Techniques for Visualizing Software Execution

A Research Proficiency Exam Presented

By

Justin Seyster

Technical Report FSL-08-03

March 6, 2008

Abstract of the RPE

Techniques for Visualizing Software Execution

by

Justin Seyster
Stony Brook University

March 2008

Instrumenting or profiling a program's execution yields enormous amounts of data that the programmer can use to diagnose bugs and performance problems or even explore how a program works. However, the raw data is often too vast for any programmer to analyze unaided. The goal of execution visualization is to present that data visually in a way that allows the viewer to better understand a program's behavior and to draw conclusions about that behavior. This paper surveys tools that visualize program executions by graphing data from execution traces, showing the steps of the program's execution, or displaying the contents of runtime data structures. We evaluate the techniques these tools use based on how much data they show with static displays, animations, and interactive queries and how clearly that data communicates information the user is looking for. We also examine the availability of these techniques in production software visualization packages. As part of our survey, we present our own system that visualizes output from the Memcov memory profiler to aid in identifying sources of leaky or wasteful memory allocations. Finally, we explore the types of data generated by program executions and discuss displays that are effective for each.

This work was partially made possible thanks to a Computer Systems Research NSF award (CNS-0509230) and an NSF CAREER award in the Next Generation Software program (EIA-0133589).

Contents

List of Figures

Chapter 1

Introduction

A software execution visualization seeks to be a window into what a program does while it runs. The user of a visualization wants the ability to examine a program's execution simply by looking at the visualization. Beyond that ability, many software execution visualizations let the user zoom and manipulate the view to explore the visual features of an execution at different levels of detail.

The first step in visualizing an execution is collecting the data for display. Software instrumentation, the most common method of collecting data from a program's execution, is an active area of research. Knowing how instrumentation works is not important to understanding the key motivator of visualization tools, however. That motivator is that an instrumented program can produce immense data sets. Consider a program instrumented to output some result for every function call. How long would the program need to run before it outputs millions of these results? Probably only than a few minutes.

In its raw form, all this data is useless; no developer would search through it all. Instead, most developers would instinctively construct a graph to compare results from different functions. Such a graph is a simple visualization. This paper surveys a range of tools that solve the same problem, from those as simple as the previously mentioned graph to much more sophisticated techniques.

The visualization tools discussed here fall into four different categories.

1. Execution visualizations show the individual steps that a program takes as it runs.
2. Data structure visualizations show the layout and contents of a program's data structures.
3. Object-oriented program visualizations show the relationships between program classes and objects during an object-oriented program's execution.
4. Trace visualizations show measurements of other program properties, such as processor utilization or memory access time.

These categories include both online and offline visualization tools. Offline tools collect data during a program's run and provide a view of the data once the run finishes. Online tools display the data during program execution. One advantage to an online view is that the user can quickly see how a program action affects the program state by watching the visualization.

One online visualization is Memcov Viz, our tool for visualizing dynamic memory events in real time. Memcov Viz emphasizes this real time approach, allowing the user to see immediately accesses to all of a program's memory areas. We conclude the survey of visualization tools in Chapter 2 by presenting Memcov Viz.

The rest of this paper is organized as follows. Chapter 2 discusses visualization tools in the four categories. Chapter 3 gives a theoretical overview of the types of data a software execution visualization can present. Chapter 4 suggests avenues for research in software visualization.

Chapter 2

Visualization Tools

In this chapter, we evaluate visualization tools from four different categories. We classify them according to the different ways they present data. Execution visualizations focus on presenting data with animations that illustrate program actions, an approach well suited to understanding algorithm implementations. Data structure visualizations, on the other hand, are typically static but show a more detailed view of a program's data. Object-oriented program visualizations provide views of runtime data that help reveal the static and dynamic structure of a program. Trace visualizations, the last type of visualization we discuss, capture and visualize various other statistics that can indirectly provide a programmer with insight into a program's inner workings.

2.1 Execution Visualization

The purpose of an execution visualization is to illustrate the steps of a program's execution visually. Each data entity, perhaps an array element or an object pointer, appears in the visualization, and each program action is an animated interaction between those objects. The result is like opening the program's hood and watching its previously hidden internal workings.

Understanding these workings is the first step in debugging an application. Usually, that means stepping through the application in a debugger. Because it shows a program's every action, a debugger with a source level view is like an execution visualizer. Source code is not a visual medium, however, and a debugger only shows half the picture. Each statement shows a program action, but the data entities never appear in the debugger's source view.

The 1981 short movie *Sorting out Sorting* [2] is an early visualization of sorting algorithms. Vertical bars with varying heights star as array elements, switching places with each other to demonstrate the swaps that the sorting algorithm chooses until the bars stand in height order. What if we could have a view as simple and powerful as *Sorting out Sorting* for any program? Rather then staring at source code as a program executes, we could watch the program's internal entities, searching for bugs or performance problems. This goal motivates our discussion of execution visualizations.

2.1.1 XTANGO and POLKA

In 1990, Stasko implemented the XTANGO system for animating algorithms written in C [45]. The programmer adds animation events directly into the program by way of calls to XTANGO functions. These animation events move and resize animated shapes that represent program objects. The system emphasizes using smooth animation to keep the visualization clear. Shapes that change position in the animation move along a path rather than snapping from place to place. Figure 2.1 is one frame from an XTANGO animation.

Though XTANGO cannot automatically create animations for arbitrary programs, as this is not generally possible, designing an XTANGO animation is simple enough that it is potentially useful as a tool for debugging algorithm implementations.

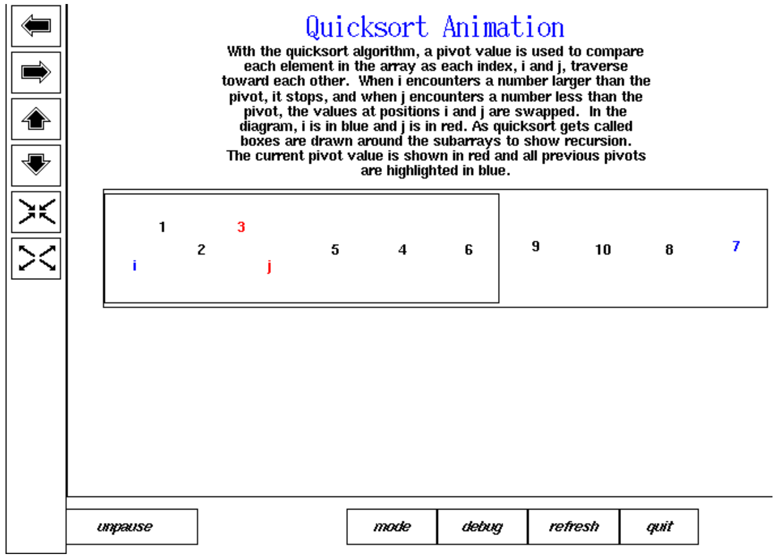


Figure 2.1: Animated XTango view of a quicksort implementation swapping two elements.

The 1993 POLKA system, implemented by Stasko et al., expands XTANGO's animation approach to visualize parallel programs [47]. POLKA relies on an "animation choreographer" that lets the user manually control synchronization in the animation by adjusting the ordering and speed of animation events in separate threads. POLKA itself independently handles animation events that occur at overlapping times, as is necessary to visualize concurrent operations. In addition to two-dimensional output like XTANGO, POLKA supports three-dimensional animations.

2.1.2 Lens

Also in 1993, Mukherjea and Stasko implemented the Lens system [30], designed to create visualizations like XTANGO's without requiring changes to a program's source code. The animation designer selects lines of code to spawn shapes and then places those shapes graphically. The designer controls a shape's animation by adding animation events that move the shape to a position based on the value of a program variable. These events can similarly adjust a shape's size. Lens reads the values of program variables by running the program in the dbx debugger [49].

2.1.3 HotWire

In 1994, Laffra et al. implemented HotWire, a visual debugger for object-oriented C++ and Smalltalk programs [25]. HotWire includes a constraint language for displaying shapes for each program object. These shapes animate during program execution.

Laffra et al. present a case study of debugging a performance bug in a word processor. Their example draws a box in the execution view every time the word processor redraws a portion of its display. Each box has the same size and position as the redrawn region. Running the program with this visualization reveals which operations redraw unnecessarily large regions of the display.

HotWire also includes a display that shows a box for every object in the program. Each box is color coded by how often its instance methods are called. Instances of the same class appear in the same row, forming a view that is also useful as a bar chart.

2.1.4 3D-PP

In 2004, Okamura et al. implemented the "three-dimensional visual programming system" 3D-PP [36]. Rather than visualizing an existing program, 3D-PP allows a programmer to create the program visually.

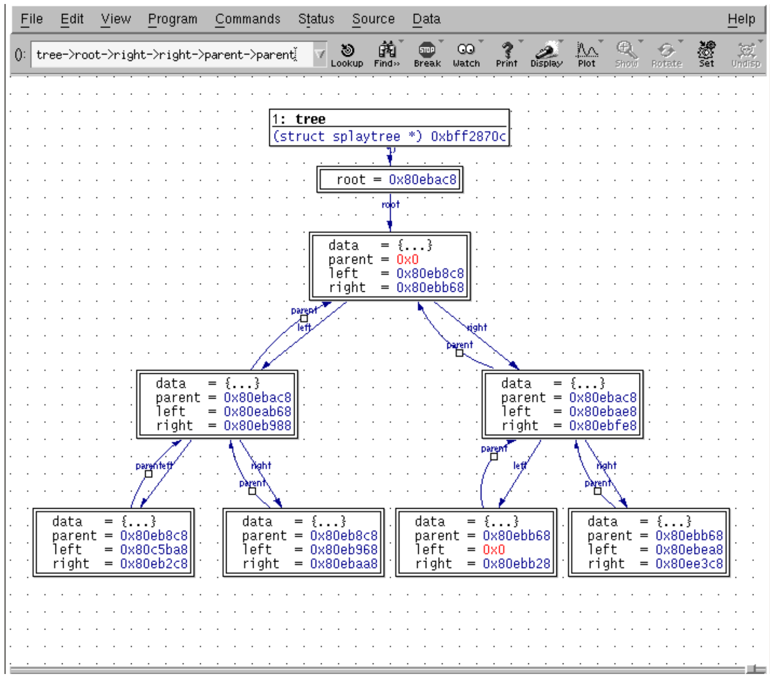


Figure 2.2: Box-and-arrow diagram of a binary tree data structure from a DDD debugging session.

The programmer places operands (input numbers) and operators (such as arithmetic and comparison operators) and then creates connections from operator outputs and operands to operator inputs.

When the user executes the program, the program itself animates to visualize its own execution. As each operator executes, its input operands move towards it until the operands and operator combine and turn into the result of the operation. Operators execute in this fashion until the last operator finishes, yielding the final output. The user can pause and rewind the animation.

2.2 Data Structure Visualization

While execution visualizations focus on the actions that a program takes, data structure visualizations focus on the data entities. Data structure visualizations are typically static, showing the state of a data structure at a particular time during program execution.

The most flexible and most common type is the box-and-arrow diagram, a view included in all of the tools in this section. However, the potential exists for as many kinds of data structure visualizations as there are data structures.

2.2.1 Incense

In 1983, Myers implemented Incense, a tool for showing abstract views of data structures during program execution [32]. Incense uses "Artists," which know how to read a data structure and then draw it.

One included Artist draws box-and-arrow diagrams of records. The box-and-arrow Artist draws a box for each record with that record's fields inside of it and an arrow for each pointer from one record to another. Figure 2.2 shows an example of a box-and-arrow diagram.

Incense users can write Artists for any kind of data structure. Another included Artist draws a tuple of $(hours, minutes, seconds)$ as a clock face. Myers also suggests potentially useful Artists for ring buffers and list iterators. A ring buffer Artist could draw the buffer as a circle to show its ring structure. An iterator Artist could draw the iterator variable as a progress bar to show how far it is through its iteration.

2.2.2 GDBX

In 1985, Baskerville implemented the GDBX frontend to the DBX debugger [3]. GDBX focuses on improving Incense's approach to box-and-arrow diagrams, rather than allowing for many specialized data views. One improvement is the "two-dimensional space allocation" mechanism for placing boxes in the visualization as first suggested in Myers [32]. GDBX can place more boxes on the screen than Incense's box-and-arrow Artist, and rather than shrinking boxes to fit more in a limited space, GDBX places data structures on a larger, scrollable canvas.

GDBX data structure visualizations are interactive. The user can hide or show boxes and move them around to represent any structure. During program execution, the user can move an arrow to a different box to actually change the underlying structure. GDBX automatically updates pointers to stay consistent with the moved arrow.

2.2.3 DDD

In 1996, Zeller et al. implemented DDD [52], a free-software frontend for GDB [13] and dbx [49]. DDD supports interactive box-and-arrow diagrams as shown in Figure 2.2.

More recent versions of DDD have the ability to plot one- and two-dimensional arrays with gnuplot [11]. While the target program is paused, the user selects an array for display, and DDD creates a bar chart for a one-dimensional array or a surface plot for a two-dimensional array. The user can also add a breakpoint that automatically graphs an array, creating an animated plot that shows the array changing over time.

2.2.4 Other Debuggers

A number of other debuggers also implement similar visualization features. The 1987 VIPS by Isoda et al. is an ADA debugger that includes Incense-like facilities for box-and-arrow diagrams and custom data views [21]. The commercial SoftBench [18] and CodeCenter [9] debuggers also support box-and-arrow diagrams.

2.3 Object-Oriented Program Visualization

Object-oriented programming focuses on encapsulating data in objects that interact by passing messages, or method calls. It is natural that this view of programming leads to visualizations that seek to show these message-passing interactions.

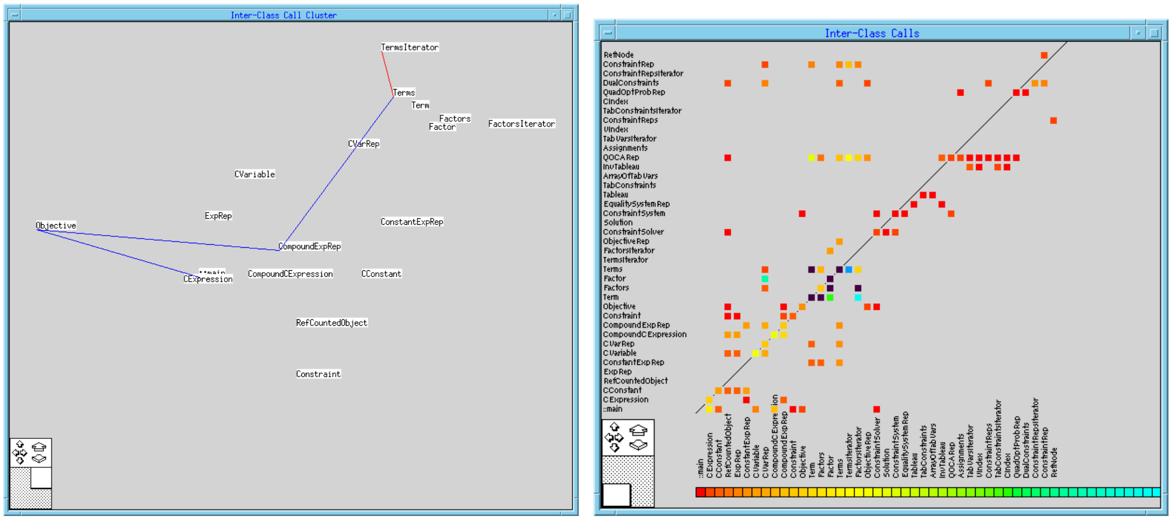
Some of the visualizations presented here use the class as the primary visual entity, aggregating the actions of all objects that belong to the same class. Other visualizations show individual objects for a finer-grained view of the program.

2.3.1 Call Clusters and Call Matrices

In 1993 De Pauw et al. implemented a visualization tool for object-oriented programs written in C++ that focuses on interactions between classes [37]. The system's four animated views show the state of the program while it runs or during playback of a recorded execution. The user can pause execution to examine each of the views.

An "inter-class call cluster" view, shown in Figure 2.3(a), displays a label for each class in the program. The view places classes closer together the more they call each other, so that tightly-bound groups of classes cluster together. The cluster view shows the current call stack as a path of lines through the class labels. The path starts from the class at the bottom of the stack (i.e., the class that contains the function at the bottom of the stack) and passes through each of the classes above it, in order, until it reaches the class at the top of the stack. The idea of a call stack trajectory is one that appears in several later visualizations.

An "inter-class call matrix" view, like the one in Figure 2.3(b), shows the number of calls between pairs of classes up until the current time. This view uses the same principle as the message-queue matrix in ParaGraph [16] discussed in Section 2.4.1. A program with n classes has an $n \times n$ grid. Box (i, j) is colored



(a) Inter-class call cluster

(b) Inter-class call matrix

Figure 2.3: Screen captures taken from De Pauw et al. [37]. In the call cluster, the blue line shows the current state of the call stack. The red segment of the line is at the top of the stack. The call matrix shows which pairs of classes that call each other's methods.

by the number of calls that class i made to class j so far. Dark horizontal lines indicate classes that make calls to many other classes, and dark vertical lines indicate classes that many other classes call.

A second matrix view shows the number of *allocations* classes make. The view is the same as the call matrix, except that box (i, j) shows the number of times that class i allocated an object of class j. This view is designed to show the source of heavy memory usage.

A bar chart shows the number of instances of each class.

2.3.2 Program Explorer

In 1995, Lange et al. implemented Program Explorer, a tool to visualize object-oriented executions that focuses on object instances rather than classes [26]. To cope with the large number of objects a program may have, the user can choose to display only objects of a certain class (and its subclasses).

An "object graph" view shows the objects in the program and their creation relationships. Edges exist from every object to the objects it created. A similar call graph view instead shows method call relationships, as seen in Figure 2.4. An edge (labeled by method name) exists for every object that calls another object's method. The user can also choose to display only certain methods.

An "invocation chart" view, also shown in Figure 2.4, shows the sequence of method calls. Objects appear in the chart as a row of vertical bars. Events appear in sequence going down the chart. The beginning of a bar (the top of the bar) represents object creation, and the end of the bar represents object destruction. A horizontal arrow (labeled by method name) from the calling object's bar to the target object's bar represents a method call. This view is similar to a UML sequence diagram.

2.3.3 Module Views

In 1998, Walker et al. developed a visualization tool for object-oriented programs [50]. After the target program runs, their tool divides its execution into intervals of time and draws a *cel* for each interval.

The user defines several program modules, assigning program classes to those modules, and each cel contains a graph of those modules. For every function call across modules during a cel's interval, there is a directed edge in the graph. A curved path goes through the modules in the call stack at the end of the interval, like the call stack trajectory in the "inter-class cluster view" from De Pauw et al. [37].

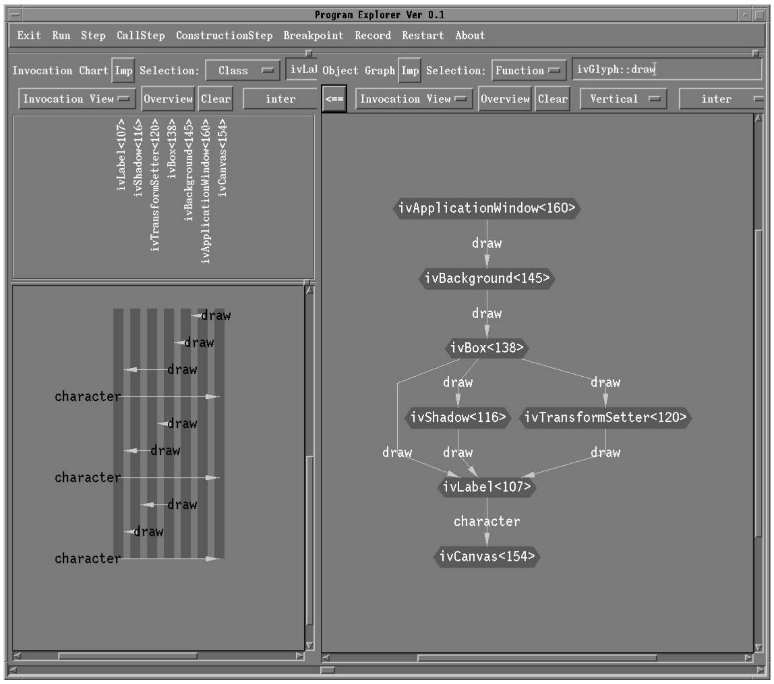


Figure 2.4: A screenshot of Program Explorer taken from Lange et al. [26]. The invocation chart on the left shows a sequence of method calls between five objects, and the view on the right shows those same method calls in an object graph.

Every node contains a histogram of the number of objects in the module (i.e., instantiated from a class in that module) by age.

2.3.4 Jinsight

The IBM Jinsight project implements several views for examining object-oriented program executions.

Execution patterns. In 1998, De Pauw et al. implemented a visualization tool with an "execution patterns" view for showing a sequence of method calls [38]. The execution pattern view is modeled after UML sequence diagrams. Vertical bars represent method calls, but the bars are not placed on per-object tracks like in a sequence diagram. While such tracks show communication between small numbers of objects very clearly, they are impractical for large systems.

Each bar is labeled with an object number and is colored to show the object's class. The method calls that a method makes all appear immediately to its right, each connected by an arrow labeled with the method name. A method's distance to the right indicates how deep it is in the stack.

The Jinsight implementation includes several means to manage the complexity of an execution pattern from a long execution, as shown in Figure 2.5. When a sequence of calls repeats itself, Jinsight draws the sequence only once, along with the number of repetitions. Additionally, the user can click any method bar to collapse it, hiding all of its method calls. Rather than collapsing an entire subtree, the user can show just a summary of the subtree with a list of its classes.

The user can zoom out to see more of the view. When it is zoomed out far enough, Jinsight turns off labels and arrows. Only small colored boxes remain, providing a bar chart of stack depth over time.

Object references. In 2001, De Pauw et al. implemented a visualization of object references designed to help find wasted memory in Java programs [39]. Though Java programs are garbage collected, they can still leak memory if they retain references to objects that are no longer necessary.

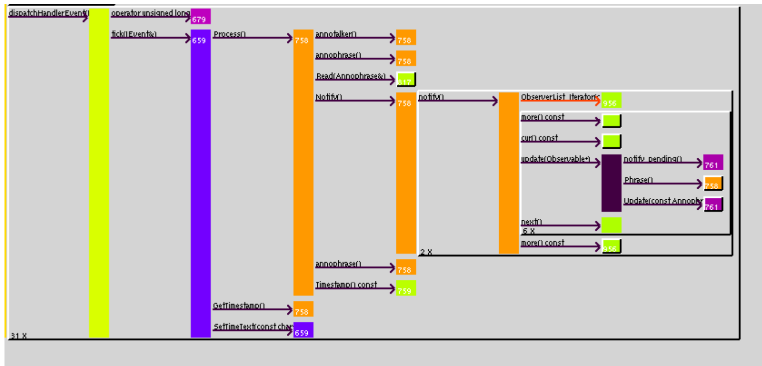


Figure 2.5: A screenshot of an execution pattern in Jinsight taken from Lange et al. [38]. Jinsight combined several repeated sequences in this view. Method bars with a bevel are collapsed. The user can click them to see more of the tree.

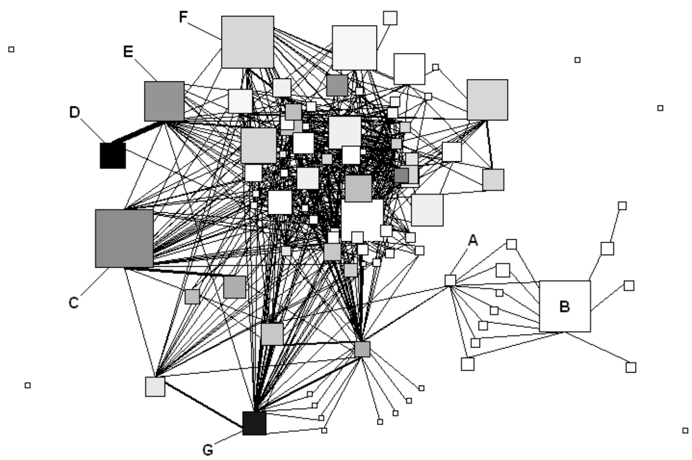


Figure 2.6: A "communication interaction view" taken from Bertuli et al. [4]. The larger rectangles represent classes with a lot of methods. Classes that are closer together are more tightly-bound. That is, they call each other's methods very frequently.

The visualization seeks to show patterns of memory references with a tree view. The user chooses a set of objects for the roots of the tree. There is one node for each class of objects in the set, and each of these nodes forms the root of a tree. A node's children consist of all the objects that hold references to one of the node's objects, again grouped into sets of objects in the same class.

By choosing objects that remain in memory even after their useful lifespans for the starting set, the user can find what objects are holding unnecessary references to those objects.

2.3.5 Run-time Polymetric Views

In 2003, Bertuli et al. implemented a tool to visualize object-oriented program executions using "run-time polymetric views" [4]. Classes in this kind of view are rectangles. A rectangle's color and each of its dimensions represent statistical properties of the class. The rectangles also serve as nodes in a graph.

As an example, the "instance usage overview" shows the class rectangles in an inheritance tree, with an edge from each superclass to its subclass. A rectangle's width indicates how many instances of that class the program created during its execution, and its height indicates how many methods the class has (excluding methods that are never called). A rectangle's color represents the total number of calls to methods from the class.

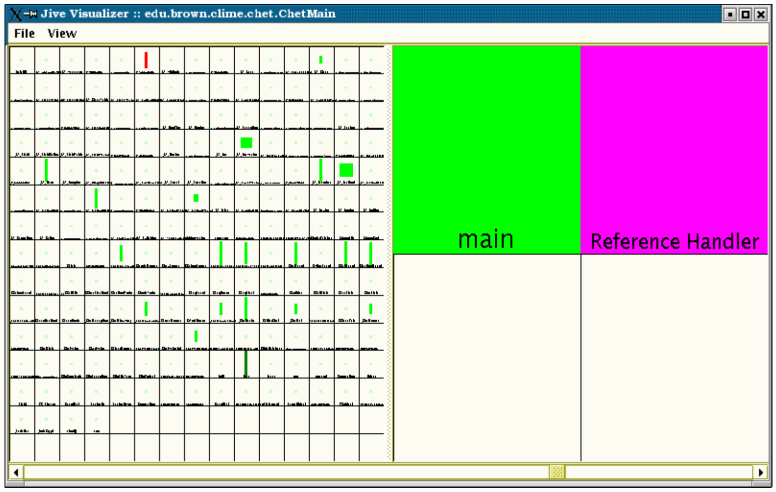


Figure 2.7: A screenshot of Jive taken from Reiss [41]. The width of each colored box in the left panel indicates how many methods the class has, and the height represents how many allocations the class made.

The "communication interaction view," seen in Figure 2.6, shows method invocations between classes. The height and color of rectangles in this view use the same metric as in the instance usage overview. The height is always the same as the width, so all the nodes are squares. There is an edge between two rectangles for each class that invokes a method in another class. The more frequently a pair of classes invoke each other's methods, the heavier their edge is weighted. Edge weights do not appear on the graph, but the graph embedding draws nodes with highly weighted edges closer together. This behavior forces groups of tightly-bound classes to cluster together.

2.3.6 Jive

Also in 2003, Reiss implemented Jive, a real-time visualization for object-oriented programs [41]. Figure 2.7 shows an example Jive session. Like a runtime polymetric view [4], the view has a box for each class. The boxes are arranged in a grid, however, and their positions convey no information. Each box has five ways of displaying a value: width, height, hue, saturation, and brightness. The saturation indicates whether the program recently called any of the class' methods. Other values include how many instances of the class exist and how many new objects the class created.

Each running thread also has a box. Thread boxes are colored coded by thread state. The dimensions of the box indicate how much time the thread has spent so far in its state.

All the boxes update in real time as the target program executes.

2.3.7 TraceVis

In 2006, Deelen implemented TraceVis, a tool for visualizing executions of object-oriented Java programs [10]. Figure 2.8 shows an example TraceVis session. TraceVis' main view is a class communication view. Every class is a node in the view, and a directed edge exists between two nodes if one class makes method calls to the other during execution. Instead of arrows, TraceVis uses a gradient to show an edge's direction.

The user can drag a timeline view to change the class communication view. Nodes change in size to show a user-selected property, such as the number of calls made to the class so far. Edges change in thickness to show the number of calls made along each edge so far. TraceVis highlights nodes in red that are on the call stack at the selected time and traces the path from the bottom to the top of the call stack in red, creating a call stack trajectory like the one in De Pauw et al. [37].

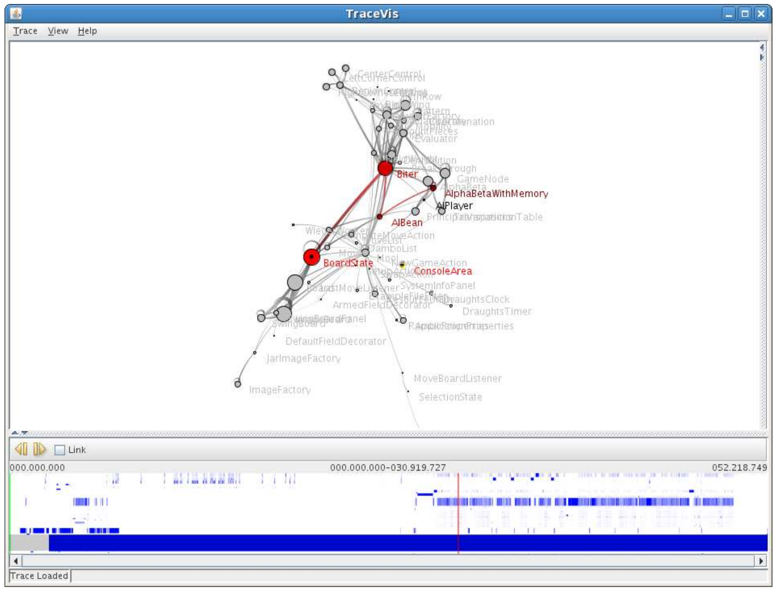


Figure 2.8: A screenshot of the TraceVis class communication and timeline views taken from Deelen [10]. There are two active threads at the selected time. The `BoardState` *class is at the top of one thread's call stack, and the* `ConsoleAnim` *class is at the top of the other thread's call stack.*

The timeline itself has a horizontal strip for every class. The strip changes color across its length to indicate what percent of time it is "active" (that is, execution is in one of that class' methods). The timeline is designed to look like a UML sequence diagram.

2.4 Trace Visualization

There is more to a program execution than program actions and data structures. Profilers can trace myriad behind-the-scenes statistics, like processor utilization, memory access times, or program speed. Amidst that trace data are valuable clues for the programmer seeking to tune performance and track down subtle bugs. A trace visualization can be the best way to find those clues.

We further categorize trace visualizations into offline and online tools. A programmer using an offline tool first runs the target program to collect trace data and then views that data with the visualization tool. Some visualizations, however, allow the programmer to view trace data in real time, while interacting with the target program.

2.4.1 Offline Tools

ParaGraph

In 1991, Heath et al. implemented ParaGraph, a software package with a number of visualizations for trace data captured from executions of parallel programs [16]. Most of ParaGraph's visualizations are animated over time.

Some ParaGraph visualizations use a horizontal time axis. These views scroll to the left as time passes in the animation. Gantt chart views have a timeline for each processor that is color coded to show the processor's state at each time (Figure 2.10(a)). The state can be whether the processor is busy or waiting, or it can be the task that the processor is working on. For task views, the programmer has to add calls to the target program that indicate transitions from one task to another. Every Gantt chart has a corresponding stacked-bar chart view that shows the number of processors in each state at each point in time.

A "space-time" view (Figure 2.10(b)) is a Gantt chart augmented with messages between processors.

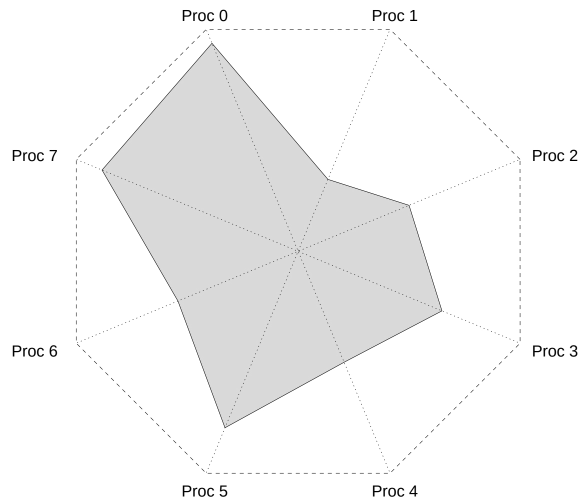


Figure 2.9: Example of a Kiviat diagram showing utilization for eight processors.

Each message is shown as a diagonal line from the sending processor's timeline to the receiving processor's timeline. On the time axis, the line starts at the time it is sent and ends at the time it is finally processed by the receiving processor. Line slopes roughly show how fast processors handle messages.

Other visualizations have no time axis, so they either show aggregate statistics from the entire program execution (and are therefore static), or show information from only a single instant in time. Each Gantt chart view has a corresponding static stacked-bar view that shows the total time each processor spent in each state.

An animated bar chart view shows the number of messages each processor has in its message queue. A more sophisticated message-queue visualization uses an $n \times n$ grid to show message-passing activity in an n-processor system. During the grid's animation, box (i, j) is filled in when there is a message from processor i in processor j's message queue. A third view draws each processor as a node in a graph, color coded to indicate it if is busy, and draws edges between nodes to show the source and destination of queued messages. Edges can be color coded to show whether they lie along a physical interconnect.

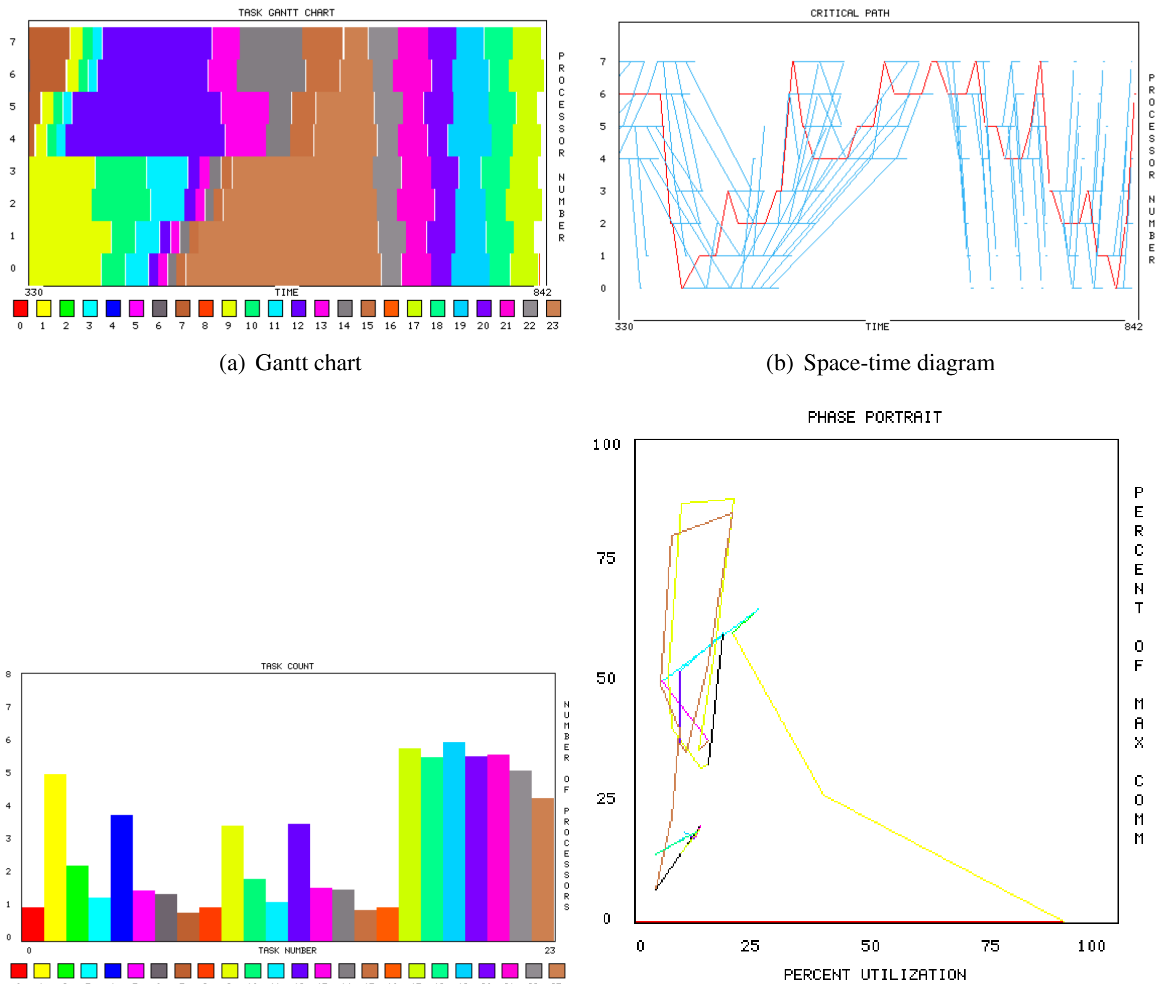
A Kiviat diagram shows the load average of all processors, like the example shown in Figure 2.9. The Kiviat diagram for an n-processor system is an n-gon with the distance of each vertex from the center of the diagram proportional to one processor's load average. When utilization is very high among all processors, the result is a regular polygon with a large area. When load is spread unevenly among processors, the Kiviat diagram appears lopsided.

A "phase portrait" shows the relationship between two variables as a point on a 2D plane (Figure 2.10(d)). ParaGraph's phase portrait has the number of busy processors on one axis and the number of messages waiting on message queues on the other axis. As these two values change during the course of the animation, the point moves around the view, tracing its path as it goes.

PARvis and VAMPIR

In 1994, Nagel et al. implemented PARvis, which also provides animated visualizations of parallel program execution traces [33]. The main PARvis view shows a rectangle for each processor divided into a top and a bottom half. The top half shows the processor's task at the current time in the animation, and the bottom half shows the percent of time the processor has spent on a user-selected task. A statistical view shows a pie chart or bar chart for each processor with the total time spent in all tasks.

PARvis also has a space-time view like ParaGraph's with interactive zooming so the user can see a detailed view of the tasks processed in a small period of time. Clicking on a line in the space-time view shows a description of the message that line represents.



(a) Gantt chart

(b) Space-time diagram

(c) Task count bar chart

(d) Phase portrait

Figure 2.10: Screen captures of a ParaGraph visualization. These four views are of a trace file for a program run on an eight processor system.

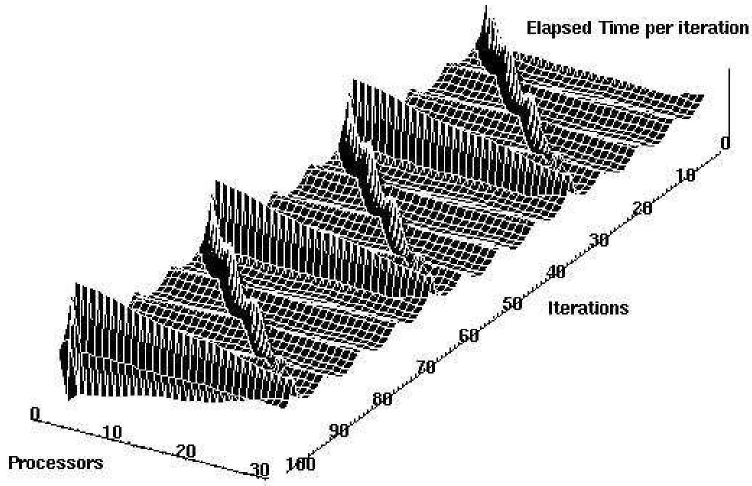


Figure 2.11: This do-loop-surface, taken from Naím et al. [35], shows the main loop of a matrix multiplication program. The three straight peeks show communication delays while one processor broadcasts a row to the other processors.

The 1996 VAMPIR, by Nagel et al., is based on PARvis [34]. It includes a grid view similar to the $n \times n$ grid in ParaGraph for visualizing message passing activity [16]. Box (i, j) is color coded to show the number of messages sent from processor i to processor j.

In 2005, Knüpfer et al. implemented Vampir NG, which stores traces in a "Complete Call Graph" data structure for faster drawing of timelines with more events [23].

Do-Loop-Surface

In 1996, Naím et al. presented the "Do-Loop-Surface" (DLS), a visualization of loop performance in a parallel program [35]. A DLS, like the one in Figure 2.11, shows the time that each loop iteration takes on each processor. The surface is a three-dimensional surface plot with the loop index on one axis, the processor number on another axis, and the total time that processor spent executing that loop iteration on the final axis.

SvPablo

In 1998, De Rose et al. implemented SvPablo, a parallel program analysis tool that overlays its visualization directly on program source code [43]. SvPablo displays the name of each program function with two color-coded boxes. One box is colored according the number of calls to the function, and the other is colored with the total time spent in the function. Similarly, SvPablo has a source display that annotates every line of code with eleven boxes. Each box is colored with a different parameter, such as the total time spent executing that line of code.

Other Parallel Program Visualizations

In 1992, Reed et al. implemented the Pablo performance analysis environment, which includes a visualization creation tool that is powerful enough to implement many of the parallel program visualizations described here [40]. It also includes "sonifications" which represent trace data with sound.

In 1994, Geist et al. implemented XPVM, which features a space-time view that updates in real time as the program executes. It can also generate space-time views after program execution from trace files [15].

In 1995, Miller et al. implemented Paradyn, which includes a visualization component with line plots and bar charts for processor utilization [29].

In 1996, Labarta et al. implemented the DiP parallel program development environment [24]. DiP's visualization tool, Paraver, displays space-time views from trace files. In 2002, Freitag et al. implemented parallel program tracing with a visualization component that shows Gantt charts [14].

GPV

In 2001, Weaver et al. implemented the Graphical Pipeline Viewer (GPV) to visualize trace files from processor simulations [51]. The visualization, inspired by instruction views used in architecture text books [17], shows a timeline for each instruction in the execution trace.

An instruction timeline is horizontal and changes color over its length as the instruction moves through different pipeline stages (or stalls). Time, measured in processor cycles, is on the x-axis.

Viewed from a distance, a line going through the timelines shows the rate of instruction execution. Because the timelines are stacked vertically, a steep line (i.e., close to vertical) indicates a high rate of Instructions Per Second retired (IPS). A shallower line indicates lower IPS.

GPV supports interactive zooming, so the user can zoom in to examine local phenomena or zoom out to see IPS variation throughout the program's execution. At close enough zoom, GPV prints the assembler instruction next to each timeline. The user can click on each timeline to reveal even more detailed information about the instruction.

The instruction view can show several program executions overlaid in the same view. Viewing traces together is a good way to compare the performance of different attempts at optimizing a program.

In addition to the instruction view, GPV provides a "resource view" that sits below the instruction view and shares its time axis. The resource view has lines plots of several variables that vary over time, such as power consumption.

Weaver et al. [51] includes a case study using GPV to improve an RC6 cipher implementation. Viewing the execution trace, they found a significant number of branch mispredictions in the cipher's inner loop. Removing an `if` statement and then unrolling the loop eliminated the mispredictions and improved execution time by 24%.

TraceVis

In 2004, Roberts implemented TraceVis, which features an instruction view like GPV's with additional features [42]. Figure 2.12 shows screen captures of TraceVis' instruction view. Note that, though this TraceVis shares its name with the TraceVis discussed in Section 2.3.7, the two tools are not related.

The user can search for instructions by program address and event (such as a cache miss). After searching, non-matching instructions appear dimmed, and the user can move the view through matching instructions. TraceVis can also "bookmark" instructions so that the user can return to them later.

Like GPV, TraceVis can graph statistics on a shared axis with the instruction. TraceVis shows statistics per instruction, however, rather than over time. As a result, the plot is a histogram that runs vertically along the y-axis of the instruction view.

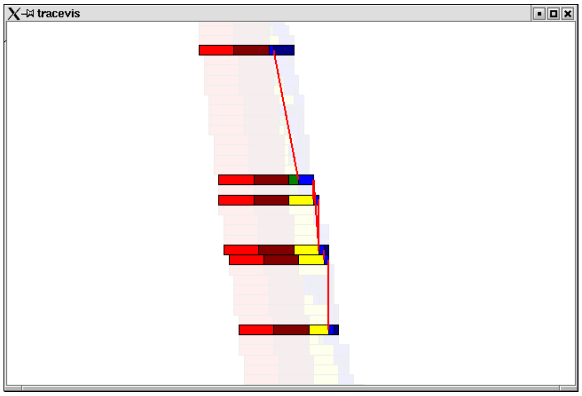
The user can also select a group of instructions by drawing a box around them in order to view the values of statistics for that region of instructions.

"Static code correlation" aids in analyzing the performance of specific regions of code. The user can assign colors to selected regions of high-level source code. TraceVis then colors each instruction according to the region of code it belongs to. When the view is zoomed out too far to color static regions separately, the static regions are shown as a statistics histogram.

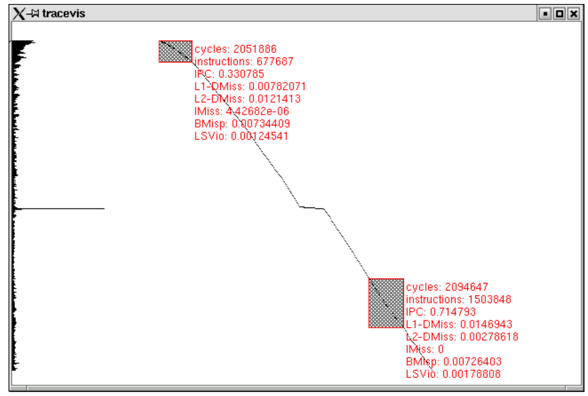
Finally, the user can view data dependencies for an instruction in the trace. TraceVis draws a line from an instruction to the phase in a previous instruction that it is dependent on and then draws that phase's data dependencies recursively.

Rivet

The Rivet system (Bosch et al.) provides a framework for computer system visualizations [6]. Though it is not publicly available, there are several visualization projects built on Rivet, which we describe next.



(a) Close-up view of individual instruction timelines with data dependencies



(b) Zoomed-out view of entire execution

Figure 2.12: Screen captures of TraceVis taken from Roberts [42]. Each timeline represents a single processor instruction. From far away, it is not possible to see individual instructions, but it is easy to see the execution speed (in IPS) at any time during the program execution. The zoomed-out view also has a histogram on the left and detailed statistics for two user-selected regions.

Hyperbolic Graph View. Among the facilities that Rivet provides is the hyperbolic graph view, implemented in 1998 by Munzner [31]. By laying out the graph in hyperbolic space and projecting the result onto a Euclidean sphere, it is possible to draw very large graphs that the user can move through interactively. This view is especially useful for graphs that have a meaningful spanning tree, like a call graph. Figure 2.13 shows a FORTRAN program call graph.

Seesoft. In 1992, Eick et al. implemented the Seesoft system for displaying statistics for lines of code [12]. Though the Seesoft project is not related to Rivet, it is included here because many Rivet visualizations use an implementation of this method provided by the Rivet framework.

Seesoft displays a zoomed-out view of the source code itself. Every line of code is drawn as a line one pixel in height. The text of each line is no longer discernible, but the view preserves indentation, providing a rough view of the program's control structures.

The system can color each line to show a property or statistic. Thousands of lines of code, all annotated by color, can fit on a regular display.

SUIF Explorer. In 1999, Liao et al. implemented the SUIF Explorer system for parallelizing programs [27]. The system uses the "bird's-eye view" of the code first described in Eick [12] to present the result of the first pass of parallelization. Loops are colored to show which loops were successfully parallelized. The user can also use filters to find loops that take a certain percent of execution time or are at a certain depth. These loops appear highlighted in the "bird's-eye view."

PipeCleaner. In 1999, Stolte et al. implemented the PipeCleaner system for visualizing application performance on superscalar processors [48]. PipeCleaner combines trace visualization and execution visualization.

PipeCleaner considers every cycle that does not retire the maximum possible number of instructions as a stall and chooses a reason for the stall (such as cache misses or data dependencies). A "strip chart" shows the causes of stalls over a program's execution.

The strip chart is a stacked histogram with processor cycles on the x-axis. The user can select a region of the histogram to see a zoomed-in view of the chart, along with a histogram for the types of instructions in the pipeline.

After choosing a region of time in the strip chart, the user can watch an animated execution visualization for that time period. Instructions appear as rectangles labeled with the assembly language text of the

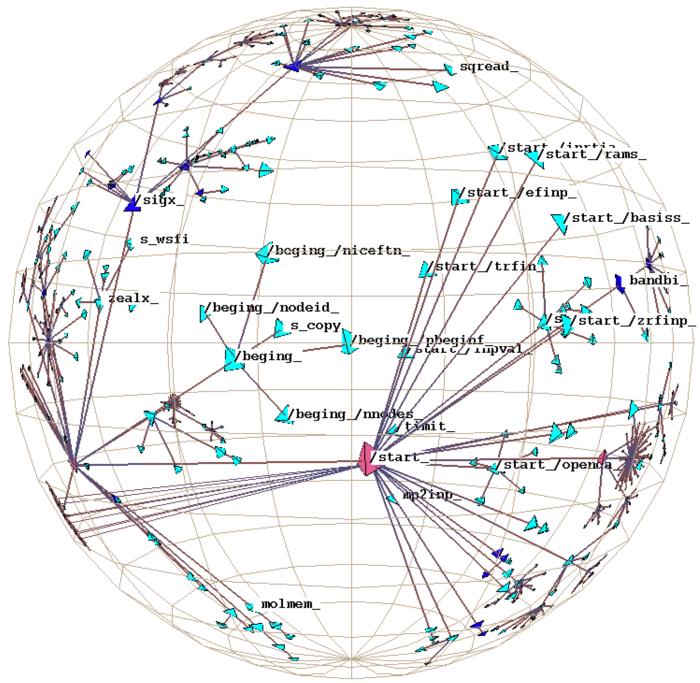


Figure 2.13: A hyperbolic space view of a FORTRAN program's call graph taken from Munzner [31].

instruction and colored by instruction type. Speculatively-executed instructions have a colored border. Lines between instructions represent data dependencies.

Different regions of the view represent pipeline stages and functional units. As each instruction changes stages, its rectangle moves smoothly between regions. The user can pause the animation, control its speed or watch it backwards.

Finally, PipeCleaner also provides a "bird's-eye view" of the source [12] that is color coded to indicate lines in the selected region of time and lines that are executing in the animated pipeline view.

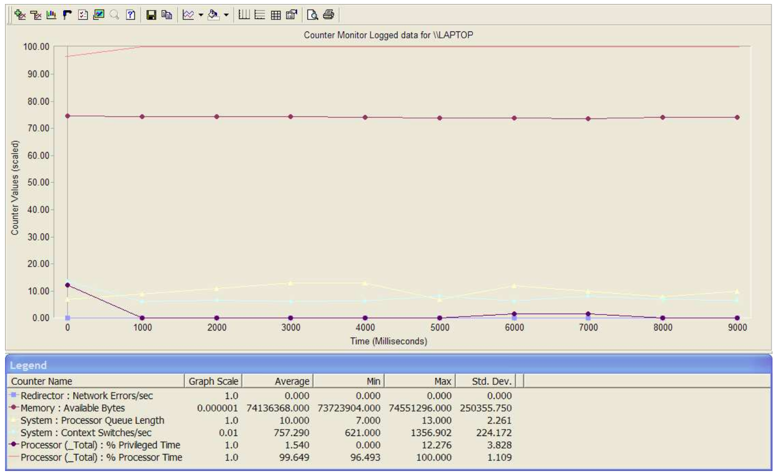
Argus Case Study. In 2000, Bosch et al. [5] presented a case study using Rivet visualization tools to find a performance problem in the Argus [19] parallel rendering library.

They first developed the MView visualization to examine the performance of memory accesses. MView includes:

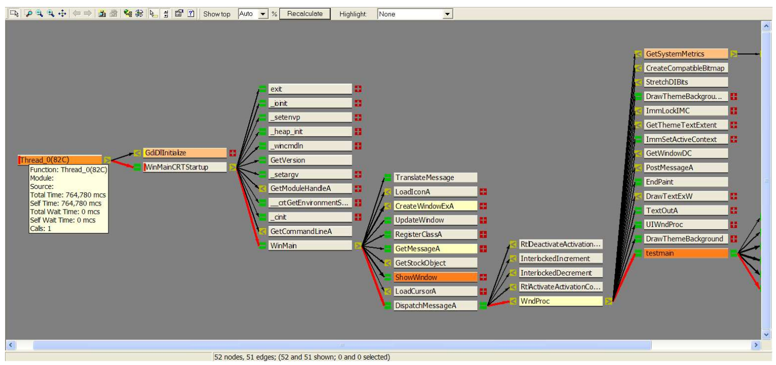
1. "bird's-eye" source views [12] that are color coded by how much time each line of code spent waiting for cache misses to resolve;
2. bar charts showing the time that the program spent waiting on cache misses for each region of memory, arranged by physical and virtual addresses; and
3. strip charts for each process, showing the time that the process spent running, waiting, and resolving cache misses.

The strip charts are zoomable and at close enough zoom become Gantt charts showing exactly how long a processes spent in each state.

Examining the strip charts revealed that, later on in the application's execution, processes were spending increasingly more time on the wait queue. Further analysis revealed that the wait time was the result of lock contention in the kernel. A simple workaround alleviated the contention and dramatically improved scalability.



(a) Line plot of statistics



(b) Call graph

Figure 2.14: Examples of VTune's two visualization views, a line plot of statistics over time, and a call graph view. The red edges in the call graph trace out a "critical path."

Thor. The Thor system visualizes the result of cache-miss profiling on parallel programs [7]. The memory profile gives each cache miss a type and categorizes it by processor, function, and data structure. The main Thor view can show a stacked bar graph with a bar for every combination of the three categories (e.g., every cache miss made by the n^{th} processor in function `foo` to a particular data structure). Each bar shows the number of cache misses of each type.

When displaying a bar for every possible combination, there are too many bars to be useful, so Thor provides several ways to filter the display. The user can choose to collapse one of the categories, so the view will only show bars for each combination of the other two categories (e.g., every cache miss made by the n^{th} processor in function `foo` to *any* data structure). The user can also choose which processors and functions to show bars for. Finally, the user can hide all bars with a total number of cache misses that is below a specified threshold.

Intel VTune

The Intel VTune Performance Analyzer is a performance profiler with a visualization utility [20]. After collecting data, VTune displays a line plot, shown in Figure 2.14(a), with several statistics, including the target program's memory and processor usage.

A call graph visualization, shown in Figure 2.14(b), shows a graph with program functions as nodes. Expanding a node, by clicking a button on the right of the node, shows its callees. The user can also use a

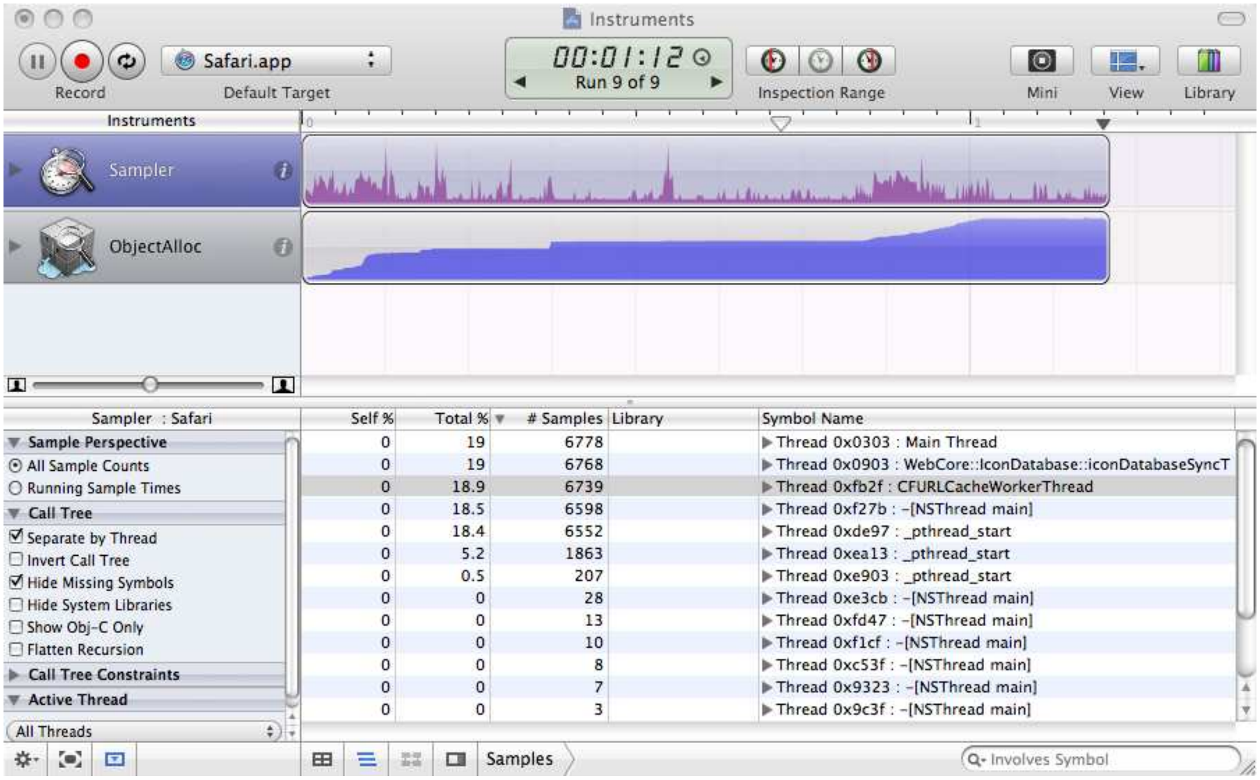


Figure 2.15: An Instruments session with two instruments. The sampler instrument shows a graph of call stack depth over time, and the ObjectAlloc instrument shows a graph of memory usage. The lower panel shows detailed information from the sampler instrument. Each entry shows how much time the program spent in a function.

button on the left to expand a function's callers.

Each function has one critical line that goes to the callee that the function spent most of its time in, which the call graph draws in red. By following the red arrows, a user can trace a "critical path" in the program execution.

Holding the mouse over a function reveals a tooltip with detailed statistics about that function.

2.4.2 Real-Time Tools

Instruments

In 2007, Apple released Instruments, a profiler with several instrumentation backends, including DTrace [28], that provides a real-time visualization of its trace data [1]. The user chooses "instruments" that collect data from the target program. Among the available instruments are a sampling profiler, which plots call-stack depth; a memory usage profiler, which plots bytes allocated; and a processor utilization monitor, which plots overall CPU utilization for all processes. The user can add custom instruments by writing DTrace "probes."

Each instrument collects data and draws a line plot of that data while the program runs. All the line plots share a time axis, so it is possible to correlate events from different instruments in time. The user can zoom both the shared time axis and the individual y-axes.

A panel below the instrument plots shows the individual events that the selected instrument captured. By default, the bottom panel shows all the collected data, but the user can select a region of time to see only data for that time.

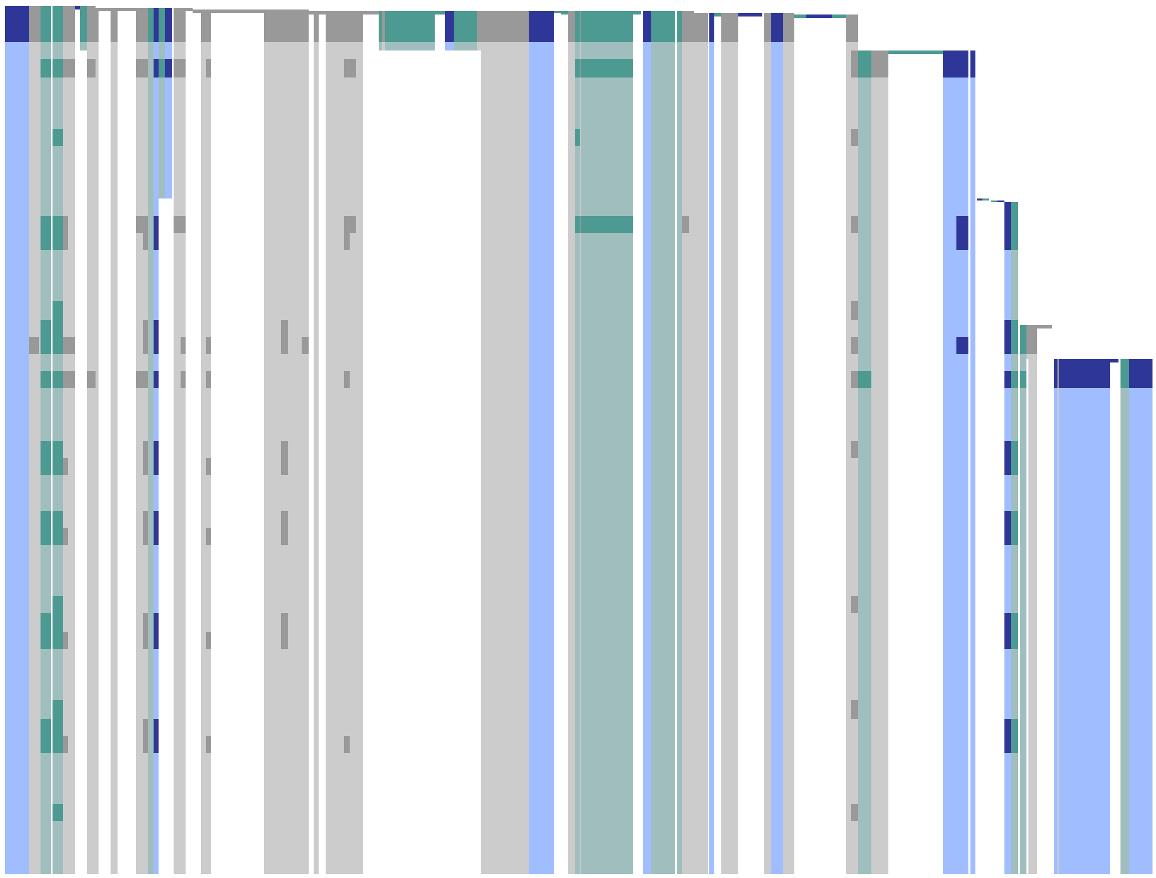


Figure 2.16: A Memcov Viz display of a running text editor. Time advances downward in this view, so the bottom of the figure represents the time the screenshot was taken at. The thick blue bars all the way to the right are large allocations that the editor created when changing modes. Periods of memory access (darker regions in the bars) further down the view correspond to the user entering text in the editor. The blue and green bars are each Gantt charts for individual ranges. The gray regions aggregate data for smaller ranges.

Memcov Viz

Memcov Viz is our real-time memory access visualization tool. It uses the Memcov system as a backend to track dynamic memory allocations, deallocations, and accesses [8]. Memcov observes an allocated range of memory by turning off its read and write protections and then waiting for a segmentation fault. The segmentation fault indicates an access to the region. Memcov learns about ranges by interposing on `malloc` and `free`.

We designed Memcov Viz as a tool to investigate interactive programs. For instance, a user may want to understand what happens when entering text in a text editor. The user can watch the Memcov Viz memory view while typing text in the target program and then immediately see which regions it accesses, allocates, and destroys.

When the target program creates a region with `malloc`, Memcov Viz shows the region as vertical Gantt chart, with a width equal to the range's size. Large regions stand out because they have thicker Gantt charts. Figure 2.16 shows an example memory view.

Each Gantt chart shows when accesses occur for its memory region. Dimmer regions of the chart indicate periods of non-access. Over time, the Gantt charts scroll up, so that the bottom of the display is always the current time.

Using the mouse wheel, the user can smoothly zoom the x-axis (measured in bytes) to show more memory ranges or to see more detail. Memcov Viz emphasizes smooth animation of the view. The user can quickly zoom out to search an overview of the whole graph for an interesting region then zoom in to investigate that region.

In order to provide a useful overview, Memcov Viz needs to show a lot of memory ranges on the screen at once. At far enough zoom to show all of a program's ranges, most of those ranges are too thin to display. To cope with this problem, Memcov Viz combines adjacent ranges together until they form a Gantt chart that is thick enough to display. These aggregated Gantt charts are gray to distinguish them from Gantt charts for individual regions. Memcov Viz marks them in the accessed state when any of their ranges are accessed. To see all the ranges in an aggregated Gantt chart, the user need only zoom in on it.

Moving the mouse over a memory range's Gantt chart highlights it and displays text information. The user can see the range's exact size in bytes, the time since its last access, and its call site. A memory range's call site is the address of the call to `malloc` that created it.

Since a user might be interested in examining the behavior of memory ranges created by a particular line of code, Memcov Viz provides the option of grouping ranges by call site. With this option on, MemcovViz orders ranges so that those created by the same call to `malloc` appear next to each other, instead of ordering them by creation time.

In order to achieve the fast animation we need for fluid zooming, we use OpenGL as a rendering backend. OpenGL allows us to take advantage of hardware acceleration for drawing and scaling.

Chapter 3

Visualizing Data

3.1 Types of Data

In this chapter, we endeavor to classify the types of data that execution visualization tools present, giving examples from the tools discussed in Chapter 2. Our classification adds to several existing comparisons of visualization tools.

Stasko et al. present a scheme for classifying execution visualization tools by four properties [46]. The four properties, "aspect," "abstractness," "animation," and "automation," are practical metrics for comparing visualization tools. One useful observation is that high aspect and abstractness tools, those that provide a high-level view, are low automation. Such views are specific to the program they visualize, whereas high automation tools need to be more general.

Shneiderman presents a more general classification for all types of visualizations, called a "task by data type taxonomy" [44]. It includes seven visualization tasks and seven data types. Though our data type classification is narrower, focusing only on program execution visualization, it is also much simpler, defining only three data types.

Shneiderman's seven tasks are inspired by his "Visual Information Seeking Mantra:"

> Overview first, zoom and filter, then details-on-demand.

The mantra orders four of the tasks into a workflow that is a basis for an interactive visualization. Bosch et al. cite the mantra as an important insight while developing the Rivet visualizations discussed in Section 2.4.1 [6].

Memcov Viz (Section 2.4.2) also draws inspiration from Shneiderman's mantra. Aggregated timelines make it possible to see a whole program overview even when a program has a lot of memory ranges. The user can smoothly zoom using the mouse wheel and filter by call site. Finally, highlighting a memory range provides details on demand.

Our classification system categorizes the data that execution-visualization tools present into three types: *statistics*, *actions*, and *relationships*. All the tools discussed in Chapter 2 show views of some combination of these data types.

3.1.1 Statistics

Statistics are the most basic data that a visualization can show. We define a *statistic* as a value that changes over time, so a statistic is most at home on a line plot with time on one axis.

Any input can be a statistic. Memcov Viz, for example, takes a program's sequence of memory accesses as its input. Such a sequence serves as a more detailed input than a simple statistic. Simply measure the percent of recently accessed memory ranges, however, and the input is expressible as a statistic.

For just this reason, statistics are the simplest and most flexible subject for visualization. There exists no system that one cannot visualize by measuring its properties and then plotting them together on a common time axis.

This type of data is limited in its power, however. A line plot with the number of recently used memory ranges, for instance, will show you that a program is not using most of its ranges, but it cannot tell you why. At best, the user needs another statistic to see that. In short, a statistic provides Shneiderman's overview, but without other types of data, it does not provide detail.

Displays beyond simple line plots help correlate multiple statistics measured over the same time, but they do not provide any of the missing detail. A Kiviat diagram (Section 2.4.1) combines multiple statistics along with a simple approximation of their total in the form of the diagram's area. Stacked histograms show how several statistics combine to form a total, which serves as an additional statistic. Phase diagrams (also Section 2.4.1) show direct relationships between pairs of statistics.

3.1.2 Actions

Actions form an important type of data in execution visualization because a program execution is a sequence of actions. The GPV and TraceVis visualizations (Section 2.4.1) represent programs as a sprawling timeline of processor instructions. Each processor instruction is an action.

We define an *action* as a piece of data with at least an associated time: the time when the action occurred. An action can have additional information. An instruction, for instance, is associated with its instruction name and operands.

Because every action has a time, it makes sense to visualize actions on a timeline, like the GPV and TraceVis pipelines. A Gantt chart is a type of timeline that shows actions. Each change in state represents an action.

In addition to timelines, actions are naturally suited to animated views. An animation without any actions is just a picture. In the XTANGO quicksort animation (Figure 2.1), for example, each animation of two numbers switching places directly represents a program action.

3.1.3 Relationships

Finally, relationships are crucial to understanding a software system. Software with any kind of entities—data structures, objects, threads—must have relationships because those entities will interact with each other.

Graphs are an intuitive way to show relationships. Consider the *has a pointer to* relationship. Box-and-arrow diagrams (Figure 2.2) show this relationship as a graph with records as nodes and their relationships as edges. Graphs also appear in most object-oriented program visualizations. They show relationships like those between a class that calls another class' method or an object that instantiates another object.

Matrix views, like the ParaGraph message-passing matrix (Section 2.4.1), are also useful for showing relationships. A matrix view is not far removed from a graph view; programs commonly represent graphs in memory as adjacency matrices. Visually, a matrix has the advantage of avoiding the clutter from many overlapping edges that occurs in dense graphs, as in Figure 2.6. However, it is hard for a viewer to follow a path through multiple relationships in a matrix view, an operation that is very natural in a graph view.

3.1.4 Combining Types of Data

Developers of visualizations need not limit themselves to representing data from these categories alone, however. Many visualizations combine these three types of data.

Many graph views, for example, annotate nodes and edges with extra information. This technique is a powerful way to combine statistics and relationships. Run-time polymorphic views [4], discussed in Section 2.3.5, pull graph nodes together to visualize the strength of relationships in the graph (as in Figure 2.6). The object-oriented TraceVis tool, discussed in Section 2.3.7, puts a statistic directly on each edge by varying its thickness with the value (as in Figure 2.8).

The ParaGraph space-time view (Figure 2.10(b)) combines action and relationship data. The view shows inter-processor messages, which are each actions. Since each message connects two processors, the sending and receiving processors, it also serves as relationship data. Views closely modeled after UML sequence diagrams, like the Program Explorer invocation view (Figure 2.4) discussed in Section 2.3.2, are similar to the space-time view. They also combine action and relationship data. This time, the action is the method call, and the relationship is between the calling object and the callee object.

The overlaid bar charts and histograms in GPV and TraceVis (as in Figure 2.12(b)) combine statistics and actions. The means of combining the two, laying one on top of the other, is simple but powerful. The viewer can see each instruction in the context of the statistical data. The statistics provide the overview, and the instructions provide the missing details.

Chapter 4

Areas of Future Research

4.1 Interacting with the visualization

Some of the visualization tools presented here run at the same time as the target program. Using a real-time visualization, the user can perform experiments on the running program by interacting with it in some way and then watching for the effect in the visualization.

An interactive visualization tool, however, could allow experiments that the user initiates directly from the visualization window. The user could alter program entities by selecting them in the view and then immediately see the results of the change.

Data-structure visualizations already support this kind of manipulation. GDBX [3] (Section 2.2.2) and similar debuggers allow the user to directly modify record values and reassign pointers by dragging arrows. The views are not real-time, however. The user must pause the target program, make the change, then resume the program.

To give an example of such an interactive visualization, Memcov Viz (Section 2.4.2) could provide tools to manipulate allocated memory ranges directly. A developer of mission-critical algorithms that rely on redundant data structures might want a way to corrupt program data for testing purposes. The developer would click a range in the Memcov Viz view to corrupt the data it holds and would then see accesses to the redundant regions as the target program recovers from the corruption.

Alternatively a user might want to be able to click a region to deallocate after becoming convinced that the program will not access it again. That way, the program would have more memory to use for the rest of the debugging session.

4.2 Overviews

Shneiderman's "Visual Information Seeking Mantra" [44] demands an *overview*, but when a visualization provides huge amounts of data, it is not always obvious how to show all of it in a single overview. A computer monitor's resolution limits how many entities a visualization can present. Extremely high-resolution monitors could push this boundary back, but it would not take long to hit the more fundamental limits presented by the viewer's visual acuity.

Several of the tools presented here provide interesting ways to distill large amounts of data into a relatively small area. The GPV and TraceVis tools (Section 2.4.1) can shrink instruction timelines until they form a thin line, and Memcov Viz aggregates memory ranges together until there are few enough entities that it is practical to display them all.

Jerding et al. present a more general way for aggregating large numbers of objects for an overview called an *information mural* [22]. The information mural colors each pixel in its view according to how many objects pass through the area that pixel covers. Dark areas in the view represent areas that are densely packed with objects.

All of these methods throw away all the visual patterns that are available at close range, however. Though there is no way to avoid omitting detail when constructing an overview, it might be possible to preserve enough to be useful to the viewer.

Perhaps computer vision and image processing algorithms could find the patterns that a viewer with a perfect view of the entire visualization would see. For example, in a GPV view of a program with a rapidly fluctuating rate of instructions per second, the line created by the view would form a stair-step pattern. After zooming out far enough, however, the stair-step pattern would be too small to see, instead appearing as a diagonal line. A computer vision algorithm could see this pattern as an interesting feature and add an annotation to regions of the view that follow it.

4.3 Three-dimensional views

Only a few of the tools presented here take advantage of three-dimensional views. New kinds of three-dimensional views could potentially display more data and reveal more sophisticated patterns.

Ubiquitous graphics hardware makes 3D visualizations even more tempting. Data sets represented by hundreds of thousands of polygons are well within reach using consumer hardware, and programmable pixel and vertex pipelines are advanced enough to share some of the burden of mapping input data onto the 3D output.

However, the problem of displaying large overviews gets even worse in 3D. When entities clutter a 3D view, they occlude each other. In the worst case, the bulk of data could be hidden from any viewpoint.

Solving the occlusion problem requires a more advanced way to navigate the data. Perhaps the user could fly through the visualization or interactively construct cutaway views to reveal different parts of the data set.

Chapter 5

Conclusion

In this paper, we have surveyed tools and techniques for visualizing program executions. These visualizations can illustrate patterns that are hidden in the huge number of events that a program run generates: information that is useful for optimizing performance, fixing bugs, or learning an unfamiliar program.

Additionally, we presented our own visualization tool, Memcov Viz, which visualizes dynamic memory events. Memcov Viz seeks to present a powerful, real-time overview, even for large numbers of memory regions.

After examining these visualization tools, we turned an eye to the data they present, categorizing the tools by the types of data they show. The three categories of visualization data offer a framework for understanding what kinds of program properties are good targets for visualization.

We concluded with a discussion of possible future directions for research in software execution visualizations. New visualizations have the potential to control the program execution that they display or to visually interepret results for the user.

Bibliography

[1] Apple, Inc. *Instruments User Guide*, 2007.

[2] R. M. Baecker and D. Sherman. Sorting out sorting. Shown at SIGGRAPH '81, 1981. 16mm color sound film.

[3] D. B. Baskerville. Graphic presentation of data structures in the DBX debugger. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1985.

[4] R. Bertuli, S. Ducasse, and M. Lanza. Run-time information visualization for understanding object-oriented systems. In *International Workshop on Object-Oriented Reengineering*, 2003.

[5] R. Bosch, C. Stolte, G. Stoll, M. Rosenblum, and P. Hanrahan. Performance analysis and visualization of parallel systems using SimOS and Rivet: A case study. In *HPCA*, pages 360–, 2000.

[6] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: a flexible environment for computer systems visualization. *SIGGRAPH Comput. Graph.*, 34(1):68–73, 2000.

[7] R. P. Bosch, Jr. *Using Visualization to Understand the Behavior of Computer Systems*. PhD thesis, Department of Computer Science, Stanford University, August 2001. `http://graphics.stanford.edu/papers/bosch_thesis/`.

[8] S. Callanan, R. Grosu, J. Seyster, S.Ã. Smolka, and E. Zadok. Model Predictive Control for Memory Profiling. In *Proceedings of the 2007 NSF Next Generation Software Workshop, in conjunction with the 2007 International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long beach, CA, March 2007.

[9] CenterLine Software, Inc. *CodeCenter Tutorial*, 1995. `http://products.ics.com/products/codecenter/codecenter-4.1.1-tutorial.pdf`.

[10] P. Deelen. Visualization of dynamic program aspects. Master's thesis, Technische Universiteit Eindhoven, June 2006. `http://www.win.tue.nl/~wstahw/projects/finished/PieterDeelen/downloads/deelen2006.pdf`.

[11] Universität des Saarlandes. *Debugging with DDD*, 2004. `http://www.gnu.org/manual/ddd/`.

[12] S. G. Eick, J. L. Steffen, and E. E. Sumner, Jr. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[13] The Free Software Foundation, Inc. GDB: The GNU Project Debugger. `www.gnu.org/software/gdb/gdb.html`, January 2006.

[14] F. Freitag, J. Caubet, and J. Labarta. A trace-scaling agent for parallel application tracing. In *ICTAI '02: Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'02)*, page 494, Washington, DC, USA, 2002. IEEE Computer Society.

[15] A. Geist, J. Kohl, and P. Papadopoulos. Visualization, Debugging, and Performance in PVM. In *Proceedings of Visualization and Debugging Workshop*, October 94.

[16] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Softw.*, 8(5):29–39, 1991.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[18] Hewlett-Packard Company. *C and C++ SoftBench User's Guide*, June 2000. `http://docs.hp.com/en/B6454-97413/B6454-97413.pdf`.

[19] H. Igehy, G. Stoll, and P. Hanrahan. The design of a parallel graphics interface. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 141–150, New York, NY, USA, 1998. ACM.

[20] Intel Corporation. Intel VTune Performance Analyzer. `http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm`, 2007.

[21] S. Isoda, T. Shimomura, and Y. Ono. VIPS: A visual debugger. *IEEE Softw.*, 4(3):8–19, 1987.

[22] D. F. Jerding and J. T. Stasko. The information mural: Increasing information bandwidth in visualizations. Technical Report GIT-GVU-96-25, Georgia Institute of Technology, Atlanta, GA, 1996.

[23] A. Knüpfer, H. Brunst, and W. E. Nagel. High performance event trace visualization. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 258–263, Washington, DC, USA, 2005. IEEE Computer Society.

[24] J. Labarta, S. Girona, V. Pillet, T. Cortes, and J. Cela. A parallel program development environment, 1995.

[25] C. Laffra and A. Malhotra. Hotwire: a visual debugger for C++. In *CTEC'94: Proceedings of the 6th conference on USENIX Sixth C++ Technical Conference*, pages 7–7, Berkeley, CA, USA, 1994. USENIX Association.

[26] D. B. Lange and Y. Nakamura. Program Explorer: a program visualizer for C++. In *COOTS'95: Proceedings of the USENIX Conference on Object-Oriented Technologies on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 4–4, Berkeley, CA, USA, 1995. USENIX Association.

[27] S. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. SUIF Explorer: an interactive and interprocedural parallelizer. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 37–48, New York, NY, USA, 1999. ACM.

[28] Sun Microsystems. Solaris dynamic tracing guide. `docs.sun.com/app/docs/doc/817-6223`, January 2005.

[29] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[30] S. Mukherjea and J. T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 456–465, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[31] T. Munzner. Exploring large graphs in 3d hyperbolic space. *IEEE Comput. Graph. Appl.*, 18(4):18–23, 1998.

[32] B. A. Myers. INCENSE: A system for displaying data structures. *SIGGRAPH Comput. Graph.*, 17(3):115–125, 1983.

[33] W. Nagel and A. Arnold. Performance visualization of parallel programs—the PARvis environment, 1994.

[34] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[35] O. Naím, T. Hey, and E. Zaluska. Do-loop-surface: an abstract representation of parallel program performance. *j-CPE*, 8(3):205–234, April 1996.

[36] T. Okamura, B. Shizuki, and J. Tanaka. Execution visualization and debugging in three-dimensional visual programming. In *IV '04: Proceedings of the Information Visualisation, Eighth International Conference on (IV'04)*, pages 167–172, Washington, DC, USA, 2004. IEEE Computer Society.

[37] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. *SIGPLAN Not.*, 28(10):326–337, 1993.

[38] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[39] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 116–134, London, UK, 1999. Springer-Verlag.

[40] D. Reed, R. Aydt, T. Madhyastha, R. Noe, K. Shields, and B. Schwartz. An overview of the Pablo performance analysis environment. Technical report, University of Illinois, Urbana, IL, USA, 1992.

[41] S. P. Reiss. Visualizing Java in action. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 57–ff, New York, NY, USA, 2003. ACM.

[42] J. E. Roberts. TraceVis: An execution trace visualization tool. Master's thesis, University of Illinois at Urbana-Champaign, July 2004. `http://www-sal.cs.uiuc.edu/~zilles/papers/tracevis.msthesis.pdf`.

[43] L. De Rose, Y. Zhang, and D. A. Reed. SvPablo: A multi-language performance analysis system. *Lecture Notes in Computer Science*, 1469:352–??, 1998.

[44] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society.

[45] J. Stasko. Animating algorithms with XTANGO. *SIGACT News*, 23(2):67–71, 1992.

[46] J. Stasko and C. Patterson. Understanding and characterizing program visualization systems. Technical Report GIT-GVU-91-17, Georgia Institute of Technology, Atlanta, GA, October 1993.

[47] J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *J. Parallel Distrib. Comput.*, 18(2):258–264, 1993.

[48] C. Stolte, R. Bosch, P. Hanrahan, and M. Rosenblum. Visualizing application behavior on superscalar processors. In *INFOVIS '99: Proceedings of the 1999 IEEE Symposium on Information Visualization*, page 10, Washington, DC, USA, 1999. IEEE Computer Society.

[49] Sun Microsystems, Inc. *dbx man page*. Sun Studio 11 Man Pages, Section 1.

[50] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 271–283, New York, NY, USA, 1998. ACM.

[51] C. Weaver, K. C. Barr, E. Marsman, D. Ernst, and T. Austin. Performance analysis using pipeline visualization. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2001)*, Tucson, AZ, November 2001.

[52] A. Zeller and D. Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996.