

# Demonstration: On-Line Visualization and Analysis of Real-Time Systems with TuningFork

David F. Bacon<sup>1</sup>, Perry Cheng<sup>1</sup>, Daniel Frampton<sup>2</sup>,  
David Grove<sup>1</sup>, Matthias Hauswirth<sup>3</sup>, and V.T. Rajan<sup>1</sup>

<sup>1</sup> IBM T.J. Watson Research Center

<sup>2</sup> Department of Computer Science, The Australian National University

<sup>3</sup> Università della Svizzera Italiana

**Abstract.** TuningFork is an online, scriptable data visualization and analysis tool that supports the development and continuous monitoring of real-time systems. While TuningFork was originally designed and tested for use with a particular real-time Java Virtual Machine, the architecture has been designed from the ground up for extensibility by leveraging the Eclipse plug-in architecture. This allows different client programs to design custom data formats, new visualization and analysis components, and new export formats. The TuningFork views allow the visualization of data from time scales of microseconds to minutes, enabling rapid understanding and analysis of system behavior.

## 1 Introduction

When designing and developing system software of significant complexity, meeting performance goals is as important and challenging as correctness. In the case of a real-time system, coarse-grained performance characteristics such as overall throughput alone are not enough to verify responsiveness or determine the causes of failure. The ability to measure and visualize fine-grained events is necessary for determining correctness and analyzing why the system misbehaved.

The large volume of data often generated by these systems is hard to understand without visualization. In production systems where downtime is unacceptable, online monitoring and analysis can be useful for problem determination and resolution. During development, a real-time system must be tested for performance regression automatically and a useful analysis tool must also support scripting commands.

In the accompanying presentation we will demonstrate *TuningFork*, an online, scriptable, and re-configurable data visualization and analysis tool for the development and continuous monitoring of real-time systems. TuningFork is an Eclipse plug-in using the Rich Client Platform (described at [www.eclipse.org](http://www.eclipse.org)), and itself exports a plug-in architecture that allows user-defined data stream formats, stream filters, and visualizations. Because TuningFork is still under rapid development, it is not yet available for download.

TuningFork is a combination of known and novel techniques and visualizations, but it is the whole that is greater than the sum of the parts. TuningFork's

features include a real-time centered design that adapts to data loss and event reordering due to resource constraints in the traced system, the ability to handle very large volumes of data online with a running system, an adaptive data summarization framework allowing even more past data to be viewed, the ability to play the data in forward and reverse, plugin-based extensibility of trace formats and views, a composable data stream abstraction that allows creation of new synthetic events, and the ability to run the same system in batch mode with a scripting language.

Novel views include the “oscilloscope” view that presents interval data in a sequence of “time strips”. With a large LCD display, this allows 2-3 seconds of data to be visualized at  $10\mu\text{s}$  resolution, or 20-30 seconds of data at 10ms resolution, with user-selectable continuously variable time scales. When play mode is off, data can be viewed down to the nanosecond scale. Furthermore, a statistical superimposition facility allows the overall behavior of huge amounts of periodic high-resolution data to be visualized (hence the oscilloscope analogy).

The demonstration will show how TuningFork is used to diagnose run-time anomalies in real-time behavior in Metronome, our real-time garbage collector for Java implemented in the IBM J9 virtual machine product. The various views allow the identification of a failure to meet a high-level response-time specification using a time-strip animation, followed by identification of the cause using a histogram which categorizes different atomic sections of the garbage collector, and culminating in the identification of the precise point of failure in the execution of the program using the oscilloscope view. We will also show how to evaluate memory policy using a spatial view that shows the physical or logical state of the heap.

## 2 Architecture

At the high level, TuningFork’s architecture consists of a thin client-side layer which transmits application or JVM events and the server-side TuningFork visualization application which we simply call TuningFork. The client is instrumented at various points to collect special information and send the data in an application-specific *feed* to TuningFork via a socket or to a file for post-mortem analysis. Although our primary client of TuningFork is a JVM, any system that emits trace files in the specified format is a suitable target for TuningFork.

At the high-level, the feed is broken into *chunks* which are the units of network transmission to TuningFork. Certain initial chunks describe overall properties as well as the format of the rest of the feed. The event chunks are the most interesting and constitute the bulk of the feed. Each event chunk includes a chunk identifier so that TuningFork can obtain the appropriate interpreter plugin for that application. Since the client application may be multi-threaded, the data feed is broken into feedlets and each event chunk contains data only from one feedlet.

Because TuningFork is fundamentally a time-based tool, all events have a time stamp, typically the value of a cycle counter which on current architectures

provides nanosecond-scale resolution. In order to present a globally time-ordered view of events to the rest of TuningFork, data from different feeds are merged into a single global feed by making data at a certain time visible only after all feeds have reached that point in time.

Because TuningFork is built on top of the Eclipse Rich Client Platform, it is simple for the application developer to export application-specific portions such as an *event chunk interpreter* to TuningFork via the plug-in architecture. The application also can export *filters* which convert events to non-application-specific streams. These streams can then be composed into figures for visualization.

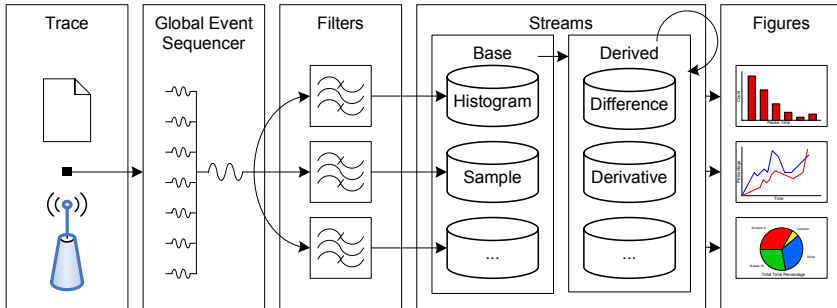


Fig. 1. TuningFork Architecture

## 2.1 Streams

In a real-time system, many quantities of interest are time-series data. Because TuningFork can be used to monitor an online system or large post-mortem trace files, the volume of data will generally exceed the memory capacities of TuningFork. The traditional method of implementing a ring buffer is simple but has the disadvantage of losing data that is older than the size of the buffer, making diagnosis of certain problems difficult and understanding of long-term trends impossible. In addition, computation and display of data streams is complicated by requiring a constant awareness of this possibility.

This problem is greatly reduced by continuing to use a stream abstraction so that a stream appears to be a function whose domain is an ever-increasing time range and whose range is dependent on the particular stream. For example, a stream representing memory usage would map time values to bytes while a stream representing interrupt handler execution would map to time intervals. However, depending on memory pressure, the precision of older data may be continuously degraded by aggregated data into a collection of statistics.

The functional aspect of streams simplifies the computational model by allowing standard mathematical functions like addition, differentiation, and smoothing via convolution. Streams are initially created by applying filters to the events that enter the system. For example, a “used memory” filter would generate pairs of values  $(t, m)$  which are used to create the used memory stream which logically consists of the function  $memory(t)$  and the range  $[t_{start}, t_{end}]$ . A new stream,

*allocation rate*, can be created by applying the differentiation filter to the “used memory” stream.

Certain operators will take operand streams of different types. For example, one can take a value stream (e.g. used memory) and a time interval stream (e.g. time intervals when the garbage collector is off) and create a value stream which shows only used memory when the garbage collector is off.

Other base types include categories which is useful for understanding the relationship of a set of quantities. For example, we might have a category-value stream which would show the duration of each GC pause and the type of activity the collector was performing in that time interval. A histogram of such a stream would then show not only the average and maximum pause time but also what the collector was doing during those pauses.

## 2.2 Figures

At the heart of TuningFork are the visualization components, called *figures*. Figures are responsible for taking streams of data and displaying them to the user. The figure architecture has been designed for extensibility, device-independent rendering, and high performance to allow the display of live data feeds with high data rates.

Visualizations are typically composed of several common reused subcomponents. Histograms, axes, legends, and time series plots may occur many times within different visualizations, albeit with minor differences in display characteristics. This approach is important because of our goal of allowing the user to extend the system by plugging in custom views.

In order to facilitate the rapid development of new visualization components, TuningFork introduces two key design features: a high-level drawing interface tailored to on-line visualization, and *painters*. The high level interface allows device-independent drawing; we currently have both an SWT implementation for the user interface, and a PDF implementation for printing functionality. The programming interface includes simple painting functionality for basic shapes. Painters build on top of this simple interface to provide more complex, data-dependent visualization components such as axes, histograms, and time series plots.

Within this design framework, the role of a figure becomes to divide the visualization display into different areas, determine the precise data that needs to be drawn, and delegate drawing to various painter implementations. Additionally, the figure contains all state regarding the display settings for the visualization component. This can be accessed both through a host eclipse view, and via the programmatic *conductor* interface.

## 2.3 Conductor

The interactive visualization and analysis of TuningFork is very powerful. However, there is also a need for automated analysis, in particular for such tasks as regression testing where the results of the analysis must be fed into automated

tools that report performance anomalies and automatically create work items in the product development database.

Such text-based analyses are typically written as entirely separate tools. However, the modular stream processing, filtering, and transformation facilities in TuningFork are extremely useful for building such analyses. In order to minimize code duplication, facilitate the creation of automated analyses, and to provide a more productive environment for power users, TuningFork includes the *conductor* – a lightweight scripting environment.

It is possible to perform nearly all visualization operations from within the conductor, such as connecting to traces, opening figures, performing analysis, and exporting PDF files. Additionally, due to the pluggable nature of the application, it is possible to run the conductor outside of the graphical user interface, an important capability for automated testing. This allows the creation of tools produce purely textual results for use in larger programmatic systems, and for the creation of visualizations of exceptional conditions that can be uploaded into a web-based graphical database.

### 3 Comparisons and Conclusions

TuningFork has drawn on many sources of inspiration (space constraints unfortunately do not permit formal citations), particularly the work of Tufte on visual display of quantitative information. It is perhaps most similar to the *PV* Program Visualizer (Kimmelman et al), which can visualize very large event traces without loading the complete trace into memory. It provides an animated visualization of the information in a sliding window over the trace. *PV* supports temporal vertical profiling, integrating information from hardware, operating system, native libraries and native applications. It mainly focuses on visualizing events, states, and the corresponding source code, but can also visualize the value of a metric over time.

Much prior visualization work has focused on parallel systems and their complex behavior, including Paradyn (Miller et al), Jumpshot (Zaki et al), Pablo (Reed et al), and others. Real-time behavior presents its own unique challenges, but shared with such systems a need to coordinate the time scales of many independent parts running on potentially distributed or unsynchronized clocks.

TuningFork is a comprehensive tool for visualization and analysis tool for real-time systems. TuningFork allows visualization of real-time events as they are happening, and provides views that allow data to be visualized across a very wide range of time scales, while still providing a high degree of resolution. Our experience has shown that the broad range of visualization capability promotes a deep understanding of the detailed behavior of real-time systems at both macro and micro time-scales.