaller than this threshold (currently set to 4 degrees) are grouped and represented collectively rather than individually. Children of the nodes in the group, if any, are not represented. The rationale behind this choice is that nodes with low metric values are most likely unimportant for the analysis task, so no precision is lost by not representing them.

Figure 1 shows\(^2\) an example of metrics being mapped to visual attributes in the Sunburst. We show both the individual and region values for each node. Note that the values are displayed on the arcs for illustration purposes only; in practice, arcs are not labelled.

In the figure, node A has three children. The 360° available to A are therefore divided proportionally among them. For example, node B has a region metric value of 27 while its parent has a total value of 57. B will therefore use 27/57 of the space (171°). For the color metric, blue represents an individual metric value of 0, and red corresponds to the maximum value, 18. Nodes A, C and D have an individual value of 0, and are therefore shown in blue. B has a value of 18, and is shown in bright red. The remaining nodes have intermediate values, and are represented with colors that are a mix of blue and red. For example, node G has a value of 9, corresponding to equal parts blue and red (i.e., purple).

This representation allows a large amount of data to be displayed in a scalable way. For example, Figure 2 area ① shows the structure and metric mapping for a larger CCT.

D. Interactions

The typical usage scenario in our approach is an iterative process in which the user first attempts to locate the most significant source of churn. Once this source has been identified, the user can remove it from the view and repeat the process until no sources of churn remain. A typical visualization session will therefore alternate between inquiry actions, in which the user attempts to gain a better understanding of the application, and filtering actions, where the user removes certain elements from the view. In the remainder of this section, we describe the interactions supported by Vasco.

Navigation and filtering. At the start of the visualization process, our tool displays the entire CCT, thus giving the user a high-level overview of the entire execution. The user can then navigate and filter this view during exploration through various possible interactions. For instance, the user can select a new root for the visualization (i.e., display only a subtree of the CCT) by left-clicking one of the nodes. All displayed data is then recalculated to reflect this change. Note that changing the root could cause previously grouped nodes (i.e., nodes that were not displayed because of their low importance) to be expanded in the updated view, if their relative contribution becomes sufficiently large.

\(^2\)Figures should be viewed in color to better understand their perceptual values. They can be accessed at geodes.iro.umontreal.ca/projects/vasco/icsm12.
Additionally, a desaturated representation of the view prior to the selection is added to the context sidebar (Figure 2 area 2). At any point in time, the user can see the most-recent contexts in the sidebar, and return to any of them by simply clicking them. This visual representation of contexts allows the user to effortlessly navigate within complex structures. The visual cues from the context representation also help users to reason about the currently visible portion of the data in its context, since the selected method at each context is highlighted to show the progression from the global view to the current one.

Navigation is one of the forms of filtering supported by our tool, since it effectively limits the view to nodes located within a single selected region. The tool also supports the opposite operation, namely hiding a single region within the current view. Users can close nodes to reduce their usage of screen real estate. A closed node remains partially visible in the current view, as illustrated in Figure 3. Leaving a visual cue indicating the presence of closed nodes is important for two reasons: it allows users to revert the close operation (e.g., clicking a closed node expands it in our implementation), but most importantly it helps users to quickly adapt to the resulting change in layout. The closed node effectively acts as an anchor point during the transition. In our implementation, closed nodes are therefore represented with a distinct color (black by default) and span as many levels in the Sunburst as their entire regions. While a closed node is not excluded from metric computations, most of the space previously occupied by this node is made available to its siblings. This could cause previously grouped nodes to be expanded.

While collapsing entire regions of the view into a single arc is a valuable interaction during the exploration process, users sometimes prefer to completely remove some regions from view. For example, after an iteration that results in the identification of a region responsible for churn, it may be beneficial to hide it from view before starting the next iteration. Vasco supports two additional filters for this purpose: removing a selected node from the view and removing all invocations of the selected method. Once removed, a node is excluded from all metric computations and is also completely invisible in the view.

**Contextual information.** While the Sunburst metaphor is often able to quickly point a user toward program regions of interest, more detailed information is necessary in order to understand how these regions use temporaries. This information should be readily available to the user, but should not clutter the main visualization. Therefore, our tool uses an information panel (Figure 2 area 3) located just above the main view (Figure 2 area 1) to display the most useful data about the entity currently under the mouse pointer. For clarity, the arc under the cursor is also highlighted. The information panel displays the name of the method under the cursor, as well as information regarding its allocated objects, allocated types and captured objects. The panel also includes a nested panel that graphically shows the number of types of either allocated objects or captured objects at the node under the cursor, depending on the metric that is currently selected. The information panel makes many common inquiries fast and non-disruptive, allowing the user to remain focused on the visualization. For example, a common interaction is to investigate a long sequence of calls from allocation to capture. By simply moving the mouse along the arcs composing the sequence, a user can quickly obtain the required information. Detailed information about the behavior of a method (such as the reduced connection graph generated by Churni) is also available via a contextual menu. This information is made available on demand to avoid unnecessarily cluttering the interface. In practice, we found it to be most valuable when a user has already identified a potential source of churn. In order to allow users to investigate groups of methods collectively, Vasco supports multiple selections. When multiple nodes are selected, the information panel displays aggregated information.

**Metric selection.** Metrics provide different views of the data. Selecting the metric to assign to either the color or size of arcs is a common interaction during the analysis phase. The user can modify the mapping of the metrics directly from the menu bar (Figure 2 area 4). Metrics used for the color attribute and the arc size can be selected independently. For the color mapping, the value corresponding to the maximal color can also be selected by the user, either by entering a value directly (area 4) or by interacting with the
color gradient (area 6). The selection is reflected in real time on the visualization, thus allowing the user to fine-tune the threshold with very little effort.

Search. It is often useful to identify program locations related to the one under consideration. For example, a user who has identified a potential source of churn may be interested to know how many other regions exhibit the same suspicious behavior. Our tool, therefore, supports inquiries regarding a selection. For example, using a popup menu, the user can select all other invocations of the same method in the CCT. Note that this constitutes a multiple selection. The information panel then displays aggregated information about all selected invocations. This feature allows a user to evaluate the impact of removing a source of churn prior to carrying out the necessary transformations.

The tool can also help users track captured objects from their creation to their capturing node, one of the most important activities when trying to identify sources of churn. For instance, a user will often first identify regions of the program responsible for the creation of a large number of potential temporaries. However, in order to determine whether these temporaries can indeed be eliminated, one must also understand why they were created in the first place. This involves identifying the capturing location for these objects, and understanding the flow of objects between the creation node and the capturing node. To facilitate this process, the tool visually identifies all the nodes that capture objects allocated from a selected node. The user can then investigate the intermediate nodes on the path from creation to capture, for example, by examining the detailed connection graphs for these nodes in Vasco. The reverse operation, i.e., identifying nodes allocating objects captured at a selected node, is also supported in the same way, thus allowing users to start their investigation from the capturing nodes rather than the allocating ones.

Finally, users can enter a search string that will cause all matching nodes to be selected (Figure 2 area 5). Nodes are matched based on class and method names.

IV. Case Studies

A. Experimental Setting

In order to evaluate the effectiveness of our approach, we used three different applications of varying complexity.

Trade6. Trade is a financial transaction benchmark for the WebSphere platform. We used version 6.0.1 of the Trade benchmark running on WebSphere 6.0.0.1 and DB2 8.2.0. We traced a single transaction after warming-up the benchmark with 5000 iterations of the built-in scenario. We configured Trade to use a direct connection to the database. As this software is proprietary, it is not possible to disclose more information about it.

Jazz server. Jazz is a collaborative client-server development platform. Clients are integrated development environments (IDEs) based on Eclipse. The server is based on WebSphere. We traced a single query to retrieve the work items assigned to a single developer in version 1.0M5a of the Jazz server.

Commercial document management server (CDMS). This application is a commercial platform that provides management and workflow for documents. We traced a part of the test suite responsible for storing 10 documents in the database. As this software is proprietary, it is not possible to disclose more information about it.

Despite tracing a single operation for each benchmark, Table I shows the complexity of the work performed by each application. Columns 2 and 3 show the total number of distinct types that were allocated and the total number of instances (i.e., observed object allocations), respectively. The last four columns show the total number of distinct methods executed, the total number of method invocations, the number of nodes in the CCT, and the maximum depth of the call stack during execution. Note that the number of nodes in the CCT is much lower in all three cases than the number of calls (i.e., the number of nodes in the full call tree), ranging from 19% of the calls for Jazz to only 1.6% for CDMS. This significant reduction in the number of nodes plays a key role in rendering the visualization approach scalable.

We have used our approach to analyze the behavior each benchmark with respect to temporaries. We describe the results of our evaluation in the next subsections. These case studies are meant to demonstrate the usefulness of the approach at identifying the most significant sources of churn, rather than a thorough and complete analysis of the churn found in the applications we studied.

B. Trade

Figure 4a shows the initial view presented to the user for the Trade benchmark. This view maps the churn metric to the color of the arcs, and the number of allocations to their size. Therefore, the brightest red arcs are the most likely sources of temporaries. Trade is a benchmark that has been optimized through numerous revisions, and as a result does not feature many optimization opportunities. There are, nonetheless, two bright-red arcs in the initial view. Placing the mouse pointer over the arc labeled 1 reveals that it corresponds to the method DecimalFormat.parse, and that it creates 216 objects over 54 different invocations. Most of these objects are used for bookkeeping and are difficult to optimize. More interestingly, the arc labeled 2 corresponds to the constructor of the Calendar class,
which creates 99 objects over 9 invocations. Most of these objects are arrays that store the state of the calendar objects. Selecting the “Show capturing invocations” option for this node indicates that these objects are captured in the `DateSerializer.getValueAsString` method (fuchsia arc in Figure 4b). In previous work, we have identified this method as responsible for creating 8 `HashSet` objects, all of which are also captured by `getItemType`. A quick inspection of the call stack from `getItemType` to `HashMap.addEntry` reveals that `getItemType` calls the `getAllModelURIs` method in the same class, which creates and populates a fresh `HashSet` object on every allocation. In Java, the `HashSet` class is implemented using a `HashMap` internally, which explains the observed behavior. In this trace, the set returned by this method always contains the same elements, and could therefore be easily cached. To evaluate the impact that such a caching strategy would have on the program, we select all invocations of the `getAllModelURIs` method and inspect the statistics for the selection (Figure 5c). This operation reveals that this method is responsible for the creation of 3197 objects in total, all of them temporary. Caching would therefore be particularly fruitful in this case. Also note that the method responsible for the churn in this case, `getAllModelURIs`, is neither the allocating nor the capturing method. This illustrates the need to explore the call stack in order to properly diagnose sources of churn.

D. CDMS

Figure 6a shows the initial view for the CDMS benchmark. All the most significant sources of churn (bright-red nodes in the figure) are string-related. To better understand the behavior of this application, we change to the captures view, as illustrated in Figure 6b. To clearly indicate the differences in the metrics used, we also change the gradient to go from yellow (few captures) to green (many captures). This view indicates that the 1120 allocated strings are captured by the `PropertiesImpl.isPropertyPresent` method (arc labeled 1 in Figure 6b). The region induced by an invocation of this method includes a second source of churn (arc labeled 2 in the figure) accounting for as many `StringBuilder` instances resulting from calls to `String.toLowerCase`. The `isPropertyPresent` method is called a total of 5370 times in the trace. Its top four calling contexts alone account for a total of 8960 temporaries. Clearly, thousands of temporaries are created to ensure that property descriptors do not contain uppercase letters. It would be much more efficient to enforce this invariant by sanitizing all inputs only once before their use in the entire system. Note that it is not uncommon for large applications, and web applications in particular, to create a large number of string instances. Vasco allows the user to quickly locate suspicious strings in the context of their capturing regions.
Figure 5. JAZZ

Figure 6. CDMS

Figure 6c shows the same view with all previously identified sources of churn removed (by using the “Select other invocations of this method” option followed by “Remove selection”). The threshold for the metric has also been adjusted to show more capturing locations. In the figure, the mouse pointer is located on the SecurityServer.hasConnectAccess method, which captures a large number of temporaries. As the information panel indicates Figure 6d, this node captures 300 temporaries of 8 different types. Its region captures an additional 700 temporaries. Interestingly, an inspection of the immediate sibling of this node in the view reveals exactly the same local capture behavior. Selecting the “Show allocating methods” option for either of these nodes reveals another clue. The hasConnectAccess method was invoked 20 times in the trace, and it created 20 SecurityDescriptor structures (one per invocation). Each security descriptor is created and initialized by deserializing a stream of bytes, resulting in 1000 temporaries being created in the process.

A manual inspection of the code reveals that each SecurityDescriptor instance is created for a specific Id instance. Caching SecurityDescriptors within their associated Id objects would therefore prevent most of these temporaries from being repeatedly created. It is also worth noting that hasConnectAccess is guarded by a global flag that determines whether security checks are enabled. During performance testing, it is conceivable that such checks could be disabled, thus leading to very different performance characteristics.

V. RELATED WORK

Visualizing execution data. A large body of existing work proposes visualization techniques to help with program
understanding tasks. Such approaches typically fall within two main categories: visualizing the program structure and visualizing execution traces. Approaches based on representing the program structure (e.g., [17]) typically use a 3D city metaphor, where classes are represented visually as buildings. Calls between classes are represented as links in the 3D space between the buildings. In contrast to these approaches, our aim is to make regions of the execution explicit and intuitive. Naive trace visualization techniques rarely scale to the size the applications considered here. To address this problem, some authors propose trace reduction strategies (e.g., [18]) while others target more scalable visualization metaphors, either by utilizing screen real estate efficiently (e.g., [19]) or by providing multiple granularity levels (e.g., [20]). In contrast, Vasco is not concerned with understanding a sequence of events, but rather displays a higher-level representation of the behavior of a program. Program Explorer [21] supports two tree views: objects and calls. While this approach is conceptually similar to ours, it suffers from a lack of scalability due to a much more detailed representation of the behavior.

Recently, some approaches have focused on analysing large CCTs. Moret et al. [22] successfully used a Sunburst representation of CCTs to identify performance bottlenecks in Java applications. Their approach allows a user to quickly identify “hot” methods visually, for example, by visualizing the number of executed bytecodes. This approach has been incorporated into the Senseo infrastructure [23]. Similarly, FlyBy ([24]) is a performance diagnosis tool that allows users to flag inefficient operations during an execution and cause a CCT to be generated. This CCT is then explored visually to identify the performance bottleneck. FlyBy is implemented using the Trevis infrastructure from the same authors, a tree visualization toolkit that supports Sunburst as well as other metaphors. Vasco shares the high-level goal of explaining dynamic behavior of large software systems with both approaches, and supports many of the same interactions. However, both our data representation and our interactions are much richer than those used by other tools (e.g., navigation based on the escape analysis data, visual context). Also, common techniques used for scalability are not always applicable in the context of churn analysis. For example, limiting the depth of the tree can be problematic since temporary objects are often allocated in or near the leaves. However, Vasco could borrow some features from Trevis, such as the ability to compare CCTs visually.

Fisher II et al. [15] have also studied visual representations of churn-related data. They proposed Hi-C, a tool for visualizing data generated by Elude. While the authors have shown that Hi-C was able to help detect some sources of churn in framework-intensive applications, Vasco attempts to address some of its shortcomings. First, Hi-C uses a simple tree-based visual metaphor, making program regions difficult to identify. In contrast, Vasco’s use of Sunburst makes these regions explicit and intuitive. Second, to scale to large CCTs, Hi-C hides methods that do not allocate or capture objects. This makes analyzing the results difficult because (i) tracking object across methods is more difficult when the sequence is incomplete, and (ii) the real source of temporaries can often be hidden using this strategy. Vasco also hides some unimportant methods, but our strategy never hinders the exploration process. Third, Hi-C presents the raw analysis data to the user, which is often too low-level or too complex to analyze manually. In contrast, Vasco favors interactions based on the task to be executed rather than the available data.

**Bloat.** Mitchell et al. have studied many aspects of bloat, including data structure usage [25] and ownership [26]. Chis et al. [2] have characterized several causes of memory bloat in industrial software. Zhao et al. [27] have studied the impact of excessive object allocations on the scalability of a number of Java benchmarks to a high number of CPU cores. Shankar et al. [8] have developed a technique to remove some sources of churn automatically during JIT compilation. Our approach is complementary to theirs, in that it allows developers to identify and fix sources of churn that are not automatically optimizable.

Xu et al. have proposed several approaches to automatically identify various kinds of bloat (e.g., inefficiently-used containers [6] and low-utility data structures [5]). Bhattacharya et al. [1] have proposed an approach that identifies temporary objects created within loops. In contrast, our approach aims to help developers to find sources of churn that cannot be located by automatic techniques, and to better diagnose the symptoms identified by these techniques.

**VI. CONCLUSIONS & FUTURE WORK**

Bloat, and in particular object churn, is a widespread performance problem in framework-intensive applications. We presented an interactive visualization approach that helps developers to locate and understand significant sources of churn in their applications. Our approach leverages dynamic data generated by an existing blended escape analysis tool. We implemented this approach in Vasco, a new scalable visualization platform. Vasco relies on known preattentive perception principles in order to reduce the cognitive effort required to perform the visual analysis task.

We demonstrated the usefulness of our approach by applying it to three framework-intensive systems of various complexities, including a commercial application. We identified previously known as well as unreported sources of churn quickly and easily. Our results show that this approach is a promising way to explore the dynamic behavior of complex, framework-intensive systems.

We plan to extend this work in three ways. First, we want to support a richer set of interactions, such as the ability to search for invocations allocating or capturing the same types of instances. We also plan to integrate Vasco
in Eclipse. Second, we are developing a profiler that will remove some limitations of the existing Jinsight profiling tool. For instance, it will support profiling much longer program executions. Third, we plan to extend our approach to other key components of bloat and, more generally, other problems and program understanding tasks that require a precise exploration of the behavior. For example, Vasco could be used to track the flow of data within a complex application.

REFERENCES


