IDE Integration for Execution Mining Data

Alexander Bradley

CPSC 538W Final Project Report
University of British Columbia

April 30, 2010

1 Introduction

Execution mining systems such as Tralfamadore [6, 7] record a complete trace of a virtual machine’s execution. Analysis of this data can help users to answer many potentially interesting questions about their code: What functions call a given function, and how often do they do so? What values does a given function return in practice, and with what frequency? What values are usually passed into a function as parameters? What is the data flow history of a given object? (That is, what are the call stacks through which the object typically passes before reaching an execution point of interest?) There are many other potential questions of interest, but this project will focus on the ones just mentioned. We will consider the problem of integrating answers to these questions into the display of source code to which they apply.

This problem has already been considered in the Tralfamadore project. Tralfamadore has been applied to the Linux kernel, and a modified version of the Mercurial web source code browser has been used to show some prototypical visualizations of execution mining data. This viewer can display histograms of typical values or graphs of data flow history for variables, show typical execution paths through functions with marginal coloured arcs, and summarize callers and outgoing calls for a function. However, the current prototype lacks many features typical of modern IDEs (e.g., full syntax highlighting, expanding/collapsing code blocks, and easy navigation to files, function definitions, and macro/variable definitions anywhere in the codebase.) The goal of our project is to integrate visualization of execution mining data with an IDE—specifically, the Eclipse C/C++ Development Tools (CDT)—thus making Tralfamadore’s execution mining data more accessible to developers. We have created an Eclipse plugin that allows users to find answers to all of the questions mentioned in the first paragraph directly from their development workbench.

2 Related Work

The Eclipse Profiler project [1] (now defunct) provided a variety of features for displaying profiling data collected from the execution of Java programs, including call graphs, tables showing the number of calls to and time spent in each class/package in the system, thread-specific call trees, and heap usage graphs. The current Eclipse Test and Performance Tools Platform (TPTP) project [9] provides similar features. The direction of this work is very similar to ours; indeed, it might be useful to leverage TPTP for future versions or variants of our project.

Edwards [2] proposed the idea of “example centric” programming environments, in which code execution results are tightly integrated into a programming UI, creating an “example-enlightened” editor in which developers can immediately see the concrete effects of their work. His system is designed for much tighter coupling between source editing and production of execution results than ours, but the philosophy behind his user interface has much in common with our own.
Seyster [8] has surveyed a wide variety of techniques for execution and trace visualization, including call graph displays, resource usage plotting, sequence diagrams, call matrices, and data structure displays. Some of the visualization techniques he references might be interesting to incorporate into future work on this project.

3 Approach

3.1 Feature Overview

We have realized our project as an Eclipse plugin that augments the Eclipse CDT source code editor with visualizations of execution mining data. The CDT can already be used for full-featured browsing of the Linux kernel source code, as shown by Kågström [5] and others. Our plugin replicates several existing features of Tralfamadore’s web interface and adds some improvements over the existing functionality.

The Tralfamadore web interface currently allows the user to select any parameter of a function and view a histogram of the values that parameter takes at runtime. Our interface provides similar functionality: users can hover over any parameter and a histogram of its values will be shown in a hover popup (see Figure 3.) Furthermore, as shown in Figure 1, our interface allows users to hover over a function name and see a similar histogram of its return values (if it is not `void`.)

![Figure 1: Histogram popup showing return values and callers for current_fs_time function.](image)

The Tralfamadore web interface also shows the calling function to which a function returns through an overlay box at the line where the return occurs. Our plugin provides similar functionality by showing a list of a function’s callers in the hover popup that appears when a user hovers over the function’s name. (See Figure 1.) When a user clicks on any function name in this popup, the Eclipse CDT code indexes are searched for that function and the source code for that function is immediately shown in an editor by the IDE. The plugin also adds a ruler to the editor’s left margin (see Figure 2) which shows a green box next to every function that is called at least once in the trace. The number of calls to the function is shown in the box.

Furthermore, the Tralfamadore web interface allows users to select any pointer parameter of a function and obtain a data flow tree for that object. Our interface allows users to obtain a similar graph; when they select a function parameter and choose “Show Parameter Dataflow” in a right-click menu, the tree is constructed in a separate view. Compared to the Tralfamadore web interface tree, our data flow view has some additional features: users can perform a textual search which highlights matching function names in
3.2 Implementation Details

Our plugin acts as a client for the existing analysis web services provided by the Tralfamadore project. (The user can specify the URL of the Tralfamadore server for an Eclipse project through the project preferences.) We have modified the existing di_function network service, which originally provided data values only for function parameters, to also provide function return values, names of function callers, and counts of callers for sets of functions. The request protocol for our modified di_function is shown in Table 1.

<table>
<thead>
<tr>
<th>Request</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>P ⟨filename⟩ ⟨line number⟩ ⟨parameter name⟩</td>
<td>Parameter values for function at given location</td>
</tr>
<tr>
<td>R ⟨filename⟩ ⟨line number⟩</td>
<td>Return values for function at given location</td>
</tr>
<tr>
<td>C ⟨filename⟩ ⟨line number⟩</td>
<td>Names of callers for function at given location</td>
</tr>
<tr>
<td>A ⟨filename⟩ ⟨line numbers…⟩</td>
<td>Numbers of callers for functions at given lines in file</td>
</tr>
</tbody>
</table>

Table 1: Request protocol for modified di_function.

Our plugin uses abstract syntax trees computed by the CDT to determine the locations of function names and parameters in source files. These locations, once computed, are cached for performance, as are the function caller counts downloaded from the server. (Histogram and data flow queries are not yet cached.) Our histograms are created by generating HTML which is rendered in Eclipse browser controls. Navigation from caller histograms and data flow trees to function locations is facilitated by CDT code indexing. Our data flow view is constructed by parsing JSON data from a Tralfamadore web service using the org.json library [4] and drawing trees using the Eclipse Zest framework [3].
4 Usage Scenarios

Lefebvre et al. [7] provide a number of application scenarios to demonstrate how the Tralfamadore web interface can be useful in answering developer questions. We will demonstrate the usefulness of our interface by showing how it supports use cases similar to those in section 5.1 of their work. In both our work and theirs, the visualization interface to Tralfamadore assists a user in understanding the execution of the Linux kernel. When we ran our scenarios, the server used was a Xen virtual machine (2.5GHz CPU, 727MB RAM, Ubuntu 9.04) provided by the NSS Lab at UBC. Our client machine was an IBM ThinkPad X60 (2GHz CPU, 2GB RAM, Ubuntu 9.10) running Eclipse 3.5.1 with CDT version 5.1.0. The slowest network link between the machines was a residential ADSL connection. A relatively small Tralfamadore trace (163MB) was used.

4.1 Argument value distribution

Let us suppose that a user is interested in the size of network buffers (skbuffs) allocated by the kernel. If she knows that these network buffers are allocated by the _alloc_skb function in skbuff.c, she can start her search for information by navigating to that function. A green box in the vertical ruler to the left of the editor informs her that there have been calls to that function in the Tralfamadore trace, and the number in the box indicates to her that the function has been called 46 times in total. When she hovers over the size parameter, a popup appears within a second (see Figure 3) showing a histogram of the values taken by that parameter. She can see that the most commonly requested buffer size is 272 bytes, with buffers of 1532 bytes requested almost as frequently.

![Figure 3: Histogram popup for size parameter to _alloc_skb function, showing allocated network buffer sizes.](image)

4.2 Dynamic data flow

A user may be interested in viewing the data flow for network packets processed by the kernel. He might choose to generate a data flow graph for the skb parameter of the netif_rx function in dev.c. He can do this by right-clicking on skb in the editor and selecting “Show Parameter Dataflow” from the context menu. A data flow tree will be assembled in a new view. The tree will be computed server-side by a Tralfamadore analysis script, and graphical nodes will be added in the Eclipse view as soon as corresponding data is
received from the server. For this function, the tree begins to form within a few seconds, but downloading and displaying the entire tree takes about 110 seconds. The user can select whether to display the tree at “full size”, in which case it will not fit on a single screen (see Figure 4), or to display it so that all nodes fit on a single screen (see Figure 5).

Figure 4: Section of full-size view of data flow tree for skb parameter of netif_rx function. Functions with names containing the string tcp are highlighted in green.

Figure 5: Data flow tree for skb parameter of netif_rx function, sized to fit screen. Functions with names containing the string udp are highlighted in green.

If the user is interested in seeing where particular functions occur in the tree, he can use the search box (in the top right-hand corner of the view) to filter the tree by function name. In Figure 4, he has searched for the string tcp, and can see that many paths in the tree lead through TCP processing functions. However, by searching for udp he can see that there was 1 UDP packet on one path through the tree (Figure 5).
5 Limitations and Future Work

5.1 Performance and Robustness

To make our system acceptable for production use, greater consideration will need to be given to performance optimization and fault tolerance. Our hover boxes and marginal editor rulers currently require requests to the Tralfamadore server every time they are generated and do not render until those requests are complete. We have found this approach acceptable for testing our proof-of-concept prototype with a small trace, but it obviously lacks scalability. Rendering the hover popup immediately and constructing the histogram in real time as the results stream from the server, in the fashion of the current Tralfamadore web interface, would be a more scalable approach. Furthermore, more testing and error handling will be needed to ensure that server-side failures or delays will not cause hangs or crashes in UI rendering for the hover popups and rulers.

5.2 Enhancements to Current UI

Our histograms are currently sorted in decreasing order of frequency, and this order cannot be changed. To increase the flexibility of our histogram interface, it would be desirable to give users the ability to sort the histogram in ascending or descending order of frequency or value, to match the features of the web interface.

Caller information and return value histograms currently appear in the same popup when the user hovers over a function name, which can result in a large, hard-to-read popup. It might be better to separate these interface features into separate popups or to provide controls in the popup which enable the user to alternate between the two. It might also be useful to provide users with a right-click context menu option which would enable them to see the caller, return value and parameter value histograms all at once in a separate view.

Clicking on caller function names in the caller histogram currently shows the top of the function in an editor. It would be very useful if the exact call site inside the function could be displayed instead. Similarly, it would be useful if it was possible to navigate from nodes in the data flow tree (or edges, as in the Eclipse Profiler [1]) to exact call sites where an object was passed from one function to another.

5.3 Handling Code Edits

One significant limitation of our current approach is that while we are adding features that are tightly integrated into the display of source code text in an editor, these features break down rapidly if the user edits the code. Since the server-side code for querying Tralfamadore traces currently relies on file names and line numbers to specify code locations, our client-side interface for querying properties of a function would fail as soon as the user changed the line at which the function was located. That technical problem could be mitigated to some extent by changing the server-side code to accept queries based on function names, rather than line numbers. However, a more fundamental general challenge remains: when integrating execution traces with the user’s code editing process, the execution pattern reflected by the trace will originate from a version of the code that lags the user’s current edited version to some degree. Closing that gap is difficult, since users edit code relatively rapidly and compiling code and regenerating traces takes much longer (especially if the code is on the scale of the Linux kernel.) Therefore, in developing interfaces like ours, it will be important to give the user some indication of how outdated trace information is and to make judgments as to when the user has changed the source code so much that the trace information is obsolete and should not be shown.
5.4 Showing Execution Behaviour Inside Functions

Our current interface is focused on calls and data flow between functions. It would be interesting to extend our interface to show which paths execution takes at the statement level inside a function. We could show typical paths through a function using marginal arcs (as in the current web interface), or by providing a list of typical paths taken at the top of the function and highlighting the executed code when the user selects one such path. It would also be interesting to show which outgoing calls were actually made on those paths.

A further idea would be to use the background colour of each source code statement, or a marginal ruler, to display other properties of interest. For instance, showing how frequently a statement is executed or how much CPU time or wall time it takes on average would help the developer to gain awareness of hotspots in the code. CPU time, wall time and stack usage could even be displayed with marginal vertical “sparkline” graphs in the style of Tufte [10].

6 Conclusion

We have presented a prototypical user interface for integrating execution mining data into an IDE. Our interface, realized as an Eclipse CDT plugin, allows users to access trace analyses directly from their source code editor. A marginal ruler provides them with visual hints as to which functions have trace data available and shows the level of activity for each function. They can view histograms of the values passed as function parameters, the values returned by a function, and the names of callers to a function. They can also obtain data flow graphs for objects passed into functions. From any node in the data flow graph and any function name in a caller histogram, they can instantly navigate to the function referenced. Through usage scenarios similar to those used in previous work on Tralfamadore, we demonstrate that our interface provides powerful and useful features which can answer a number of developer questions. We believe that further integration of execution mining features into IDEs is a promising direction for future research.

References


