Understanding Complex Multithreaded Software Systems by Using Trace Visualization

Jonas Trümper
jonas.truemper@
hpi.uni-potsdam.de

Johannes Bohnet
johannes.bohnet@
hpi.uni-potsdam.de

Jürgen Döllner
juergen.doellner@
hpi.uni-potsdam.de

Hasso-Plattner-Institute, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany

ABSTRACT

Understanding multithreaded software systems is typically a tedious task: Due to parallel execution and interactions between multiple threads, such a system's runtime behavior is often much more complex than the behavior of a single-threaded system. For many maintenance activities, system understanding is a prerequisite. Hence, tasks such as bug fixing or performance optimization are highly demanding in the case of multithreaded systems. Unfortunately, state-of-the-art tools for system understanding and debuggers provide only limited support for these systems. We present a dynamic analysis and visualization technique that helps developers in understanding multithreaded software systems in general and in identifying performance bottlenecks in particular. The technique first performs method boundary tracing. Second, developers perform a post-mortem analysis of a system's behavior using visualization optimized for trace data of multithreaded software systems. The technique enables developers to understand how multiple threads collaborate at runtime. The technique is integrated into a professional and scalable tool for visualizing the behavior of complex software systems. In case studies, we have tested the technique with industrially developed, multithreaded software systems to understand system behavior and to identify multithreading-related performance bottlenecks.

Categories and Subject Descriptors

K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software Maintenance*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

General Terms

Design, Documentation, Performance

SOFTVIS'10, October 25–26, 2010, Salt Lake City, Utah, USA.

Keywords

Dynamic Analysis, Multithreading, Visualization, Program Comprehension, Performance Optimization

1. INTRODUCTION

A large fraction of costs in a software system's life cycle is spent on its maintenance [9]. A survey performed by Erlikh [14] reports an estimate of 85-90%. Common maintenance tasks include bug fixing and performance optimization. Locating and fixing bugs in software systems has always been tedious work for developers [28, 47]. The most common debugging workflow is as follows: (1) Form a hypothesis about the root cause of the bug, (2) try to fix this cause, (3) check whether the bug still exists, (4) if yes, return to (1), else (5) verify that the fix has not caused other interferences [30]. Forming a hypothesis is commonly supported by debuggers in that they provide a simple means of dynamic analysis: They allow stepping through a system's execution and support monitoring of its variable values.

Although debuggers and profilers are effective tools for debugging and performance optimization of single-threaded software systems, there are issues unique to multithreaded software systems that are insufficiently supported by today's techniques and tools [10]. These issues include deadlocks, load imbalance, data-sharing patterns, race conditions, and contention. Timing and scheduling, which have to be considered in concurrent systems, are mostly ignored by existing tools. In addition, it is difficult for developers to assess the concurrent runtime behavior, which is typically not directly reflected by the source code. That is, there is only a non-obvious mapping between control structures in the source code and a system's runtime behavior.

Moreover, the size of a system's implementation and software aging play important roles in maintenance. Hence, aged systems are often insufficiently understood and an up-to-date documentation is typically not available. Furthermore, there are substantial violations of design principles due to the system's long evolution period and its modification by a constantly changing team of developers [13, 36]. As Waters and Chikofsky [46] state: "Year after year the lion's share of effort goes into modifying and extending preexisting systems, about which we know very little".

To overcome the lack of appropriate tool support for understanding multithreaded systems, developers have 'invented' workarounds to ease the forming of hypotheses. For instance, they often manually augment the source code with statements that print debug output to the console [21, 43]. Despite this being a time consuming and tedious task,

dumping output to the console is a resource intensive operation that may cause noticeable performance drop or even alter the system's timing significantly. Thus, developers try to minimize usage of console output and apply guesswork instead. Consequently, mentally tracking the execution of a multithreaded software system is a highly demanding and additionally error-prone task. With increasing importance and market share of multi-core machines and multithreaded applications [16, 44], tools that better support maintenance of multithreaded software systems are more in demand.

The main contributions of this paper are (1) a visualization concept that applies out-of-core techniques to cope with the vast amount of trace data and that is suitable to display large traces of multithreaded software systems and (2) a technique that enables synchronization of multiple separate compacted views on trace data of multithreaded software systems. The views are compacted using a (nonlinear) mathematical time transformation. The level of compactness can continuously be changed from (a) highly compact (strong nonlinearity) to make optimal use of available screen space to (b) linear time-to-screen-space mapping to reveal performance issues such as long waiting times.

The proposed technique selectively instruments binaries of the analyzed C/C++ software system to record execution traces at runtime. These traces are post-processed and imported into the analysis tool. The proposed visualization technique generates stack based, synchronized views on sequences of method calls[1] for multiple threads. The visualization permits developers to interactively explore such traces. The tool has been tested with software systems consisting of over 4 million lines of code (LOC). Its usage and benefits are demonstrated by means of case studies with one of our industrial partners and with the Google Chrome code base.

The concept for visualizing the behavior of multithreaded software systems is implemented as prototypical plug-in as part of the *Software Diagnostics Developer Edition*—a professional tool for tracing and visualizing runtime behavior of complex software systems.

2. RELATED WORK

Since the early 1970s, multithreaded computing has evolved from being a niche technology to being a technology for the masses [8, 35]. Toolkits such as Qt [38], OpenMP [8], boost [22], or Intel's Threading Building Blocks [39] provide abstraction mechanisms to reduce the complexity that is inherent to writing multithreading code. Furthermore, a large body of research exists in the modeling and forward engineering community to cope with parallel systems. Momotko (2003) [32] proposed extensions to the Business Process Modeling Notation (BPMN) such that it allows for the modeling of parallel business processes. Among others, Mehner (2005) [31], Artho et al. (2007) [1], and Xie et al. (2009) [49] propose extensions to the Unified Modeling Language (UML) to enable modeling of concurrent behavior.

Tool support for understanding existing multithreaded systems, however, is still limited. Cornelissen et al. [10] emphasize that "the importance of understanding multithreading behavior [...] is currently not reflected" in the dynamic analysis research community. Early work on the analysis of multithreaded runtime behavior was done by Malony and Reed (1989) [29]. Their approach focuses on monitoring systems and collecting statistics such as communication bandwidth. Heath and Etheridge (1991) [19], Nutt et al. (1995) [34], Nagel et al. (1996) [33] and Zaki et al. (1999) [51] propose tools that aim at performance optimization of parallel message passing systems. Sharma (1990) [40] introduces real-time visualization for multi-core machines, targeting specific hardware. Visualization is done on a dedicated machine connected to the machine running the instrumented software system. Yamaguchi and Itoh (2003) [50] propose a visualization similar to treemaps that depicts overviews of distributed processes to ease management of distributed systems. Hao (1998) [18] introduces SmallSync, a tool for monitoring and visualizing remote processes. In contrast to our approach, these approaches focus on analyzing interaction between processes, often running on separate machines.

Stasko and Kraemer (1993) [42] introduce PARADE that targets program comprehension and performance optimization. Their tool visualizes multiple threads in an isolated and not synchronized way. This renders it difficult for developers to understand interactions between the threads. Kraemer and Stasko (1994, 1998) [25, 26] further introduce Animation Choreographer, a tool that permits to explore thread interactions in a synchronized manner. However, it uses different shapes to distinguish methods and only depicts each threads' state instead of their stacks. Kergommeaux and Stein (2000) [23], Bedy et al. (2000) [2], and Berthold and Loogen (2007) [5] propose visualization that depicts 2-dimensional graphs with life lines for each thread showing their state (running, suspended etc.). This graph visualization, by contrast, does not show the threads' actual execution contexts or call stacks. Broberg et al. (1999) [6] introduce VPPB, a tool for visualizing and predicting performance of multithreaded systems by means of kernel thread tracing. Their approach differs from ours in that it does not support program comprehension tasks and requires instrumentation of the host operating system.

De Pauw et al. (2002) [12] propose Jinsight, a tool similar to ours. However, it does not provide means of view compaction or overviews to ease analysis. TAU, proposed by Shende and Malony (2006) [41], is a framework for profiling and tracing of parallel systems that focuses on performance optimization rather than on program comprehension. Wheeler and Thain (2009) [48] propose ThreadScope, a tool for identifying structural and synchronization problems. A limitation of the tool is that the generated 2-dimensional graphs do not scale well for large execution traces. Furthermore, and by contrast to our approach, there is potential for interactive exploration that would in turn allow for enhancing their tool's scalability. Kim et al. (2009) [24] introduce a visualization that targets uncovering of potential deadlocks in multithreaded systems. Program comprehension, in comparison to our approach, is not in the focus. George and Nagpal (2010) [17] propose Concurrency Visualizer, which is a visualization technique integrated into Microsoft Visual Studio 2010. It is based on profiling samples and depicts all threads' states (running, waiting, synchronizing etc.) along the recorded time span. In contrast to our approach, exploration of a thread's execution context is restricted to selecting a thread's sample and then manually expanding its recorded stack in a list view. The tool Zinsight, introduced by De Pauw and Heisig (2010) [11], targets special mainframe software and hardware. In con-

[1]The term method is used interchangeably with the terms function, procedure, routine, etc.

trast to visualizing control flows in a single software system, their visualization is tuned towards analysis of isolated events and event patterns recorded in an operating system context.

For our approach to achieve configurable level of compactness of trace views, we apply nonlinear mathematical operations. Nonlinear mathematics is used in a wide range of research domains. Examples include human behavior [15, 27] and self-adaptive systems [3, 20].

3. THE APPROACH

The proposed approach for facilitating an understanding of the behavior of multithreaded software systems comprises two main analysis steps: (1) Applying dynamic analysis and tracing the software system. (2) Applying visualization to permit developers to gather insights into the vast amount of extracted data.

3.1 Dynamic Analysis

Before the runtime behavior of the analyzed software system can be captured, the system is instrumented. We apply a light-weight instrumentation technique provided by the *Software Diagnostics Developer Edition*'s framework. The technique is seamlessly integrated into the build process. However, it neither increases the build time nor affects the runtime behavior if tracing is disabled. Tracing can be interactively activated and deactivated for selected methods and modules of the software system at runtime. When tracing is enabled, a *trace* is recorded and stored persistently.

3.2 Trace Data Properties

The recorded trace comprises events e with timestamps i, i.e., *timestamped events*, e_i. Each event is associated with a thread x. All events of a thread x form an ordered set: Event e_{xi} happens before e_{xj} if $i < j$. The combined set of events of multiple threads (x, y) forms a partially ordered set: e_{xi} happens before or is concurrent to e_{yj} if $i \leq j$. Events are typed: They are either of type *method entry* or *method exit*. Furthermore, each event comprises two code addresses: (1) The address of the code line of method f that is the calling method and (2) the address of the start code line of the called method.

An event's timestamp has to be unique with respect to the event's thread; timing information needs to be provided with high precision to be able to distinguish timing behavior across multiple threads. Thus, we use processor ticks [37] as measure for event timestamps. On modern multi-core systems that support variable clock speeds per core, processor ticks may drift between cores [7]. In this paper, we assume that processor ticks are in sync.

3.3 Minimizing Runtime Overhead

In general, tracing a software system means introducing a runtime performance overhead. If timing is critical, this overhead may alter the system's runtime behavior unintentionally. Multithreaded software systems are particularly prone to this issue because deadlocks or race conditions, for instance, may be caused or prevented if timing is changed. To minimize the likelihood of such interferences, we provide developers with means to instrument only those parts of the system's implementation that are relevant for the functionality to be debugged: They initially run the system with full instrumentation and record a trace. Next, they explore

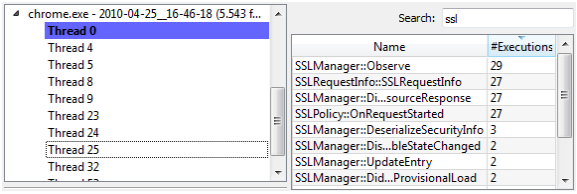


Figure 1: Textual thread overview depicting statistics on a single thread's activity.

this first trace and gain an understanding of which parts of the system's implementation are relevant for the given task. Particularly, they identify low-level methods that have short execution times and are often contained in the trace, but do not contribute to the process of understanding. The developers subjectively decide which parts of the implementation are excluded from further tracing. Finally, they exercise the functionality again with only selective instrumentation. This approach permits to obtain a low performance overhead—even with a costly event registration mechanism: The methods with short execution times are excluded from tracing; hence, the performance overhead is only added to more costly calls which results in a lower *relative* and therefore lower *absolute* overhead.

Our experience shows that this approach works well for maintenance questions regarding our industrial partner's software systems. A limitation of this approach is its limited applicability for small sized or highly time critical systems. Examples of such systems include specific embedded real-time systems in the automotive domain.

4. VISUALIZATION

A software system instrumented for method boundary tracing easily generates a million events and more within a few seconds. Consequently, a developer needs effective visualization that presents the trace data in an interactive manner and permits to apply top-down and bottom-up exploration strategies [4].

4.1 Concept

For our visualization technique to scale with large execution traces comprising millions of events, we apply out-of-core concepts: Only a subset of all recorded method invocations is kept in main memory and is depicted as detailed sequence.

Besides millions of events, in multithreaded software systems numerous threads may be spawned at runtime. To facilitate analysis of this behavior, developers should be able to choose a representative subset of all spawned threads. For this, our analysis tool provides a *textual thread overview* that depicts the activities of selected threads, i.e., lists methods invoked in the context of selected threads together with their invocation count (Fig. 1). This way, developers identify and select representative threads for visualization. For our technique, we assume that the total number of threads can be reduced to a small number for a given task; so far we did not evaluate the visualization technique for massive threads because our main focus is on applications using threads in a non-massive way.

Each selected thread's trace data is depicted in a *visual thread overview* and a *sequence view*. These views are syn-

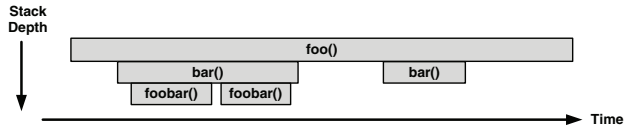


Figure 2: Concept for the sequence visualization for a single thread.

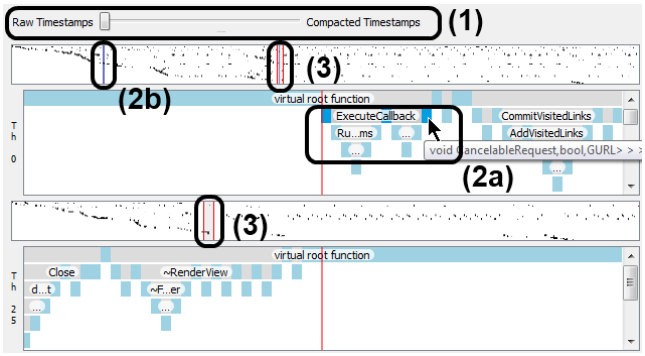


Figure 3: Sequence visualization shows 2 threads. A slider enables to configure timestamp scaling (1). Activity markers highlight invocations of methods hovered in the sequence visualization (2a) within each visual thread overview (2b). The detailed depicted sub time span (in the sequence visualization) is denoted by time span markers in the respective visual thread overview (3).

chronized, which allows developers to efficiently analyze the threads' concurrent behavior. The basic visualization concept for these two view types is as follows: In the sequence views, each thread's activity is represented by a 2-dimensional graph. The execution sequences are depicted such that developers see the call stack for every point in time. Time is mapped along the graph's x-axis and stack depth along the y-axis (Fig. 2). Invocation relations between methods are thereby given implicitly. That is, if method *foo* invokes *bar*, then *bar* is drawn underneath *foo*. Since even a single thread's complete sequence graph typically exceeds a screen's size, panning and zooming are provided so that developers can adjust the currently depicted time span as needed. Zooming only affects the time (x) axis such that the graph is compressed (or stretched) in x-direction, i.e., the aspect ratio of the sequence graph changes. Consequently, stack height (y-axis) is preserved, which in turn also preserves the readability of method annotations.

Visual thread overviews facilitate quick orientation within the trace data. *Time span markers* denote the current detailed sequence sub range within the respective visual thread overview, *activity markers* denote invocations of methods hovered in the sequence visualization (Fig. 3). As Ware [45] points out, "the human visual system is a pattern seeker of enormous power". To utilize this power, the visual thread overviews derive visual patterns from execution patterns. The overviews are generated as follows: We compute a 2-dimensional grid where methods are mapped along the grid's y-axis and time is mapped along the x-axis. When a method is executed at a specific point in time, the respective grid cell is colored black, white otherwise. This way, repeated execution of the same (or similar) functionality causes repeating visual patterns in the overview (Fig. 4). The resulting grid is then mapped to a fixed size representation such that it fits the screen's size.

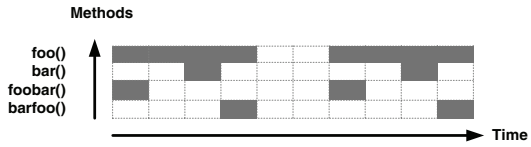


Figure 4: Concept for the visual thread overview: Multiple execution of the same (or similar) functionality is represented by repeating visual patterns.

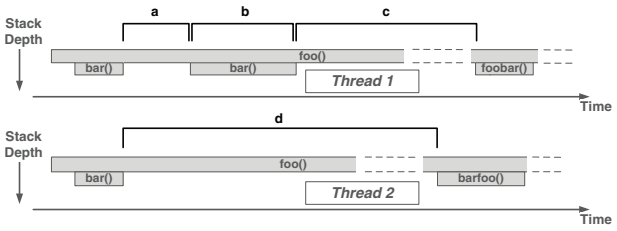


Figure 5: Raw time stamps. Execution durations of methods, e.g., *foo()* and *bar()*, vary significantly.

4.2 Configurable Level of View Compactness

Execution durations of methods typically vary significantly: Whereas execution of method *foo* takes 20,000 milliseconds on average, the execution of method *bar* may take only a few nanoseconds on average. Even for depicting a single thread's activity, this poses a major challenge: When depicting these raw execution durations, lengthy method executions would span multiple screens while very short method executions might span only a single pixel (Fig. 5). Consequently, a time warping—a *scaling*—is required for exploration.

We provide configurable scaling for time spans Δt between successive timestamps t_i, t_j. The scaling is either linear or logarithm-based. Linear scaling enables to analyze raw execution time stamps and as such is suited for performance analysis and optimization tasks. Linear scaling, however, hinders program comprehension tasks as developers typically have to explore execution traces at low detail (zoomed out) in order to grasp the context of the currently depicted method executions (Fig. 6).

In contrast, logarithm-based scaling is better suited for program comprehension tasks, as short time spans are shrinked only slightly while long time spans are shrinked massively. Hence, idle times between method executions typically 'vanish' such that relevant parts of the trace can be explored in more detail. To achieve a configurable logarithmic scaling, we do not use the plain natural logarithm. Instead, we multiply Δt with a factor and divide the result of the logarithm by the same factor to achieve configurable shrinking. This is motivated by the approximation of the natural logarithm by a Taylor series:

$$\ln(1+\Delta t) = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{\Delta t^k}{k} = \Delta t - \frac{\Delta t^2}{2} + \frac{\Delta t^3}{3} - \frac{\Delta t^4}{4} \pm \cdots$$

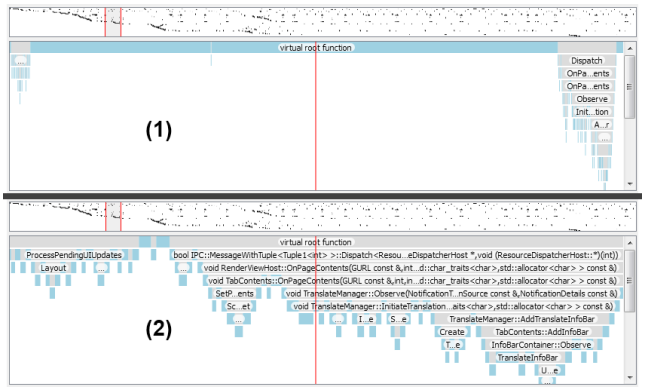


Figure 6: Sequence view depicting the same sequence sub range using linear (1) and logarithmic (2) scaling. Linear scaling requires to zoom out (low detail) due to long thread idle time. The same set of method executions can be depicted in more detail using logarithm-based scaling.

The scaling function then reads as:

$$\begin{aligned} logscale(\Delta t, \theta) &= \frac{\ln(1+\theta\cdot\Delta t)}{\theta} \\ &= \frac{\sum_{k=1}^{\infty}(-1)^{k+1}\frac{(\theta\cdot\Delta t)^k}{k}}{\theta} \\ &= \frac{\theta\cdot\Delta t}{\theta} - \frac{\theta^2\cdot\Delta t^2}{2\theta} + \\ &\quad \frac{\theta^3\cdot\Delta t^3}{3\theta} - \frac{\theta^4\cdot\Delta t^4}{4\theta} \pm \cdots \\ &= \Delta t - \frac{\theta\cdot\Delta t^2}{2} + \\ &\quad \frac{\theta^2\cdot\Delta t^3}{3} - \frac{\theta^3\cdot\Delta t^4}{4} \pm \cdots \end{aligned}$$

Whereas θ cancels out in the first term of the power series, all subsequent terms contain θ. Hence, shrinking of Δt can essentially be adjusted continuously between linear *and* logarithmic (Fig. 7) as for $\theta \to 0$ all terms but the first are eliminated:

$$\lim_{\theta\to 0}\frac{\ln(1+\theta\cdot\Delta t)}{\theta} = \Delta t$$

In contrast to that, for $\theta > 0$, all subsequent terms account for the power series and scaling is logarithm-based.

4.3 Synchronizing Multiple Sequence Views

Applying the nonlinear scaling *logscale* separately to each thread's events would hinder the visual comparability of concurrent method executions: Method executions or idle times in separate threads (time spans a, b, c and d in Fig. 5) that happen before the execution of *foobar* and/or *barfoo* may be scaled differently (Fig. 8(a)) such that method entry and exit timestamps of *foobar* and *barfoo* are warped. In other words, the partial order of scaled timestamped events (across multiple selected threads) may differ from the order of unscaled timestamped events. Thus, synchronization would fail to provide usable visual results.

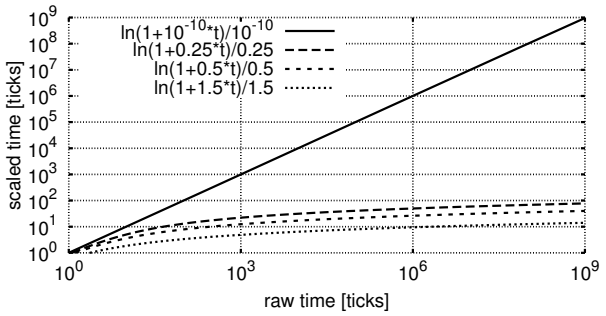


Figure 7: Plots of *logscale* for different values of θ: The plot for $\theta = 1\cdot 10^{-10}$ is already linear for Δt in $[1, 10^9]$. Larger values of θ, e.g., $\theta = \frac{1}{4}$ and $\theta = \frac{1}{2}$, allow for logarithm-based shrinking of Δt.

Consequently, a key challenge is the way to synchronize multiple views—each depicting a single thread's activity—such that collaboration between threads gets visible. In the absence of effective synchronization, developers would need to compare event timestamps manually to assess whether a specific method execution in the context of thread i is actually concurrent to another method execution in the context of thread j.

To address this issue and to preserve visual comparability of concurrent method executions, we apply on-demand scaling for the selected threads and the depicted sub time span. That is, we merge all events of all selected threads into a single queue (Fig. 8(b)). Concurrent events (having the same timestamp) are allowed to be enqueued at the same place within the queue. Thus, we can safely scale each Δt of successive events in the queue whilst preserving partial event order across all selected threads (Fig. 8(c)).

Our approach both scales time and synchronizes multiple views by focusing on a specific point in time in the trace and by updating all views such that the views are centered around this point in time.

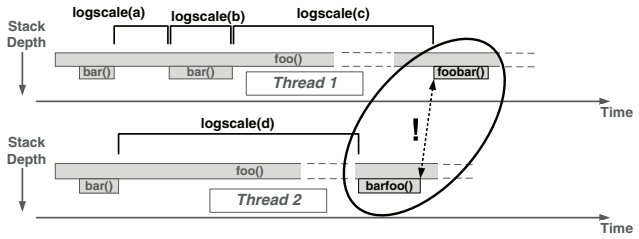
5. CASE STUDIES

We discuss two case studies that we have performed on industrially developed, multithreaded software systems: (1) *Chromium*[2], the open-source code base of Google's web browser Chrome and (2) *Building Reconstruction* (BRec), a tool for reconstructing 3D building models from laser scan data from virtualcitySYSTEMS GmbH[3], one of our industrial partners.

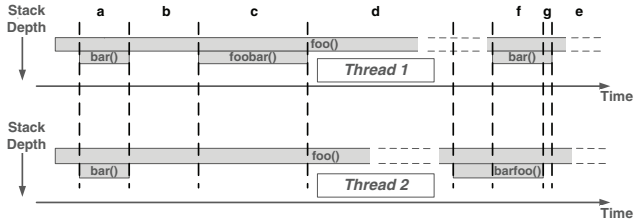
5.1 Chromium: Bookmark Search

Chromium consists of approximately 4 million lines of code[4], thereof approximately 2.7 million LOC in C and C++. According to the source code repository, more than 390 authors contributed to the code base. The implementation concepts of Chromium strongly rely on deferred processing, e.g., tasks are 'posted' by one thread and successively processed by another dedicated thread. Hence, developers

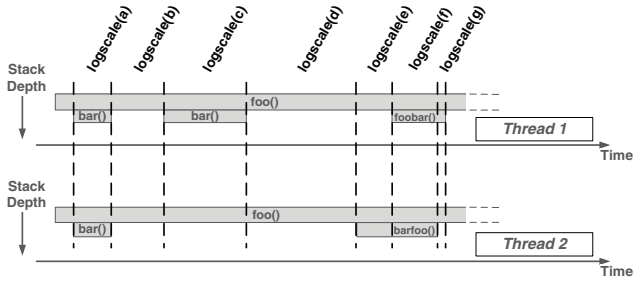
[2] `http://www.chromium.org`, last accessed 04/28/2010
[3] `http://www.virtualcitysystems.de`, last accessed 04/28/2010
[4] `http://www.ohloh.net/p/chrome/analyses/latest`, last accessed 04/28/2010



(a) Logarithmic scaling per thread. The execution of *foobar()* in thread 1 is no longer depicted as concurrent to the execution of *barfoo()* in thread 2.



(b) Partial order-preserving scaling: All selected threads' timestamped events are merged into a single queue. Timestamp differences (a, b, c, ...) are now calculated based on the merged queue.



(c) Logarithmic scaling based on the merged queue ($logscale(a), logscale(b), ...$): Visual comparability of *foobar()* and *barfoo()* is preserved.

Figure 8: Preserving visual comparability with logarithmic scaling.

concerned with fixing performance weaknesses in Chromium likely face understanding problems that also occur during development of complex closed-source multithreaded systems.

In this case study, we aim to identify the main bottleneck in Chromium's bookmark search: Searching for popular terms takes considerably more time than expected. Our initial suspicion is that the database query could be responsible for the slowdown. To verify this, we identify relevant parts of Chromium's implementation (Section 3.3) and rerun the scenario with selective instrumentation. We exercise the bookmark search scenario and import the resulting trace into our tool.

Using the textual thread overview, 3 out of 10 threads turn out to be relevant for our analysis task: Searching for 'bookmark' in the textual thread overview yields hits in one thread (Fig. 9). Two other threads are identified as relevant as they seem to be concerned with deferred processing of jobs.

We initially seek through the trace data by using the zoom and pan facilities of the sequence view with logarithm-based scaling. We identify key points for further inspection in the trace by remembering the respective patterns that are enclosed by time span markers in the visual thread overview. Subsequently, we issue a search for the term 'bookmark' in our tool and yield 9 hits. In contrast to that, a search for the same search term in Chromium's source code results in approximately 19,000 hits in more than 800 files. One of the hits in the trace data is *BookmarksFunction::Run*. As that draws our attention, we inspect it first, tuning the mathematical scaling operation to linear.

It turns out, that the respective method's invocation actually issues a query in the bookmark database (Fig. 10): (1) *BookmarksFunction::Run* calls lower-level functionality to process the database query. Among others, *ExtractQueryWords* is called. After the database query returns, (2) *BookmarksFunction::Run* invokes *SendResponse* in class *Async-*

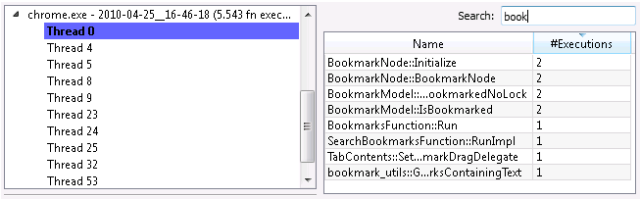


Figure 9: Textual thread overview: Identifying relevant threads in Chromium.

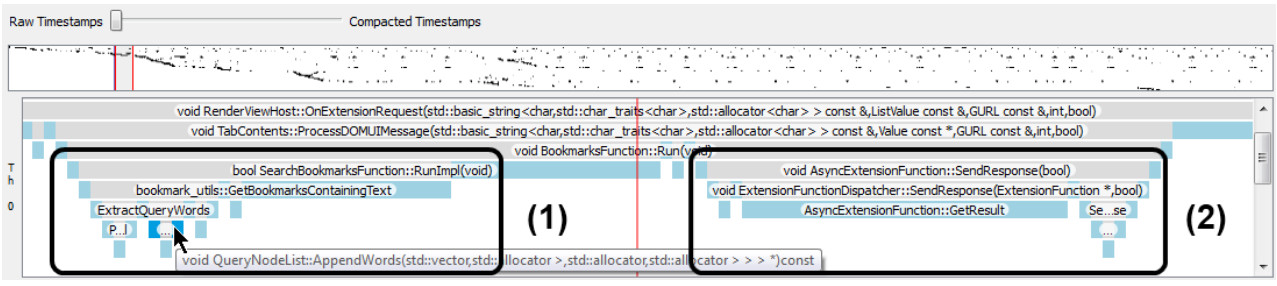


Figure 10: Chromium bookmark search in detail, scaling tuned to linear: Database query (1) and result dispatching (2).

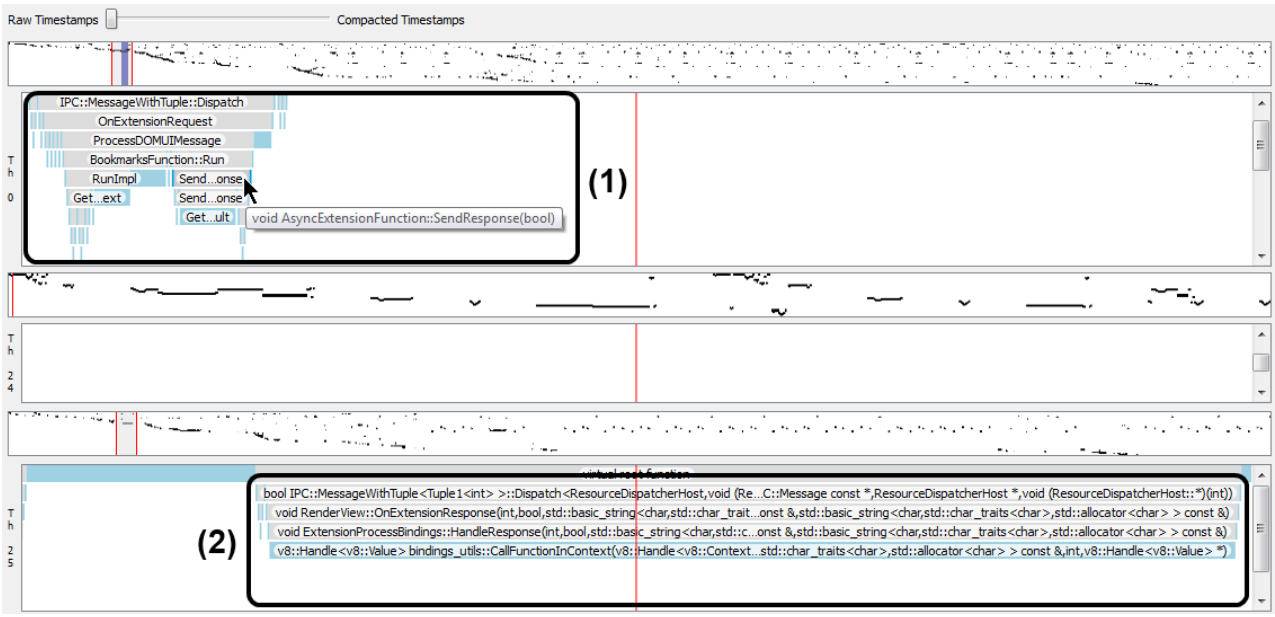


Figure 11: Chromium bookmark search and result dispatching (1) (see Fig. 10) as well as asynchronous result handling (2). Scaling is tuned to linear.

ExtensionFunction to enqueue the result processing job. Processing of the search result is done asynchronously in another thread (Fig. 11): After the database query is finished and the job enqueued (1), *RenderView*'s method *OnExtensionResponse* is invoked via an IPC to process the query result. *OnExtensionResponse* hands result processing over to *ExtensionProcessBindings::HandleResponse* that subsequently calls some JavaScript functionality in Chromium's JavaScript engine V8.

In contrast to our initial expectations, the following facts were observed: First, the database query is not the performance bottleneck. Processing the result in *OnExtensionResponse* is. It consumes more than thrice the time of the database query. Further, query execution and result processing, while being executed in separate threads, are actually executed sequentially. With parallelized querying and result processing, processor utilization on multicore systems could be increased and first search results could become visible in less time. Second, only 2 of 3 chosen threads were actually relevant for our comprehension task (threads 0 and 25). The third thread (24) did not execute any relevant functionality.

5.2 BRec: Rendering Process

The software system of virtualcitySYSTEMS GmbH *BRec* reconstructs and visualizes 3D building models from raw laser scan data. Its rendering engine triangulates the building models in parallel with multiple threads. BRec's development started more than 10 years ago. The code base comprises approx. 100k LOC written in C/C++ with 15 developers on average working on it.

In this case study, a developer is concerned with speeding up the rendering process. The developer starts with recording a trace of the complete rendering functionality. Using the textual thread overview, he identifies one coordinating thread and a vast amount of worker threads that all execute the same utility functionality: Delaunay triangulation.

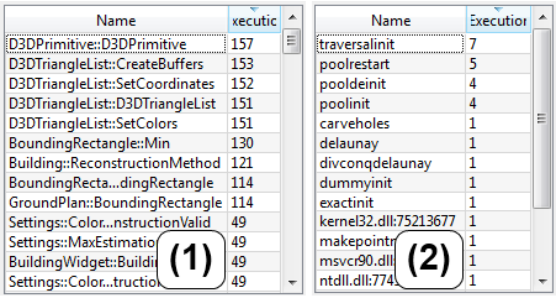


Figure 12: Textual thread overview: (1) BRec's coordinator thread and (2) a worker thread.

Identification of the coordinating thread is based on two facts: (1) It executes the main window's event loop and (2) it exhibits significantly higher invocation counts than the other threads (Fig. 12). The developer selects the coordinating thread and two worker threads as the worker threads execute the same methods with the same invocation counts.

The developer explores the trace using logarithm-based scaling to gain an initial overview of the rendering functionality and of how the coordinating thread handles worker threads. The first observation is that there is one coordinating method, which is responsible for the whole rendering process: *BuildingWidget::UpdateRenderer*. It creates instances of class *SolidRenderer* that each instantiate *Triangulation3D2*. The developer notices that *Triangulation3D2* spawns two worker threads that calculate a triangulation (Fig. 13). Each worker threads' time span marker spans the thread's complete lifetime in the visual thread overview. This shows that the thread terminates immediately after finishing triangulation and complements the initial observation, which caused the developer to classify them as worker threads.

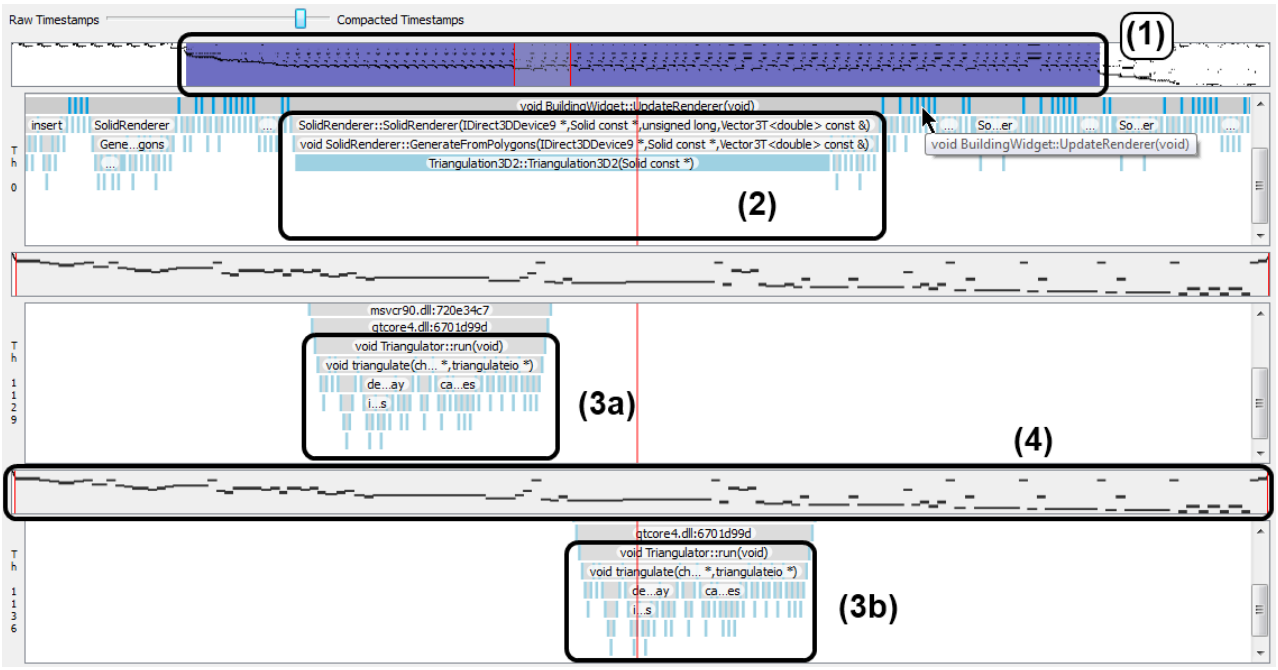


Figure 13: BRec's rendering engine: (1) Method *UpdateRenderer* of class *BuildingWidget* coordinates the rendering process. The coordinator thread (2) spawns two worker threads (3a, 3b), each calculating a triangulation. (4) Activity markers of threads 1136 and 1129 indicate that the sequence views (below) depict the threads' complete activity.

As a worker thread's sole purpose is to calculate a single Delaunay triangulation, a significant number of threads is spawned during the rendering process. Although thread creation is considered to be cheap in comparison to process creation, its overhead cannot be neglected. Hence, the developer decides to use a thread pool for triangulation jobs to speed up rendering by reducing thread creation costs.

A surprising result of this case study was that the performance issue could be located without needing to explore the trace with linear scaling, i.e., exploring the raw timestamps that show "real" waiting times. Different non-linearity grades of the logarithmic scaling were sufficient to understand the multithreading behavior and to locate the main performance bottleneck in the rendering process.

6. CONCLUSIONS

Maintenance and debugging as well as performance tuning are challenging and time-intensive tasks, especially in the case of multithreaded software systems. Generally, such tasks rely on understanding the concurrent control flows in the different threads' contexts. Furthermore, a single feature may not only be distributed across multiple implementation units but may be executed by multiple threads. A lack of appropriate tools makes understanding the behavior of such systems a costly work.

We have presented a visualization technique that supports developers in understanding multithreaded system behavior, i.e., developers can better understand interactions between threads and composition of functionality across multiple threads. System behavior can be explored on various levels of detail, which permits developers to perform top-down exploration of execution traces. In particular, the execution time spent in the different parts of a system's implementation becomes visible.

Our technique, as dynamic analysis technique, exploits that the developer's search space to understand system behavior is reduced to those parts of the implementation that are relevant for a given maintenance task. Thus, even developers with little knowledge of the system's implementation are able to quickly identify relevant parts of the system's implementation and understand their interactions. We have demonstrated our approach and shown that it scales with large software systems by case studies. The technique has been tested with two large-scale and multithreaded software systems; performance weaknesses in these systems and non-obvious system behavior have been revealed.

As future work, we plan to extend the visualization to support user-configurable elision and aggregation of parts of a call sequence to improve analysis of large traces. Moreover, we want to investigate how to (1) deliminate equal distances in time in non-linear compacted views at all zoom levels and (2) handle deep stacks that may occur due to recursion. Likewise, we plan to investigate how to improve screen-space usage such that a higher number of threads can be depicted in parallel.

Further future work includes (1) extending the tracing technique such that different threads' accesses to shared memory are recorded and (2) subsequently adapting the visualization according to this additional data so that communication events between threads can be shown. Next, we plan to perform controlled experiments to be able to quantify the increase of developer performance when using our approach to understand multithreaded software systems.

7. ACKNOWLEDGMENTS

We want to thank Software Diagnostics GmbH (`http://www.softwarediagnostics.com`) for providing us with their tracing and visualization framework. In addition, we want to thank virtualcitySYSTEMS GmbH for providing us with their BRec software system.

8. REFERENCES

[1] C. Artho, K. Havelund, and S. Honiden. Visualization of concurrent program executions. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 541–546, Washington, DC, USA, 2007. IEEE Computer Society.

[2] M. Bedy, S. Carr, X. Huang, and C.-K. Shene. A visualization system for multithreaded programming. *SIGCSE Bulletin*, 32(1):1–5, 2000.

[3] M. Beetz and H. Grosskreutz. Probabilistic hybrid action models for predicting concurrent percept-driven robot behavior. *Journal of Artificial Intelligence Research*, 24(1):799–849, 2005.

[4] C. Bennett, D. Myers, M.-A. Storey, and D. German. Working with 'monster' traces: Building a scalable, usable sequence viewer. In *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis*, pages 1–5, 2007.

[5] J. Berthold and R. Loogen. Visualizing parallel functional program runs: Case studies with the eden trace viewer. In *Proceedings of the International Conference Parallel Computing*, pages 121–128, 2007.

[6] M. Broberg, L. Lundberg, and H. Grahn. Visualization and performance prediction of multithreaded solaris programs by tracing kernel threads. *Parallel Processing Symposium, International*, 0:407–413, 1999.

[7] R. Brunner. Tsc and power management events on amd processors. Technical report, AMD Corporation, 2005.

[8] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. Scientific and Engineering Computation. The MIT Press, 2007.

[9] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[10] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[11] W. De Pauw and S. Heisig. Visual and algorithmic tooling for system trace analysis: a case study. *SIGOPS Operating Systems Review*, 44(1):97–102, 2010.

[12] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, London, UK, 2002. Springer-Verlag.

[13] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[14] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[15] A. Foster. A nonlinear model of information-seeking behavior. *Journal of the American Society for Information Science and Technology*, 55(3):228–237, 2004.

[16] M. Franklin. *The Computer Engineering Handbook: Digital Systems and Applications (Second Edition)*, chapter Multithreading, Multiprocessing, pages 35–51. CRC Press, 2008.

[17] B. George and P. Nagpal. Optimizing parallel applications using concurrency visualizer: A case study. Technical report, Microsoft Corporation (Parallel Computing Platform Group), 2010.

[18] M. C. Hao, D. Glajchen, and J. S. Sventek. Smallsync: A methodology for diagnosis visualization of distributed processes on the web. Technical report, Hewlett Packard, 1998.

[19] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8:29–39, 1991.

[20] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[21] P. Horwood, S. Wygodny, and M. Zardecki. Debugging multithreaded applications. *Dr. Dobb's Journal of Software Tools*, 25(3):32, 34–37, March 2000.

[22] B. Karlsson. *Beyond the C++ Standard Library*. Addison-Wesley Professional, 2005.

[23] J. C. D. Kergommeaux, B. D. O. Stein, and M. S. Martin. Pajé: An extensible environment for visualizing multi-threaded program executions. In *European Conference on Parallel Computing*, pages 133–144, 2000.

[24] B.-C. Kim, S.-W. Jun, D. J. Hwang, and Y.-K. Jun. Visualizing potential deadlocks in multithreaded programs. In *Proceedings of the 10th International Conference on Parallel Computing Technologies*, pages 321–330, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] E. Kraemer and J. T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations. In *Proceedings of International Parallel Processing Symposium*, pages 902–908, 1994.

[26] E. Kraemer and J. T. Stasko. Creating an accurate portrayal of concurrent executions. *IEEE Concurrency*, 6(1):36–46, 1998.

[27] R. Libby. Man versus model of man: the need for a nonlinear model. *Organizational Behavior and Human Performance*, 16:23–26, 1976.

[28] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2002.

[29] A. D. Malony and D. A. Reed. *Instrumentation for future parallel computing systems*, chapter Visualizing parallel computer system performance, pages 59–90. ACM, 1989.

[30] S. McConnel. *Code Complete 2: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

[31] K. Mehner. *Trace-based Debugging and Visualisation of Concurrent Java Programs with UML*. PhD thesis, Universität Paderborn, 2005.

[32] M. Momotko and B. Nowicki. Visualisation of (distributed) process execution based on extended

bpmn. *International Workshop on Database and Expert Systems Applications*, 0:280–286, 2003.

[33] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996.

[34] G. Nutt, A. Griff, J. Mankovich, and J. McWhirter. Extensible parallel program performance visualization. *International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, 0:205–211, 1995.

[35] V. G. Oklobdzija, editor. *The Computer Engineering Handbook: Digital Systems and Applications (Second Edition)*. CRC Press, 2008.

[36] D. L. Parnas. Software aging. In *International Conference on Software Engineering*, pages 279–287, 1994.

[37] D. A. Patterson and J. L. Hennessy. *Computer organization & design: the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[38] Qt. http://qt.nokia.com, 2010.

[39] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[40] S. Sharma. Real-time visualization of concurrent processes. In *Proceedings of the Joint International Conference on Vector and Parallel Processing*, pages 852–862, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[41] S. S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

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

[43] J. M. Stone. Debugging concurrent processes: a case study. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–153. ACM, 1988.

[44] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[45] C. Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, 2nd edition, 2004.

[46] R. G. Waters and E. Chikofsky. Reverse engineering: progress along many dimensions. *Communications of the ACM*, 37(5):22–25, 1994.

[47] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *4th International Workshop on Mining Software Repositories*, 2007.

[48] K. Wheeler and D. Thain. Visualizing massively multithreaded applications with threadscope. *Concurrency and Computation: Practice and Experience*, 22:45–67, 2009.

[49] S. Xie, E. Kraemer, R. E. K. Stirewalt, L. K. Dillon, and S. D. Fleming. Design and evaluation of extensions to uml sequence diagrams for modeling multithreaded interactions. *Information Visualization*, 8(2):120–136, 2009.

[50] Y. Yamaguchi and T. Itoh. Visualization of distributed processes using "data jewelry box" algorithm. *Computer Graphics International Conference*, 0:162–169, 2003.

[51] O. Zaki, E. Lusk, and D. Swider. Toward scalable performance visualization with jumpshot. *High Performance Computing Applications*, 13:277–288, 1999.