

# Drive-by Analysis of Running Programs

Wim De Pauw, Nick Mitchell, Martin Robillard\*, Gary Sevitsky, Harini Srinivasan

IBM T.J. Watson Research Center  
30 Saw Mill River Road  
Hawthorne, NY 10532 USA  
{wim,nickm,sevitsky,harini}@us.ibm.com

## 1 Introduction

Understanding the behavior of complex Java programs requires a balance of detail and volume. For example, a transaction server may perform poorly because it is not always precompiling a database query. Establishing this piece of information requires a relatively fine level of detail: e.g., the sequence, context, and duration of individual method invocations. To collect this detail, we could generate a detailed trace of the program. Unfortunately, recording detailed execution traces quickly becomes infeasible, due to both time perturbations and space overheads — infeasible for even reasonably complex programs. As a quick example, tracing IBM’s Jinsight visualizer from the outset consumes 37MB by the time the main window has appeared; this figure does not even include the tracing of any values, such as argument or return values. For a highly multithreaded server, such as IBM’s WebSphere Application Server, traces of similar time frames can easily grow to ten times this size. Yet, the other extreme, tracing only aggregate statistics (such as heap consumption, method invocation counts, each methods’ average invocation time, or an aggregate call graph [1, 2, 3]) often will not discover the root problem: why are these transactions slow, while those others are fast? Why does the program fail on some calls to one method, but not other calls to that same method?

We propose that it is not just *any* details that will help, but rather *details associated with a particular task*. In our transaction server example, at any point in time the server will be processing transactions in many threads, and doing administrative work in others; Figure 1 shows such a case. Many types of transaction may be in progress at the same time, and each transaction will most likely be in a different stage of its work. So just gathering details for the entire application for a period of time, or doing a broad filtering to include only certain classes, will still include too much detail unrelated to the problem. In our example, all we would like to see is the database activity associated with a specific transaction. In Section 2 we introduce the concept of a *burst* to represent the details associated with a task.

To analyze a program using bursts requires a methodology for choosing the criteria that define a burst. We rely on the tool user to establish these criteria. Moreover, the level of detail and the tasks of interest may vary as the user validates or disproves each hypothesis about where a problem may lie. For example: at first, the user may not even know the names of relevant routines; at this point, the user is not interested in seeing the second argument value of the fifth invocation of some method. Eventually, though, the user may need to know such fine details. Yet, by this time, the tool user may know that it is *only* that second argument of the fifth invocation that is interesting (and not any other argument of other methods).

Thus, the tool user *iteratively* explores tasks. To accomplish this interactive exploration, our solution exploits the repetitive nature common in, for example, the increasingly important class of server applications. In this paper, we outline the mechanisms and associated methodology for this style of *drive-by analysis*: the iterative process of user-assisted gathering of dynamic information from running programs.

---

\*Currently at University of British Columbia, mrobilla@cs.ubc.ca.

## 2 Drive-by Analysis

To support drive-by analysis, our system uses *directed burst tracing*. A burst is a set of trace execution information gathered during an interval of time, associated with a specific task in program. Consider our transaction server example. Using bursts, a user can direct our system to show that subset of the execution space corresponding to just those invocations which perform database operations when called from specific transaction tasks.

### 2.1 From the user perspective: direct-request-analyze-direct

The two main aspects of directed burst tracing are *requesting* and *directing* bursts. These two operations support an iterative analysis methodology. Typically, for each iteration of this cycle, a user: formulates a hypothesis, embodies this hypothesis in burst specifications, requests any number of these bursts, validates the hypothesis, and finally updates the current hypothesis.

As our solution relies on this direct-request-analyze-direct cycle, the analyzed programs must exhibit repetitive behavior. Such behavior is happily common for our domain: applications which process a large number of operations on a small set of operation types. Here are some examples: an online banking server continually processes a fixed set of transactions (e.g. buy, sell, exchange); an analysis tool reads in traces (a large number of events, but containing only a few event types).

How does the system respond to a burst request? First, the system only begins tracing when the first *triggering* method is invoked. While tracing, the system includes invocations and value information for only those *filtered* methods in the *requested* threads. Finally, terminate the burst when the end tracing criteria is encountered.

Our tool allows the user to direct the system as to triggers, filters, requested threads, and burst ending criteria. To do so, the user first specifies any number of *Class.Method*'s of interest. For each specified method, the user then associates a number of attributes. The “Filter Configuration” pane in Figure 2 shows the attributes in our current implementation. For example, one attribute indicates whether that method should trigger a burst. Another indicates whether the tracing should only trace in the triggering method’s thread (which we term “locking” to that thread).

### 2.2 Bursts, in comparison

Existing performance analysis tools do not provide an equivalent mechanism to carve out details. Some tools perform periodic or random sampling of dynamic executions. Consider the “snapshot” feature of existing Java analysis tools [1, 2]. A snapshot in these tools records instantaneous readings of either the system’s current state (e.g. the current heap size) or its state aggregated over time (e.g. number of times method A calls method B). In either case, since snapshots does not record sequence or context over an interval of time, they cannot be used to carve executions as we have described. Some snapshot-based tools [1] do allow user direction via triggers and filters; but, such direction is only of what and when to aggregate and when to take such snapshots. Other work allows for filtering of detailed execution (by selecting Classes of interest), but only over entire executions, and only for classes.

### 2.3 Live Connection

Our solution uses a live connection in order to analyze *running* programs. This aspect is critical when analyzing server systems: we want to analyze the running systems in their typical, heavily loaded, state. When analyzing a complex, distributed application, we can’t halt the system, as a traditional debugger would — this would likely cause network timeouts, bringing the system into some undesirable state. Also, it is not feasible to halt the system and restart it to validate every new hypothesis. Thus, drive-by analysis requires attaching/detaching and *reconnecting* to running servers. Further in the future we could imagine connecting to a remote customer site to analyze a server program in its production state. In this scenario, it is indeed advantageous to have a low-bandwidth and low-perturbation analysis tool.

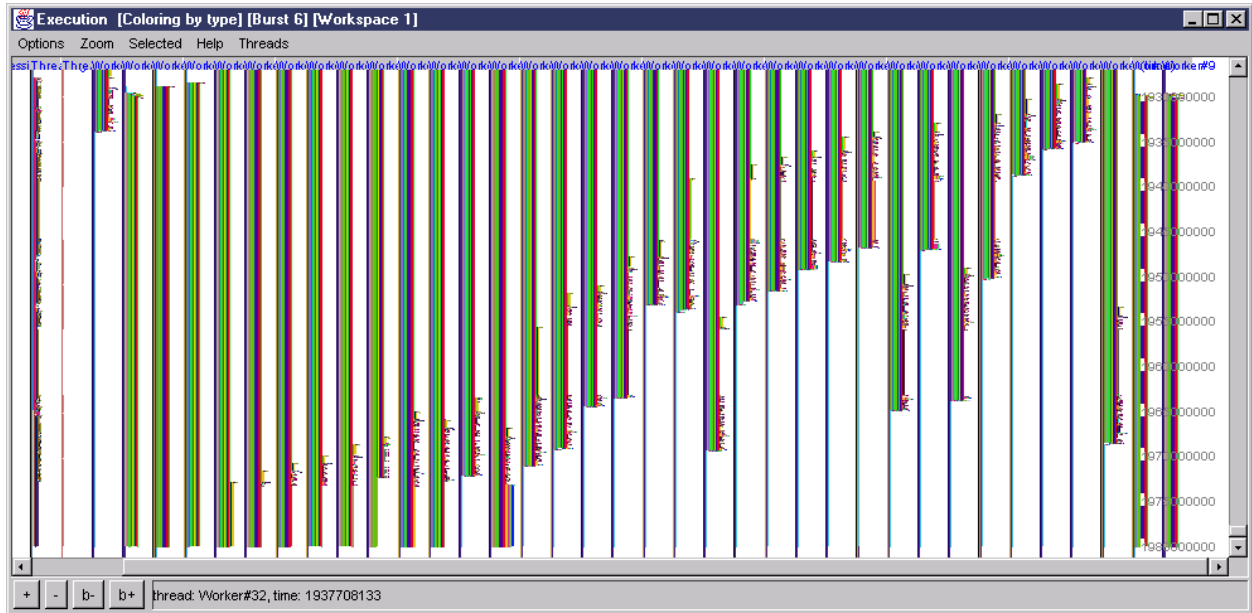


Figure 1: With complete tracing for some interval of time, we would have to collect and visualize information for every type of operation in every thread this highly multithreaded program. As seen in this figure, such complete tracing would create large traces, and highly perturb the application.

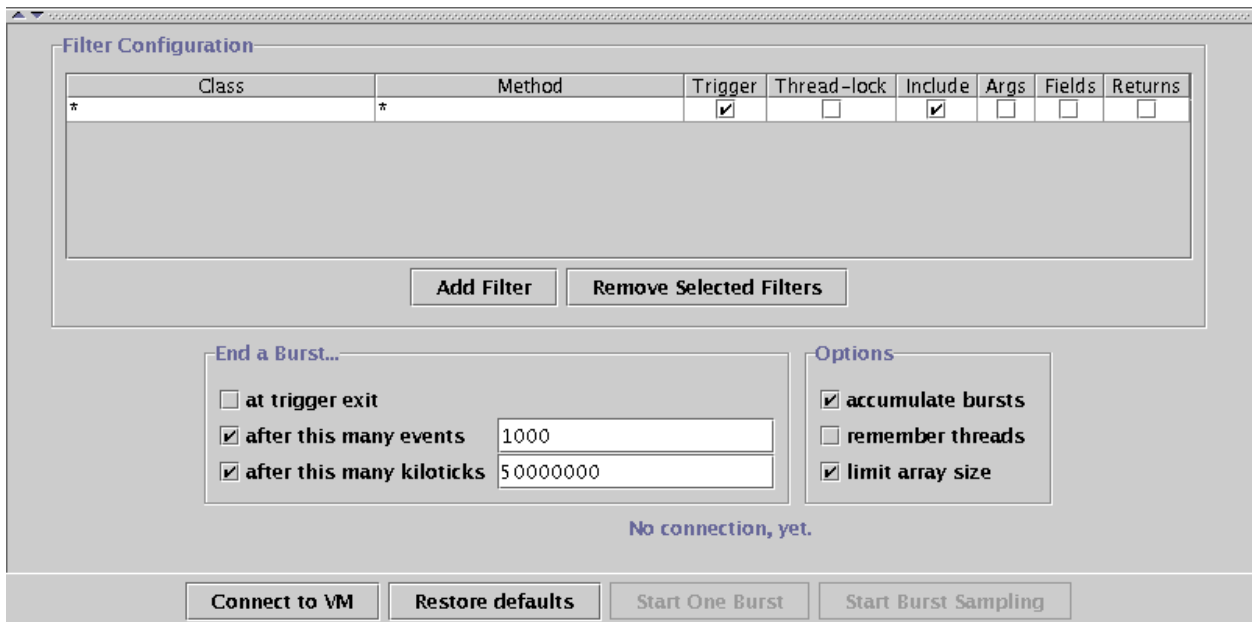


Figure 2: A user directs burst tracing using this configuration window.

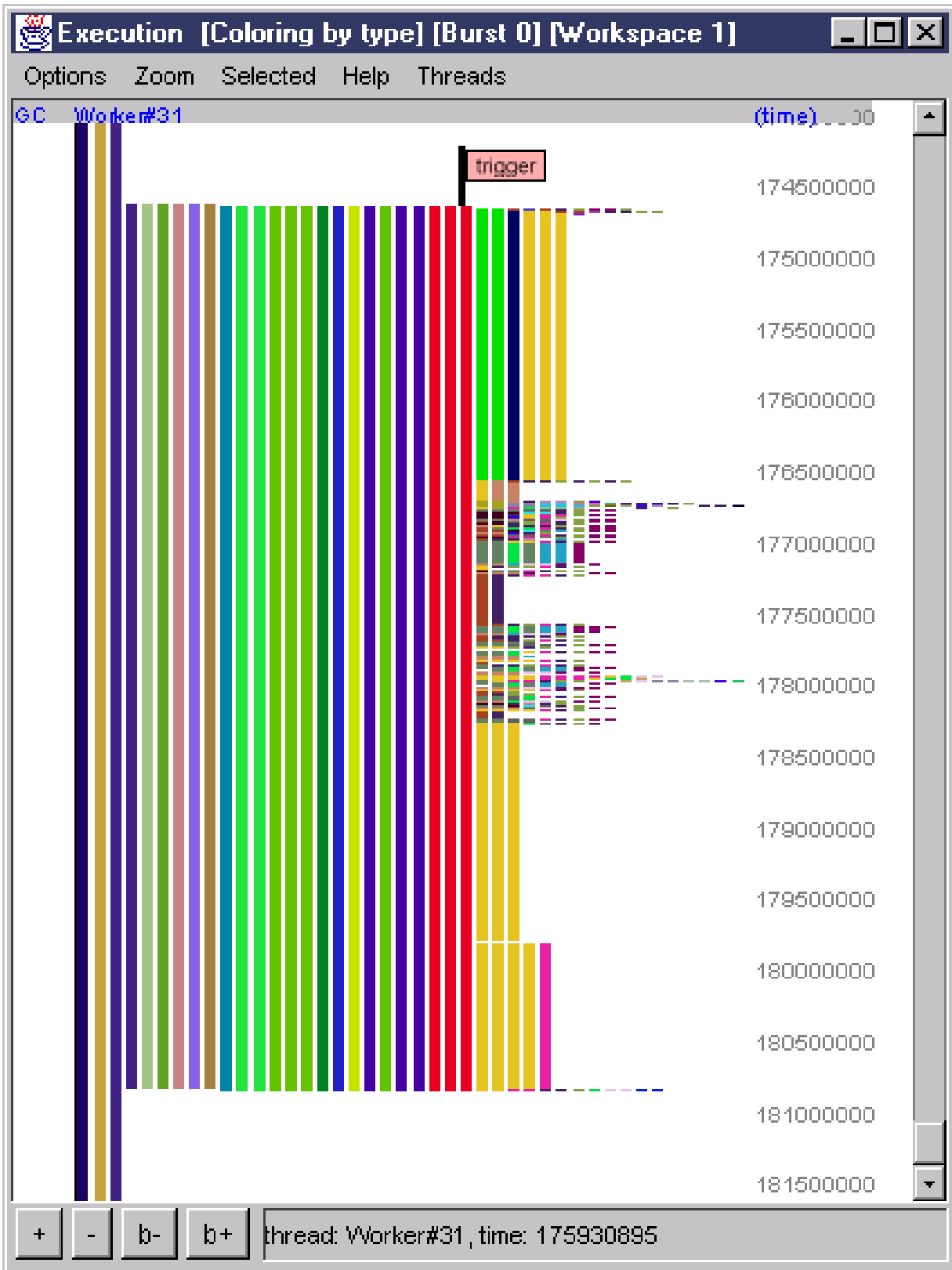


Figure 3: One burst triggered on the doGet method (this method does the work for one transaction in the transaction server being traced). The “trigger” flag marks the triggering method.

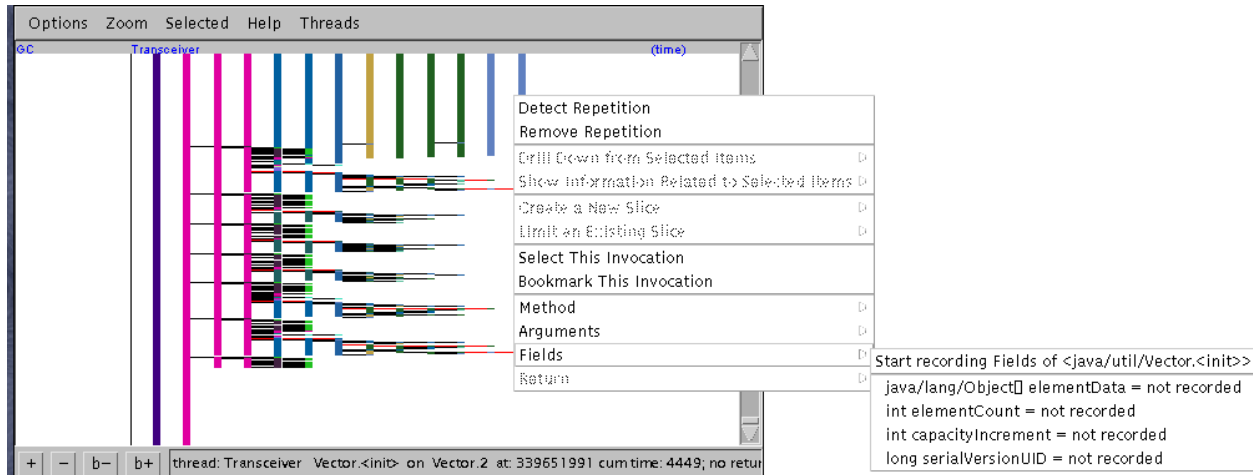


Figure 4: At this point, the user has not yet requested that field values be traced (hence the “not recorded” messages). To start tracing values of a given method, right-click on an invocation of that method and select “start tracing...”, as shown.

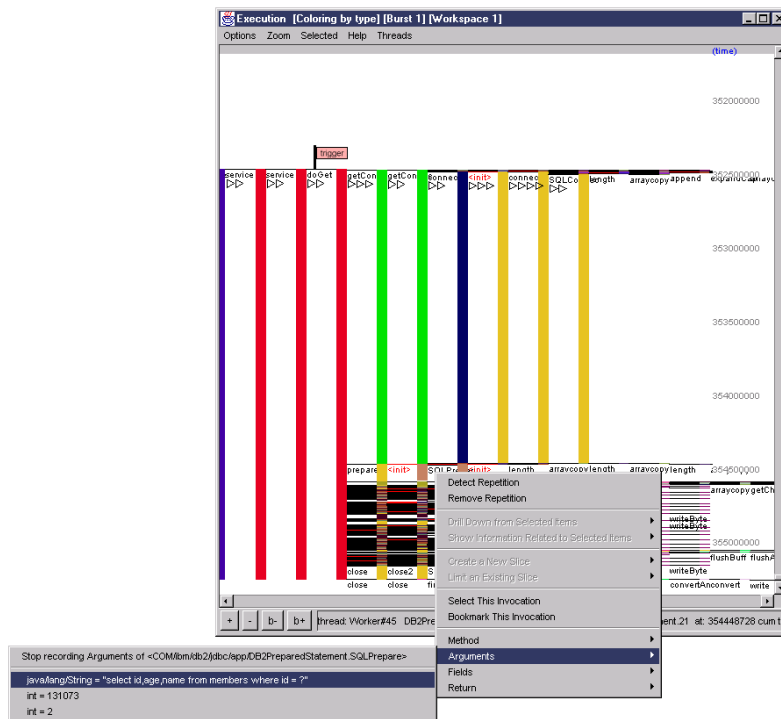


Figure 5: Once we have requested that values be traced (as in Figure 4), subsequent bursts will contain the requested values. In this case, the user has requested that some argument values be traced. This figure shows one way to visualize the trace values: right-click on an invocation and see the values of all arguments to that invocation.

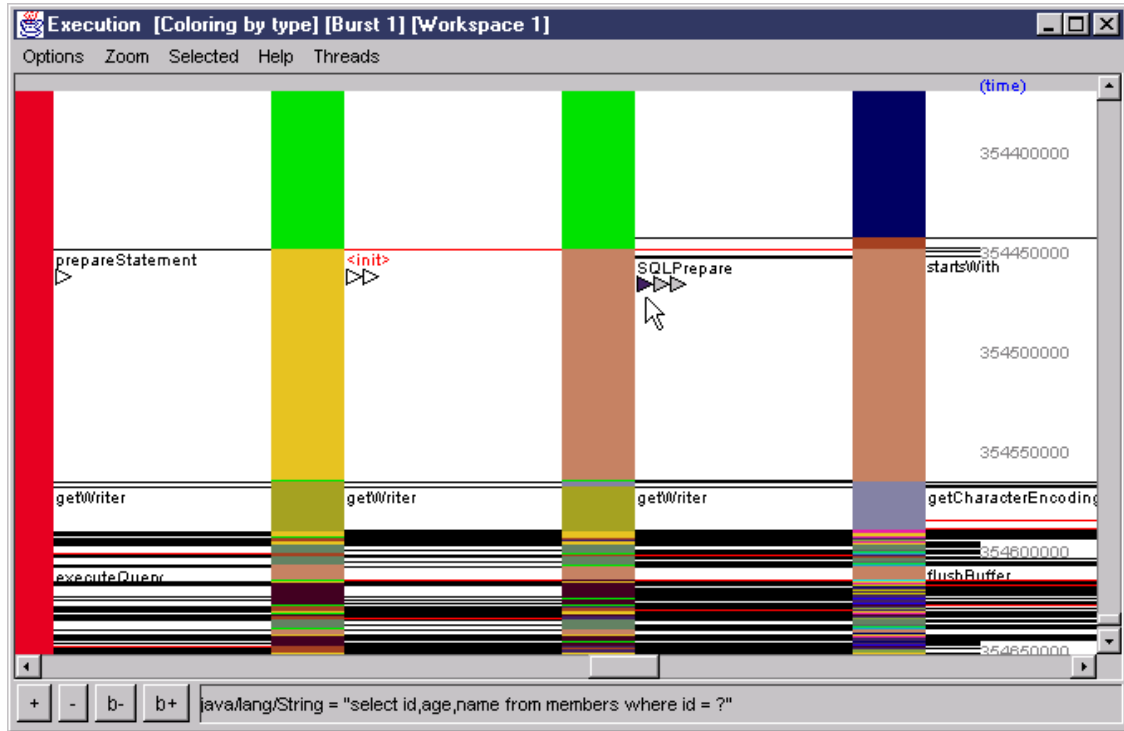


Figure 6: This figure shows an alternative way to view trace values. Each triangle in this figure represents one value. To view a value, fly over the appropriate triangle and view the value on the status line.

### 3 Acknowledgements

We greatly appreciate the assistance of Erik Jensen and Ravi Konuru.

### References

- [1] JProbe™ Profiler with Memory Debugger ServerSide Suite. <http://www.sitraka.com>.
- [2] OptimizeIt™. <http://www.intuisys.com>.
- [3] Rational Quantify™. <http://www.rational.com>.