RC24162 (W0701-116) January 24, 2007
Computer Science

IBM Research Report

TuningFork: Visualization, Analysis and Debugging of Complex Real-time Systems

David F. Bacon, Perry Cheng, Daniel Frampton*, David Grove
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*Australian National University

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

TuningFork: Visualization, Analysis, and Debugging of Complex Real-time Systems

David F. Bacon, IBM Research
Perry Cheng, IBM Research
Daniel Frampton, Australian National U.
David Grove, IBM Research

Abstract

Debugging the timing behavior of real-time systems is notoriously difficult, and with a new generation of complex real-time systems whose size is measured in tens of millions of lines of code, the difficulty is increasing enormously. We have developed TuningFork, a tool especially designed for visualization, analysis, and debugging of large-scale real-time systems. The system is capable of recording high-frequency events at sub-microsecond resolution with almost no perturbation to the application. The visualization tool is capable of viewing system activity online in real-time, and users can simultaneously explore the data interactively. TuningFork has allowed us to find numerous timing bugs and anomalies, including unexpected scheduling behavior, clock resets, delayed lock release, and introduction of non-constant-time functions.

1 Introduction

Real-time systems are becoming a pervasive part of the computing landscape, forming an ever-increasing fraction of deployed systems and developed lines of code. The control software for an automobile is expected to reach 100 million lines of code early in the next decade. Financial, aerospace, telecommunications, computer games, cellular phones, and military real-time systems will reach similar levels of complexity.

Debugging even traditional isolated real-time systems presents unique challenges, and when coupled with the scale at which real-time systems are being developed for future deployment, the difficulty of the problem increases dramatically.

Unlike the debugging of conventional throughput-oriented systems, debugging of real-time systems requires an absolutely minimal probe effect, since otherwise the very timing which one is attempting to debug will be disturbed, and if the probe effect is large enough then the system will simply cease to function. Furthermore, sampling-based approaches which compute aggregate statistics are of limited use, since an error might consist of a procedure consuming a few extra microseconds at a single point in a week-long execution.

Our work on debugging and visualization of real-time systems grew out of practical necessity: with the development of our real-time garbage collection technology [1], its deployment in commercial products [8], and its adoption for the creation of such large real-time systems, we faced the need to provide a comprehensive tool to support the software lifecycle of such complex real-time systems that involved the interaction of the operating system, the Java virtual machine, and applications written in both Java and traditional statically compiled languages.

The result is TuningFork, an Eclipse-based tool that supports online visualization and analysis of real-time activity by multiple subsystems, now publicly available via IBM alphaWorks [7].

In this paper we will begin by describing the unique constellation of requirements for debugging the timing behavior of complex real-time systems, and then describe the system architecture and visualization capabilities that were created to support those requirements.

2 Requirements for Real-time Debugging

TuningFork's unique capabilities came about because of the demanding collection of requirements created by the environment we were trying to support: the construction of safety-critical real-time systems from multiple independently developed sub-components running on a network of server-class multiprocessor machines.

Finding Needles in Haystacks. The origin of a failure could be in a single event in a week-long execution taking four microseconds instead of one. This means first of all that summarization-based approaches, popular in tools like `gprof` [5] and all of its derivatives, are not useful. It also means that the system must support the generation and processing of a very large volume of events, on both the trace generation side and the visualization tool side.

Discovery of Unexpected Behavior. Many failures are

caused by effects for which the user as yet has no quantitative measure. Therefore the tool must provide visualization capabilities that facilitate observation of unexpected behavior.

Trace Collection as a Real-time Task. The generation of trace events is the portion of the tooling that occurs in the actual real-time system itself, as opposed to being performed in the tool. Thus it is absolutely critical that trace collection itself must be implemented such that its effects are absolutely minimal, predictable, and pre-emptable.

Minimal impact requires an extremely efficient event logging mechanism, and minimal computation of derived metrics inside of the running system. Predictability means no slow off-branch cases, in particular no file output, memory allocation, or synchronization performed on the application threads. And pre-emptabiliy means that in case of overload the entire logging mechanism may be suspended, meaning that the tool must be robust in the face of lost events.

Trace Visualization as a Real-time Task. Since faults may be safety-critical, it is imperative to be able to observe, diagnose, and correct the problem as quickly as possible. This means that the tool must be able to present the information in a short, bounded amount of time after the originating event occurs.

"TiVO" Control of Visualization. In order to observe events in real-time, the system must be capable of rendering a dynamic image of events, rather than a static representation. Furthermore, it must be possible to pause, reverse, and change zoom factor in order to quickly hone in on an unusual event.

High-performance Rendering. In order to achieve high-quality real-time playable views, the rendering performance of each view must be very good: the user may have 6 views open in play mode, each showing information about thousands of events. In play mode we would like to achieve 24 frames per second, which leaves less than 7 milliseconds to render each frame of each view.

Interactive Data Exploration. Many real-time faults begin as unexpected non-failing behavior. Therefore it is necessary to enable exploration of the available data by opening new views, computing and visualizing new functions over the data, changing resolutions, and moving to arbitrary points in the execution time. Furthermore, this must all be possible while in "play mode".

Zooming Hours to Microseconds. Supporting data exploration and the search for needles in haystacks requires an ability to rapidly navigate across huge time scales: from several hours down to a few microseconds. In other words, the tool must be able to scale across 11 orders of magnitude with interactive response. This is an issue in both the user interface – providing an intuitive and perceptually stable interface to navigate across time scales – and in the trace processing infrastructure – by providing fast random access and efficient, accurate summarization.

Highly Accurate Timing. Correct diagnosis of timing failures requires very precise timing information – at submicrosecond resolutions if possible. This is considerably complicated by the fact that the highest resolution cycle timers on even small-scale multiprocessors may not be synchronized and have considerable skew and drift.

Offline Analysis of Very Large Traces. While some systems will be observed in real-time, others will be analyzed offline. A system generating a 128-bit event (64-bit timestamp and 64 bits of data) once every 10 microseconds will produce 1.6 MB/s or 138 GB/day of trace data. The system must therefore be capable of indexing, summarizing, and randomly accessing very large data sets and paging them in and out of memory.

Vertical Integration of Information. The real-time system is a composite of the hardware, operating system, virtual machine(s), and applications, some of which run on the virtual machine and some of which run directly on the operating system. Since these components may all affect and interfere with each other, the tool must be capable of integrating information from all of these sources.

Extensibility. Since the dynamic deployment environments are unpredictable, the tool must be highly extensible. This means accepting new trace formats, creating new data filters, and devloping new visualizations.

Self-Description. Traces must be highly self-descriptive so that they continue to be useful for post-mortem analysis long after they are collected. This means that they should describe in detail the environment in which they were collected, as well as the trace events that they contain. The choice of trace events is almost certain to evolve over time, and it must be possible to comprehend traces of two different versions of the same application.

Testing for Timing Regressions. The tool must support automated testing and regression analysis. This implies scripting and the ability to generate both quantitative metrics and visual information without user interaction.

3 System Design

Conceptually, TuningFork's architecture (Figure 1) consists of a thin client-side instrumentation layer which generates a trace. The trace can either be directly transmitted to a TuningFork instance over a socket to be visualized on-line, or be saved to a file for later analysis. A single TuningFork instance can be simultaneously connected to more than one trace source.

As discussed below, individual events in a trace are not totally ordered. Therefore, the first processing step that oc-

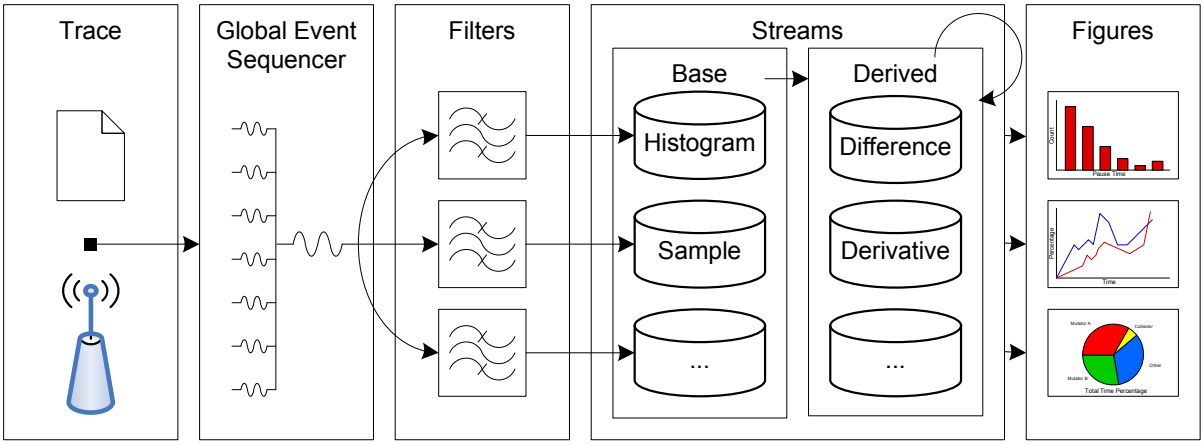


Figure 1. TuningFork Architecture

curs in TuningFork is to merge sort the incoming event data. This ordered sequence of raw events is then fed to a collection of filters to produce Streams. Streams can either be fed directly into Figures for visualization or be combined and transformed to produce derived Streams.

3.1 Trace Generators

3.1.1 Trace File Format

The format of the trace file must be designed with the performance requirements outlined earlier in mind. Because the system must diagnose timing issues that are on the order of micro- to milli-seconds, the rate of data generation can be quite high. As a general design goal, the format and trace file generation must be capable of handling a data rate on the order of one event per micro-second. The volume of data makes a binary format mandatory and rules out portable text formats such as XML. Another consideration in the design of the format is that it must be suitable for online use and be able to drop data selectively without corrupting the remaining data. Finally, data generation is multi-threaded and over-synchronization must be avoided to keep the overhead low. Because we use the same format for a *trace* file and a live *feed*, we often use these terms interchangeably.

We fulfill these requirements by imposing two structures on the trace file. First, a trace file is a concatenation of chunks of data. Chunks start and end with special bit patterns to enable boundary detection. Moreover, most chunks can be dropped without greatly affecting the data integrity of the data file. If data is dropped because the logging cannot keep up with the data generation, an explicit drop count is included on the next transmission. If data chunks are lost because of reasons beyond our control (such as by the network layer), a sequence number on the chunks allow us to detect this. The second structure in the trace file is the notion of a *feedlet*. To avoid synchronization, each thread in a system being monitored typically has its own buffers to write data into and the data belonging to this thread constitutes a feedlet. Data from a single feedlet is always time-monotone. Because the data is mostly sorted, only a final merge-sort of data from all the feedlets is required to achieve a global ordering. The locality and linearity is not important for efficiency but required for online mode because certain data computation and visualization cannot occur until all data up to a certain point in time has been received.

3.1.2 Tracing Libraries

Initially, the only source of TuningFork traces was the Real-Time JVM. As we gained experience in using TuningFork to analyze and debug the JVM's performance, it became abundantly clear that to truly understand the real-time performance of a complex system we would need the capability to instrument all of its major sub-systems. To enable this, we built generic TuningFork tracing libraries in Java and C++ (the C++ library also includes a C binding). The Java tracing library is already available as part of the TuningFork release on alphaworks [7]; we plan to include the C++ library in a subsequent release.

Using these libraries it is fairly straightforward to augment a program with instrumentation to generate a Tuning-

Fork trace. The main challenge is in determining what to instrument: the mechanics of generating the binary trace are handled by the libraries. To further reduce the instrumentation burden, we have used AspectJ [17] to build higher-level tooling to assist in instrumenting Java programs. Defining fairly simple advice files is sufficient to weave in common patterns of TuningFork instrumentation.

3.1.3 Trace Sources

In addition to the Real Time JVM, we have used the tracing libraries to instrument a number of other programs. Using the C++ library in conjunction with SystemTap [16], we instrumented the Linux kernel to generate events on thread context switches, thread creation, and thread termination. We have used the Java trace library to instrument both benchmark programs (see Section 4.2 and customer applications. By combining these trace sources, we can collect a "vertical profile" that includes OS, JVM, and application level events in a common format that can all be analyzed and visualized within TuningFork.

3.2 Events and Streams

Streams are the fundamental computational abstraction in TuningFork. Streams are either *base* streams that are defined by applying a filter over the Events in the trace or *derived* streams that are defined by applying an operator to one or more input Streams. There are several types of streams, the most commonly used are Sample Streams, which represent a series of $< time, value >$ pairs and TimeIntervalStreams, which represent a series of (possibly overlapping) intervals of time.

To give an example of how Streams can be composed, we describe how *Allocation Rate*, is derived from the primitive *Allocate Bytes* events generated by the Real Time JVM. Allocate Bytes events indicate that k bytes have been allocated at a particular time and are generated as part of all slow-path allocations. To build up Allocation Rate information, first a summing stream adds Allocate Byte events and produces an Allocated Memory Wallclock stream, which shows the allocated memory changing over physical time. An exclusion filter takes the Allocated Memory Wallclock events and the GC Thread Running events and removes the latter time intervals from the former, which produces an Allocated Memory Virtual stream which has a virtual time axis in which garbage collector execution is not considered, since we are interested in calculating the inherent allocation rate of the application when it is not being interfered with by other components. A differentiator stream differentiates the Allocated Memory Virtual stream to produce an Allocation Rate stream. Finally, a convolution filter can be applied to smooth the Allocation Rate stream, for instance to allocation rate over 1/10 second intervals, to produce the Allocation Rate Smoothed 0.1 Second stream.

TuningFork uses Eclipse's extension point mechanisms to discover pre-defined Streams (and Figures) that are applicable to the loaded trace files. This provides a means for "experts" to define commonly used Streams and Figures for a particular application domain. However, Streams and Filters can also be defined interactively from the TuningFork UI to support dynamic data exploration. The UI also supports drag-and-drop of Streams into existing figures to facilitate drill down activities.

To help discover "needles in a haystack", TuningFork supports a bookmark facility. Any TimeIntervalStream can be used to define a set of bookmarks. These points are marked in all the Figures, and the user can skip all Figures forward/backward from one bookmark to another with a single click. One common usage of bookmarks is to filter an existing TimeIntervalStream by duration to create a new Stream of just those time intervals that are unusually long (or short), then use that reduced Stream as a set of Bookmarks.

3.2.1 Indexing

Though breaking the trace file into chunks was originally motivated by lossiness and multi-threadedness, the structure proved useful for handling large trace files as well. The chunk boundaries provide a natural indexing structure and TuningFork, when encountering a new trace file, begins by creating a chunk index. Because the indexed information of a chunk is hundreds to thousands of times smaller than the chunk itself, we can afford to store the index in-core. With this index, the in-core representation of a feedlet can create a sub-index to allow random access to any event of that feedlet. Unfortunately, the data structure that is most often used is not the low-level chunks nor even feedlet events but rather a globally ordered sequence of all events. A second global index which refers to the feedlet events is needed. For example, to find the 5000*th* event, the global index would indicate that this corresponds to the 800*th* position in the first feedlet, the 2400*th* position in the second feedlet, and the 1800*th* position in the third feedlet. By starting a merge-sort at those points, we can reconstruct global event 5000 and beyond.

3.2.2 Persistence and Caching

Just as TuningFork cannot keep an entire trace file in core, it is also infeasible to keep all of the Stream data in core. In a similar fashion to how chunks are handled, we also persist Stream data to disk with an in-core cache to hold the current working set. However, to support continuous, high-performance visualization of the Streams, we also need to

maintain coarse-grained summaries of the data. This enables us to visualize the data at coarser time scales without having to access the precise data (which could consist of millions of data points) to compute the visualization. When the user zooms in on a particular region of the visualization to examine it at a fine time scale, the necessary precise data is automatically presented either by getting it from the cache or by retrieving it from disk.

3.3 Figures and Painting

One of the goals of TuningFork is to render visualizations in a device-independant manner. From rendering static images – such as PDFs, vector graphic or bitmap image formats – to displaying dynamic visualization information in real-time.

To support this, at the core of TuningFork is a custom graphics API, *TFGraphics*. Currently implemented for SWT, AWT, PDF and OpenGL, and easily extensible to others in the future, this API provides basic drawing primitives in addition to richer operations commonly required when drawing visualizations.

The core of the visualization framework in TuningFork is the *figure*. A figure performs three critical tasks. Firstly it understands the type of stream data it can interpret, providing a mechanism that allows it to be connected to source data streams. Secondly, the figure provides the rendering logic for visualizing this stream as a sequence of method calls against a TuningFork graphics API implementation. Finally, the figure provides an interface that equates to a set of control knobs that dictate both what data to display in addition to controlling how it is rendered.

TuningFork provides several basic figures by default, such as time series, histograms and pie charts. In addition, due to the unique capability of TuningFork to visualize millions of tiny events, the *oscilloscope*, described in Section 3.4 has been included. All figures understand a time axis, and can be 'played' either in real time as a live trace is fed to TuningFork, or offline through a trace file. Interestingly, this play functionality is included for the histogram, making it possible to view the state of the histogram over time.

While the included figures allows users to visualize the most common data streams, the system is designed to be extended with domain specific figures. This capability has been proven through the development of both a *staff* figure – that translates MIDI events into notes on a staff, and a *heap* figure – that takes data from an instrumented Java VM to display information about how memory is being utilized.

To support the requirements for high interactivity and the display of complex visualizations, the primary rendering engine leverages the powerful OpenGL graphics library. In combination with the stream summarization features discussed above, this enables a complete redraw of multiple visible figures to occur at a refresh rate of over 25 frames per second, providing a fluid and engaging visualization experience.

3.4 The Oscilloscope View

One of the unique features of TuningFork is the Oscilloscope view. Like a real oscilloscope, it is designed to allow visualization of very high-frequency data. Furthermore, it is able to show timing behavior across a wide range of time scales.

The oscilloscope view fundamentally visualizes time intervals. An example is shown in Figure 2. Time is shown in successive "strips", thus time proceeds from left to right and top to bottom, like a book. Each colored rectangle is a time interval, its color representing the event type. The Oscilloscope view is the most "time intensive" of views: it has time on both of its axes. By comparison, a time series view has time on one axis (and sample values on the other), while the histogram view and the application-specific heap memory view do not have any time axis, but instead present a summary or an instantaneous state, respectively, up to the current time.

The power of the oscilloscope view comes from its ability to visualize large spans of time while still retaining the ability to show very fine time resolutions. On a 1600×1200 pixel display, one can zoom out using very thin strips – 4 pixels in height. Allowing for events as small as one pixel wide, this provides the ability to display about 500,000 time units. Thus at 1 millisecond resolution one can visualize 500 seconds or just over eight minutes of execution. Because of the ability of the human eye to detect patterns, one can actually detect anomalies at the millisecond level even at this extremely wide "zoom". In Figure 2 the individual $500\mu s$ quanta are still visible [Note: this depends on your printer resolution].

However, there are cases where one needs to view even more data. For instance, we have built an audio generator which produces a half CD-quality signal, meaning that it runs at 22.05 KHz. Each time it runs and outputs a sample, we record the beginning and end of the operation. Thus we have 44,100 events per second, or one about every 23 microseconds. For this application, we need microsecond resolution, meaning that we could only view about half a second on a full screen. This is shown on the left in Figure 3.

To solve this problem, we use *folding* – several periods are overlaid on top of each other in a single strip. The color intensity at a particular pixel is the weighted average of the folded strips at that point. Thus periodic behavior shows as dense color, while aperiodic behavior results in a smear of light color. For high-frequency real-time systems where periodic behavior is crucial, if we set the amount of time

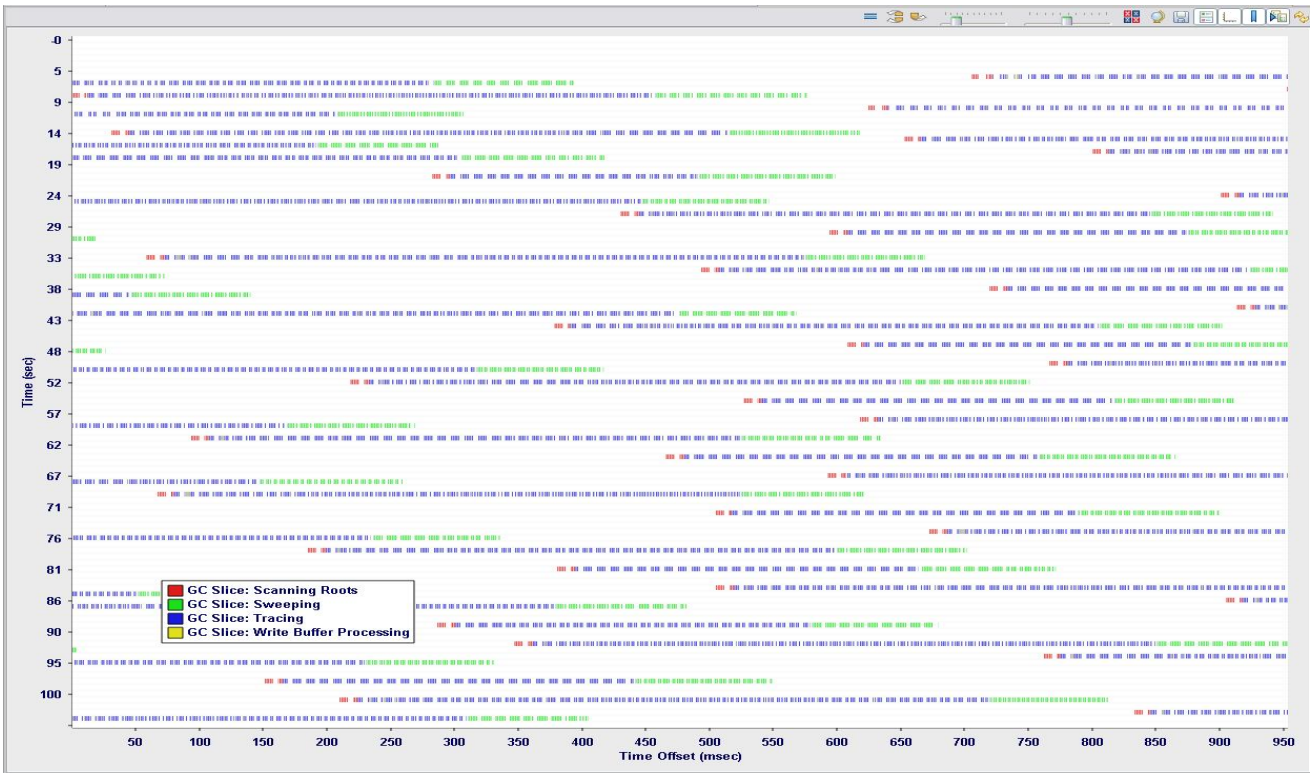


Figure 2. GC Phases Visualized in the Oscilloscope

represented by each strip to the natural period of the application (in our case 45.3515μs), then a perfectly scheduled task will display as perfectly aligned dark events. Any "smear" shows variance in the scheduling of the tasks, as is seen on the right in Figure 3, where the periods are folded 1024 times.

When the period is not known in advance, it can be varied with a slider, and it is immediately apparent when the right period has been found because the image suddenly appears "in focus". We have found folding to be effective at factors up to 1000 and more, meaning that the screen can display several minutes of execution, while still retaining microsecond timing resolution in the display. Furthermore, the system maintains interactive responsiveness at this folding level, allowing the user to move back and forth in time or to zoom in and out.

The Oscilloscope view has also proved highly effective for finding interference patterns from other periodic events. For instance, when we zoomed out on the view of the audio generation task, we noticed a periodic interruption of about 300 μs every 50 ms. This turned out to be the operating system resynchronizing the cycle counter with the lower-frequency crystal oscillator!

4 Case Studies

This section presents two case studies of how TuningFork was used for performance analysis and system understanding during the development of IBM's Real-Time Java [8]. Throughout this product's multi-year development effort, TuningFork has been a key tool for understanding the scheduling and performance characteristics of the system.

One of the key technologies in IBM's Real Time Java is the Metronome Garbage Collector [1]. Metronome is an incremental real-time garbage collector that divides the work of a single garbage collection cycle into a large number of GC quanta. The Metronome scheduling algorithm intersperses normal application execution with GC quanta to achieve a desired Minimum Mutator Utilization (MMU) [2]. As an example, the default settings in the production version of Metronome use GC quanta of 500μs, a scheduling window of 10ms, and a target MMU of 70%. Therefore, when the system is working as designed in any 10ms window of execution there will be at most 6 GC quanta, each of 500μs duration.

4.1 Unexpected GC Scheduling Decisions

One of the first unexpected discoveries we made about our production implementation of Metronome related to its

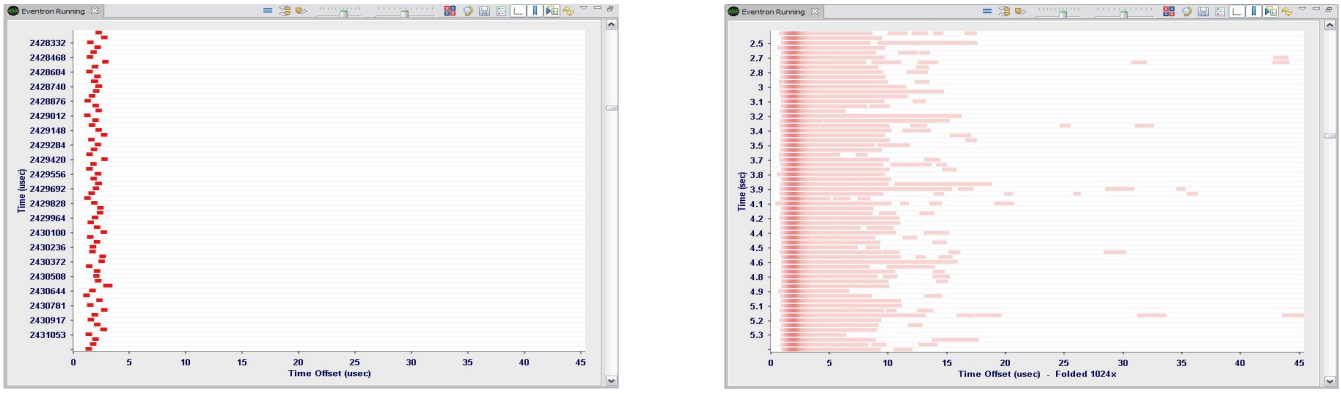


Figure 3. Folding in the Oscilloscope: each strip represents on period of the task – 45.3515μs. On the left, without folding the 50 strips show about 2.3 ms of execution; on the right, each strip is the superposition of 1024 strips, allowing visualization of over 2 seconds of data.

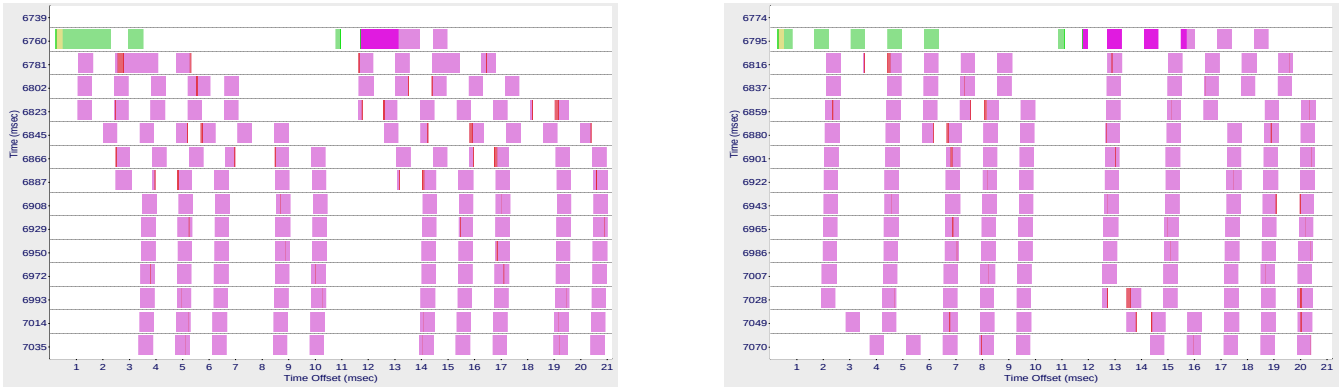


Figure 4. Oscilloscope view of unexpected (left) and expected (right) scheduling of GC quanta. Both schedules satisfy the MMU constraints (70% over a 10ms window), but the right schedule is more desirable since it minimizes irregular GC pauses.

scheduling behavior. Within the core constraint that the MMU bound must always be satisfied, the scheduler has some flexibility in how it interleaves GC and application execution. To ensure that MMU is not violated, the scheduler dynamically computes the current MMU based on a trailing window of previous scheduling decisions. Using the default parameters described above (500μs GC quanta and an MMU target of 70% measured over a 10ms window), it is safe for the scheduler to decide to schedule a GC quantum anytime that the current dynamic MMU is at least 75%. When a GC is in progress, the scheduler is invoked every 500μs to make this decision.

The first implementation of the scheduler strictly followed this MMU-centered policy. This resulted in an irregular scheduling pattern as shown on the left side of Figure 4. At the very beginning of the GC cycle, dynamic MMU was 100%, so the scheduler would proceed to schedule several GC quanta all in a row. This clumping quickly dampened as GC proceeded, but it resulted in an irregular schedule and application perceived pauses of up to 3ms. This scheduling algorithm was in use for over a year before visualizing it in TuningFork made the undesirable (but technically correct) behavior immediately obvious. We then revised the scheduling algorithm to determine the minimum number of consecutive GC quanta it needed to obtain the desired MMU target over the whole of the GC cycle (0 for MMU $> 50\%$, 1 for MMU $> 33\%$, etc), and to never schedule more than this minimum number of consecutive GC quanta no matter how much MMU slack was available. This resulted in the much more predictable scheduling behavior shown on the right side of Figure 4.

4.2 Application-Perceived Pause Times

After the initial development and performance tuning of Metronome was complete, we conducted a series of experiments to verify that Java applications were in fact obtaining the desired utilization behavior. One scenario in which Real Time Java could conceivably be deployed is as runtime underlying a Java-based transactional system. As a simple simulation of this type of workload, we used the SPECjbb2000 [15] benchmark. The goal of the experiment was to see how GC activity impacted transaction times. We used our Aspect-J based instrumentation tooling to instrument SPECjbb to demarcate the start and end of each transaction with a TuningFork event. We then ran the benchmark and collected two TuningFork trace files: one from the application and one from the JVM.

The left side of Figure 5 shows a visualization of individual transaction durations computed from the application trace, overlaid upon GC phase intervals from the JVM trace. Note that TuningFork is able to automatically correlate data from multiple trace files based on the timestamp data in the traces. An optimal result for this experiment would show a 43% increase in mean transaction time during intervals when the GC was active (during these intervals the application will only be getting 70% of the CPU). Realistically, we expect context-switching overhead and locality effects to further reduce application throughput and were expecting to see somewhere between a 50% and 100% increase in mean transaction time. However, it was immediately clear from the visualization that transaction times were increasing by 300% to 500% during the GC cycle and furthermore that they degraded more or less linearly throughout the Sweep phase of the GC (the right-most of the two shaded GC regions).

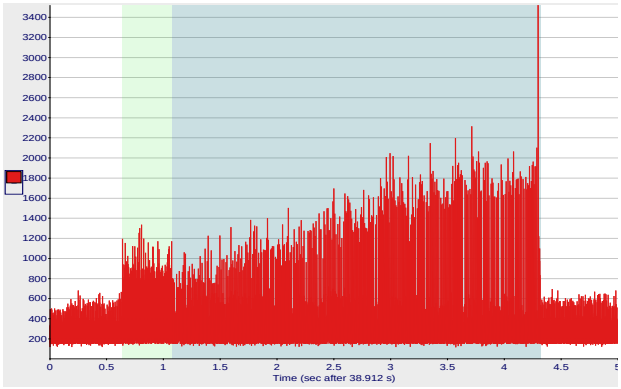
An oscilloscope visualization of the individual GC quanta of the Sweep phase (right side of Figure 5 helped us discover that as the Sweep phase of GC progressed, the quanta were gradually lengthening from the desired 500μs to 1500μs. Using TuningFork to iteratively drill down on the sub-pieces of individual GC quanta, we soon discovered that the actual core portion of each sweep quantum was ending after 480μs, but that the quantum termination code was taking progressively longer as the Sweep phase progressed. Careful examination of these functions, which comprised a very small fraction of the total GC code, revealed that at the end of each quantum, a recently added piece of statistics gathering code used to support `verbose:gc` output was incorrectly calling a function that determined precisely how many bytes of free memory was available by traversing each swept page instead of using a much faster approximate function. After changing this one function call, we were able to obtain the expected performance results, as shown in Figure 6.

The ability to collect, visualize, and analyze highly accurate profile data was critical to correctly diagnosing this problem. We needed to observe system behavior at the sub-millisecond scale without significant perturbation. We were lucky in that this problem manifested even in fairly short executions, however even these small trace files contained millions of events corresponding to all of the JVM/GC activity and hundreds of thousands of individual SPECjbb2000 transactions. Interactive visualization and cross trace file correlation were also invaluable to enable us to quickly observe the signature trend and focus our attention on the problematic code.

5 Related Work

A large body of work exists on performance visualization and analysis tools.

Kimelman et al. [9] with *PV* Program Visualizer highlights the advantages of providing instrumentation from multiple layers of the system, including hardware, the operating system, application and library code. This approach,



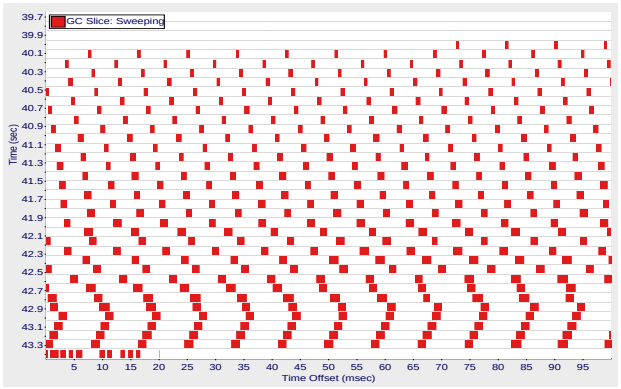
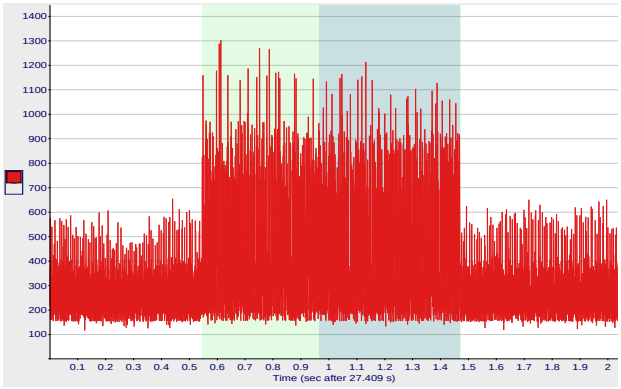


Figure 5. The left figure shows the duration of individual SPECjbb2000 transactions inμs, the shaded region indicates a GC cycle. The right figure shows an oscilloscope view of the individual GC quanta in the Sweep phase of the same GC cycle.



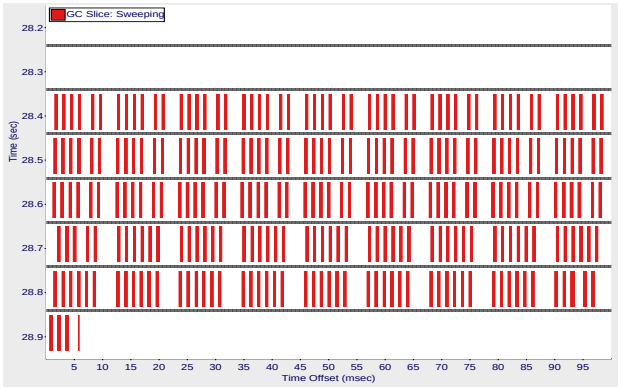


Figure 6. System performance after the GC performance bug was corrected. Transaction times now show a constant degradation during GC and GC sweep quanta are stable at 500μs.

now known as *vertical profiling*, is extended by Hauswirth et al. [6] by auto-correlating the data rather than relying on manual visual correlation.

The complexities involved in both the implementation and performance tuning of parallel and distributed systems have led to visualization techniques being embraced by that community. The *Pablo* performance analysis environment [14] is one of the more complete contributions, introducing an environment for tracing and analysis, including a self-describing trace format (SDDF), and advocates an extensible approach to visualization. *Jumpshot* [21, 19] is a tool for visualizing activity in message passing systems. It is designed for large-scale parallel computations, is built around a flexible logfile format and assists with the automatic detection of anomalous durations, drawing the user's attention to problem areas in a parallel execution.

Other visualization systems are more concerned with application profiling, where the goal is to understand where time is spent during program execution. These tools include *HPCView* [10] and *SvPablo* [3] which use a combination of hardware performance counters and sampling to hierarchically aggregate the counts and attribute them back to areas of the source code. *Jinsight* [4] is a tool designed to assist with the development of Java applications. It consists of a heavily instrumented JVM and a visualization tool. Due to the overheads involved overheads, Jinsight is unsuitable for on-line usage, although De Pauw et al. [13] show how to allow it to capture only short sections of an execution to allow some analysis of long running programs. *Paradyn* [11, 20, 12] introduces the ability to dynamically alter the active instrumentation, in addition to monitoring overheads and adjusting instrumentation based on acceptable overheads supplied as user parameters.

Significant effort has been made to ensure that TuningFork visualizations communicate information with efficiency and clarity. This effort has been guided by basic principles on the display of quantitative information – as discussed at by Tufte [18] – in addition to paying close attention to feedback from users during the development process.

6 Conclusion

TuningFork is a powerful tool for discovering timing problems in large, complex real-time systems. This problem domain is not well served by previous tools, due to a variety of inter-related requirements that require very high performance across many facets of the system: recording of events in the application, reading events into the tool, computing derived event streams, random access over traces too large to fit in memory, summarization, rapid repaint, and rapid re-computation as users dynamically alter their view of the data.

TuningFork solves these problems, and provides a rich and extensible set of visualizations for discovering and diagnosing real-time behavior. As our case studies have shown, TuningFork has proved very useful for our internal development and for other users within IBM and at selected customer sites.

With its recent public release on IBM alphaWorks, we expect to further refine the tool and expand its capabilities in response to feedback from a growing user community.

References

[1] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–298, New Orleans, Louisiana, Jan. 2003.

[2] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136. ACM Press, Jun 2001.

[3] L. DeRose and D. A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *Proceedings of the International Conference on Parallel Processing*, Fukushima, Japan, September 1999.

[4] R. K. Gary Sevitsky, Wim De Pauw. An information exploration tool for performance analysis of Java programs. In *TOOLS Europe*, March 2001.

[5] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. 17(6):120–126, June 1982. In *Symposium on Compiler Construction (SCC)*.

[6] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer. Automating vertical profiling. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 281–296, New York, NY, USA, 2005. ACM Press.

[7] IBM. TuningFork Visualization Tool for Real-Time Systems. URL www.alphaworks.ibm.com/tech/tuningfork, Aug. 2006.

[8] IBM. WebSphere Real Time product announcement. URL www.ibm.com/software/webservers/realtime, Aug. 2006.

[9] D. Kimelman, B. Rosenburg, and T. Roth. Strata-various: Multi-layer visualization of dynamics in software system behavior. In *Proceedings of the conference on Visualization '94*, pages 172–178. IEEE Computer Society Press, 1994.

[10] J. Mellor-Crummey, R. Fowler, and G. Marin. HPCView: A tool for top-down analysis of node performance. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, New Mexico, October 2001.

[11] 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, 1995.

[12] T. Newhall and B. P. Miller. Performance measurement of dynamically compiled Java executions. June 1999.

[13] W. D. Pauw, N. Mitchell, M. Robillard, G. Sevitsky, , and H. Srinivasan. Drive-by analysis of running programs. In *Workshop on Software Visualization, International Conference on Software Engineering*, May 2001.

[14] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. Schwartz, and L. F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, October 1993.

[15] Standard Performance Evaluation Corporation. SPECjbb2000 Java Business Benchmark. http://www.spec.org/jbb2000.

[16] SystemTap. SystemTap. http://sources.redhat.com/systemtap.

[17] The AspectJ Project. AspectJ. http://www.eclipse.org/aspectj.

[18] E. R. Tufte. *The visual display of quantitative information.* Graphics Press, Cheshire, CT, USA, 1986.

[19] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.

[20] Z. Xu, B. P. Miller, and O. Naim. Dynamic instrumentation of threaded applications. In *Principles Practice of Parallel Programming*, pages 49–59, 1999.

[21] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.